Efficient and Effective Similar Subtrajectory Search with Deep Reinforcement Learning

Zheng Wang, Cheng Long, Gao Cong, Yiding Liu
School of Computer Science and Engineering, Nanyang Technological University, Singapore
{wang_zheng, c.long, gaocong, ydliu}@ntu.edu.sg

ABSTRACT

Similar trajectory search is a fundamental problem and has been well studied over the past two decades. However, the similar subtrajectory search (SimSub) problem, aiming to return a portion of a trajectory (i.e., a subtrajectory), which is the most similar to a query trajectory, has been mostly disregarded despite that it could capture trajectory similarity in a finer-grained way and many applications take subtrajectories as basic units for analysis. In this paper, we study the SimSub problem and develop a suite of algorithms including both exact and approximate ones. Among those approximate algorithms, two that are based on deep reinforcement learning stand out and outperform those non-learning based algorithms in terms of effectiveness and efficiency. We conduct experiments on real-world trajectory datasets, which verify the effectiveness and efficiency of the proposed algorithms.

1. INTRODUCTION

Trajectory data, which corresponds to a type of data for capturing the traces of moving objects, is ubiquitous. It has been used for various types of analysis such as clustering [1, 16, 5] and similarity search [6, 7, 44, 29, 18, 46]. The majority of existing studies take *a trajectory as a whole* for analysis [6, 7, 44, 29, 18, 46]. Motivated by the phenomenon that two trajectories could be dissimilar to each other if each is considered a whole but similar if only some portion of each is considered, there have been a few studies, which take *a portion of a trajectory* as a basic entity for analysis [1, 16, 5, 34, 35]. Some examples include subtrajectory clustering [1, 16, 5] and subtrajectory join [34, 35]. For example, the subtrajectory clustering method in [16] first partitions raw trajectories into different subtrajectories using some principle and then groups those subtrajectories that are similar to one another into clusters.

In this paper, we study a query with its goal to search for a portion of a trajectory from a database storing many trajectories called *data trajectories*, which is the most similar to a given trajectory called *query trajectory*. In this query, a portion of a trajectory, called *subtrajectory*, is considered as a basic entity and a query trajectory is taken as a whole for analysis. Therefore, it captures trajectory similarity in a finer-grained way than conventional similar trajectory search. For instance, consider a data trajectory and a query trajectory. When considered as a whole, the data trajectory is not similar to the query trajectory based on some trajectory similarity measurement, but some portion of it is very similar to the query trajectory. With the conventional similar trajectory search query, this data trajectory would be ruled out, though a portion of it is very similar to the query trajectory, which is interesting.

Moreover, in quite a few real-life applications, subtrajectories are naturally considered as basic units for analysis, e.g., subtrajectory search [32], subtrajectory join [34], subtrajectory clustering [5], etc. One application is the subtrajectory search query on sports play data. In sports such as soccer and basketball, a common practice nowadays is to track the movements of players and/or the ball using some special-purpose camera and/or GPS devices [41]. The resulting trajectory data is used to capture the semantics of the plays and for different types of data analyses. One typical task on such sports play data is to search for a portion/segment of play from a database of plays, with its trajectories of players and/or its trajectory of the ball similar to those and/or that of a given query play [32]. This task is essentially one of searching for similar subtrajectories. Another potential application is detour route detection. It first collect those routes that have been reported by passengers to be detour routes and then searches for those subtrajectories of taxis' routes, which are similar to a detour route. The found subtrajectories are probably detour routes as well.

A key problem that is involved in answering the query mentioned above is to find a subtrajectory of a data trajectory, which is the most similar to a given query trajectory. We call this problem the *similar subtrajectory search* (SimSub) problem. While there are many existing studies on the similar trajectory search problem with each trajectory considered a whole, there are very few studies on the SimSub problem. Let T be a data trajectory involving n points and T_q be a query trajectory involving m points. We

design an exact algorithm, which enumerates all possible subtrajectories of T, computes the similarity between each subtrajectory and the query trajectory, and returns the one with the greatest similarity. We further adopt an *incremental* strategy for computing the similarities that are involved in the exact algorithm, which helps to improve the time complexity by $O(n)$. We also follow some existing studies on subsequence matching [15, 49] and design an algorithm, which considers only those subtrajectories with their sizes similar to that of the query trajectory and controlled by a user parameter. This would provide a controllable trade-off between efficiency and effectiveness.

To push the efficiency further up, we propose several algorithms, which share the idea of splitting a data trajectory into some subtrajectories to be candidate solutions to the problem and differ in using different methods for splitting the data trajectory. Specifically, the process is to scan the points of a data trajectory one by one sequentially and for each one, it decides whether to split the data trajectory at the point. Some of them use pre-defined heuristics, e.g., a greedy one. Others model the process as a *markov decision process* (MDP) [26] and use deep reinforcement learning to learn an optimal policy for the MDP, which is then used for splitting the data trajectory. These splitting-based algorithms have time complexities much lower than the exact algorithm in general, e.g., for measurements such as t2vec, each splitting-based algorithm runs in $O(n)$ time while the exact algorithm runs in $O(nm)$ time.

The major contributions of this paper are as follows.

- We propose the SimSub problem, and this to the best of our knowledge, corresponds to the first systematic study on searching subtrajectories that are similar to a query trajectory. The SimSub problem relies on a trajectory similarity measurement, and in this paper, we assume an abstract one, which could be instantiated with any existing measurement.

- We develop a suite of algorithms for the SimSub problem: (1) one exact algorithm, (2) one approximate algorithm, which provides a controllable trade-off between efficiency and effectiveness, and (3) several splitting-based algorithms including both heuristics-based ones and deep reinforcement learning based ones. These algorithms should cover a wide spectrum of application scenarios in terms of efficiency and effectiveness requirements.

- Third, we conducted extensive experiments, which verified that splitting-based algorithms in general have good efficiency and among them, the algorithms based on deep reinforcement learning achieve the best effectiveness and efficiency.

Organization. We review the related work in Section 2 and provide the problem definition and some preliminaries in Section 3. Section 4 presents all non-learning based algorithms and Section 5 presents the deep reinforcement learning based algorithms. We report our experimental results in Section 6 and conclude this paper and discuss some future work in Section 7.

2. RELATED WORK

(1) Trajectory Similarity Measurements. Measuring the similarity between trajectories is a fundamental problem and has been studied extensively. Some classical solutions focus on indexing trajectories and performing similarity computation by the alignment of matching sample points. For example, DTW [46] is the first attempt at solving the local time shift issue for computing trajectory similarity. Frechet distance [2] is a classical similarity measure that treats each trajectory as a spatial curve and takes into account the location and order of the sampling points. Further, ERP [6] and EDR [7] are proposed to improve the ability to capture the spatial semantics in trajectories. However, these point-matching methods are inherently sensitive to noise and suffer from quadratic time complexity. EDS [44] and EDwP [29] are two segment-matching methods, which operate on segments for matching two trajectories. In recent years, some learning-based algorithms were proposed to speed up the similarity computation. Li et al. [18] propose to learn representations of trajectories in the form of vectors and then measure the similarity between two trajectories as the Euclidean distance between their corresponding vectors. Some other studies [39, 47, 38] define similarity measurements on trajectories based on road segments, to which the trajectories are matched. Yao et al. [45] employ deep metric learning to approximate and accelerate trajectory similarity computation. In addition, Ma et al. [20] propose a similarity measurement called p-distance for uncertain trajectories and study the problem of searching for top-k similar trajectories to a given query trajectory based on p-distance. Different specialized index techniques are developed for these similarity measures, such as DTW distance [46, 13], LCSS [37], ERP [6], EDR [7], and EDwP [29]. However, these index techniques do not generalize to other similarity measures or subtrajectory similarity search. In this paper, we assume an abstract trajectory similarity measurement, which could be instantiated with any of these existing similarity measurements and our techniques still apply.

(2) Subtrajectory Similarity Related Problems. Measuring subtrajectory similarity is also a fundamental functionality in many tasks such as clustering [1, 16, 5] and similarity join [34]. Lee et al [16] propose a general partition and group framework for subtrajectory clustering. Further, Buchin et al. [5] show the hardness of subtrajectory clustering based on Frechet distance, and Agarwal et al. [1] apply the trajectory simplification technique to approximate discrete Frechet to reduce the time cost of subtrajectory clustering. Recently, Tampakis et al. [34, 35] proposed a distributed solution for subtrajectory join and clustering by utilizing the MapReduce programming model. Although these algorithms need to consider subtrajectory similarity, similarity computation is not their focus and they usually first segment a trajectory into subtrajectories and employ an existing measure, such as Freechet distance.

(3) Subsequence (Substring) Matching. Subsequence matching is a related but different problem. It aims to find a subsequence that has the same length as the query in a given candidate sequence, which usually contains millions or even trillions [27, 28] of elements. Efficient pruning algorithms [10, 28, 27, 3, 11, 23, 9] have been proposed for the matching, and these pruning algorithms are generally designed for a specific similarity measure, such as DTW [10, 28, 27, 3, 11, 25, 14] and Euclidean distance [9, 23], and cannot generalize to other measures. On the other hand,

substring matching [15, 49] often focuses on approximate matching based on the Edit distance. It aims to find a substring in a string to best match the query. Our problem differs from the substring matching problem mainly in two aspects. First, characters in a string have exact match (0 or 1) in the alphabet; however, the points of a trajectory are different. Second, substring matching techniques are usually designed based on the characteristics of strings. e.g., grammar structure patterns, or word concurrence patterns; however, a trajectory does not have such patterns.

(4) Reinforcement Learning. The goal of reinforcement learning is to guide agents on how to take actions to maximize a cumulative reward [33] in an environment, and the environment is generally modeled as a Markov decision process (MDP) [26]. Recently, RL models have been utilized successfully to solve some database related problems. For example, Zhang et al. [48] and Li et al. [17] use RL model for automatic DBMS tuning. Trummer et al. [36] use RL to learn optimal join orders in the SkinnerDB system. Wang et al. [40] design an effective RL-based algorithm for bipartite graph matching. Overall, there are two types of popular reinforcement learning methods: (1) model-based methods [4, 12] that require to understand the environment and learn the parameters of the MDP in advance, and (2) model-free methods [43, 21] that make no efforts to learn a model and get feedback from the environment step by step. In this paper, we follow the model-free methods because they are more efficient. Specifically, we make use of a popular reinforcement learning method, namely Deep Q Network (DQN) [21], for splitting a trajectory into subtrajectories to be candidate solutions for similar subtrajectory search.

3. PROBLEM DEFINITION AND PRELIMINARIES

The trace of a moving object such as a vehicle and a mobile user is usually captured by a trajectory. Specifically, a trajectory T has its form as a sequence of time-stamped locations (called points), i.e., $T =< p_1, p_2, ..., p_n >$, where point $p_i = (x_i, y_i, t_i)$ means that the location is (x_i, y_i) at time t_i. The size of trajectory T, denoted by $|T|$, corresponds to the number of points of T.

Given a trajectory $T =< p_1, p_2, ..., p_n >$ and $1 \le i \le j \le n$, we denote by $T[i, j]$ the portion of T that starts from the ith point and ends at the jth point, i.e., $T[i, j] =< p_i, p_{i+1}, ..., p_j >$. Besides, we say that $T[i, j]$ for any $1 \le i \le j \le n$ is *subtrajectory* of T. There are in total $\frac{n(n+1)}{2}$ subtrajectories of T. Note that any subtrajectory of a trajectory T belongs to a trajectory itself.

3.1 Problem Definition

Suppose we have a database of many trajectories, which we call *data trajectories*. As discussed in Section 1, one common application scenario would be that a user has a trajectory at hand, which we call a *query trajectory* and would like to check what is the portion of the data trajectories that is the most similar to the one at his/her hand. Note that in some cases, by looking each data trajectory as whole, none is similar enough to the query trajectory, e.g., all data trajectories are relatively long while the query trajectory is relatively short.

We note that a more general query is to find the *top-k* similar subtrajectories to a query trajectory, which reduces

to the user's query as described above when $k = 1$. In this paper, we stick to the setting of $k = 1$ since extending the techniques for the setting of $k = 1$ to general settings of k is straightforward. Specifically, the techniques for the setting $k = 1$ in this paper are all based on a search process, which maintains the most similar subtrajectory found so far and updates it when a more similar subtrajectory is found during the process. These techniques could be adapted to general settings of k by simply maintaining the k most similar subtrajectories and updating them when a subtrajectory that is more similar than the kth most similar subtrajectory.

An intuitive solution to answer the user's query is to scan the data trajectories, and for each one, compute its subtrajectory that is the most similar to the query one based on some similarity measurement and update the most similar subtrajectory found so far if necessary. This solution could be further enhanced by employing indexing techniques such as the R-tree based index and the inverted-file based index for pruning [45, 39], e.g., the data trajectories that do not have any overlap with the query trajectory could usually be pruned. The key component of this solution (no matter whether indexing structures are used or not) is to compute for a given data trajectory, its subtrajectory that is the most similar to a query trajectory. We formally define the problem corresponding to this procedure as follows.

PROBLEM 1 (SIMILAR SUBTRAJECTORY SEARCH). *Given a data trajectory $T =< p_1, p_2, ..., p_n >$ and a query trajectory $T_q =< q_1, q_2, ..., q_m >$, the **similar subtrajectory search (SimSub)** problem is to find a subtrajectory of T, denoted by $T[i^*, j^*]$ ($1 \le i^* \le j^* \le n$), which is the most similar to T_q according to a trajectory similarity measurement $\Theta(\cdot, \cdot)$, i.e., $[i^*, j^*] = \arg\max_{1 \le i \le j \le n} \Theta(T[i, j], T_q)$.*

The SimSub problem relies on a similarity measurement $\Theta(T, T')$, which captures the extent to which two trajectories T and T' are similar to each other. The larger the similarity $\Theta(T, T')$ is, the more similar T and T' are. In the literature, several "dissimilarity measurements" have been proposed for $\Theta(\cdot, \cdot)$ such as DTW [46], Frechet [2], LCSS [37], ERP [6], EDR [7], EDS [44], EDwP [29], and t2vec [18]. Different measurements have different merits and suit for different application scenarios. In this paper, we assume an abstract similarity measurement $\Theta(\cdot, \cdot)$, which could be instantiated with any of these existing measurements by applying some *inverse operation* such as taking the ratio between 1 and a distance.

3.2 Trajectory Similarity Measurements

The SimSub problem assumes an abstract similarity measurement and the techniques developed could be applied to any existing measurements. Since the time complexity analysis of the algorithms proposed in this paper relies on the time complexities of computing a specific measurement in several different cases, in this part, we review three existing measurements, namely t2vec [18], DTW [46], and Frechet [2], and discuss their time complexities in different cases as background knowledge. The first two are the most widely used measurements and the last one is the most recently proposed one, which is a data-driven measurement.

We denote by Φ the time complexity of computing the similarity between a general subtrajectory of T and T_q *from scratch*, Φ_{inc} be the time complexity of computing

$\Theta(T[i,j], T_q)$ $(1 \le i < j \le n)$ *incrementally* assuming that $\Theta(T[i,j-1], T_q)$ has been computed already, and Φ_{ini} the time complexity of computing $\Theta(T[i,i], T_q)$ $(1 \le i \le n)$ from scratch since it cannot be computed incrementally. As will be discussed later, Φ_{inc} and Φ_{ini} are usually much smaller than Φ across different similarity measurements.

t2vec [18]. t2vec is a data-driven similarity measure based on deep representation learning. It adapts a sequence-to-sequence framework based on RNN [8] and takes the final hidden vector of the encoder [30] to represent a trajectory. It computes the similarity between two trajectories based on the Euclidean distance between their representations as vectors.

Given T and T_q, it takes $O(n)$ and $O(m)$ time to compute their hidden vectors, respectively and $O(1)$ to compute the Euclidean distance between two vectors [18]. Therefore, we know $\Phi = O(n + m + 1) = O(n + m)$. Since in the context studied in this paper, we need to compute the similarities between many subtrajectories and a query trajectory T_q, we assume that the representation of T_q under t2vec is computed once and re-used many times, i.e., the cost of computing the representation of T_q, which is $O(m)$, could be amortized among all computations of similarity and then that for each one could be neglected. Because of the sequence-to-sequence nature of t2vec, given the representation of $T[i, j-1]$, it would take $O(1)$ to compute that of $T[i,j]$ $(1 \le i < j \le n)$. Therefore, we know $\Phi_{inc} = O(1)$. Besides, we know $\Phi_{ini} = O(1)$ since the subtrajectory involved in the computation of similarity, i.e., $T[i,i]$ $(1 \le i \le n)$, has its size equal to 1.

DTW [46]. Given a data trajectory $T = < p_1, p_2, ..., p_n >$ and a query trajectory $T_q = < q_1, q_2, ..., q_m >$, the DTW distance is defined as below

$$D_{i,j} = \begin{cases} \sum_{h=1}^{i} d(p_h, q_1) & \text{if } j = 1 \\ \sum_{k=1}^{j} d(p_1, q_k) & \text{if } i = 1 \\ d(p_i, q_j) + \\ \min(D_{i-1,j-1}, D_{i-1,j}, D_{i,j-1}) & \text{otherwise} \end{cases} \quad (1)$$

where $D_{i,j}$ denotes the DTW distance between $T[1,i]$ and $T_q[1,j]$ and $d(p_i, q_j)$ is the distance between p_i and q_j (typically the Euclidean distance, which could be computed in $O(1)$). Consider Φ. It is clear that $\Phi = O(n \cdot m)$ since it needs to compute all pairwise distances between a point in a subtrajectory of T and a point in T_q and in general, the subtrajectory has its size of $O(n)$ and T_q has its size of m. Consider Φ_{inc}. This should be the same as the time complexity of computing $D_{i,m}$ given that $D_{i-1,m}$ has been computed. Since $D_{i-1,m}$ has been computed, we can safely assume that $D_{i-1,1}, D_{i-1,2}, ..., D_{i-1,m}$ have been computed also according to Equation 1 (note that we can always make this hold by enforcing that we compute $D_{i-1,m}$ or any other DTW distance in this way). Therefore, in order to compute $D_{i,m}$, we compute $D_{i,1}, D_{i,2}, ..., D_{i,m}$ sequentially, each of which would take $O(1)$ time with the information of $D_{i-1,k}$ $(1 \le k \le m)$ all available. That is, it takes $O(m)$ to compute $D_{i,m}$, and thus we know that $\Phi_{inc} = O(m)$. Consider Φ_{ini}. We know $\Phi_{ini} = O(m)$ since $T[i,i]$ $(1 \le i \le n)$ has its size always equal to 1 and T_q has its size of m.

Frechet [2]. Given a data trajectory $T = < p_1, p_2, ..., p_n >$ and a query trajectory $T_q = < q_1, q_2, ..., q_m >$, the Frechet

Table 1: Time complexities of computing the similarity between a subtrajectory of T and T_q in three cases

Time complexities	t2vec	DTW	Frechet
Φ (general)	$O(n+m)$	$O(n \cdot m)$	$O(n \cdot m)$
Φ_{inc} (incremental)	$O(1)$	$O(m)$	$O(m)$
Φ_{ini} (initial)	$O(1)$	$O(m)$	$O(m)$

distance is defined as below

$$F_{i,j} = \begin{cases} \max_{h=1}^{i} d(p_h, q_1) & \text{if } j = 1 \\ \max_{k=1}^{j} d(p_1, q_k) & \text{if } i = 1 \\ \max(d(p_i, q_j), \\ \min(F_{i-1,j-1}, F_{i-1,j}, F_{i,j-1})) & \text{otherwise} \end{cases} \quad (2)$$

where $F_{i,j}$ denotes the Frechet distance between $T[1,i]$ and $T_q[1,j]$ and $d(p_i, q_j)$ is the distance between p_i and q_j (typically the Euclidean distance, which could be computed in $O(1)$). When the Frechet distance is used, we have $\Phi = O(n \cdot m)$, $\Phi_{inc} = O(m)$, and $\Phi_{ini} = O(m)$, based on similar analysis as for the DTW distance.

The summary of Φ, Φ_{inc} and Φ_{ini} for the similarity measurements corresponding to the distance measurements DTW, Frechet and t2vec is presented in Table 1.

4. NON-LEARNING BASED ALGORITHMS

In this part, we introduce three types of algorithms, namely an exact algorithm ExactS, an approximate algorithm SizeS, and splitting-based algorithms including PSS, POS and POS-D. The ExactS algorithm is based on an exhaustive search with some careful implementation and has the highest complexity, the SizeS algorithm is inspired by existing studies on subsequence matching [15, 49] and provides a tunable parameter for controlling the trade-off between efficiency and effectiveness, and the splitting-based algorithms are based on the idea of splitting the data trajectory for constructing subtrajectories as candidates of the solution and run the fastest. A summary of the time complexities of these algorithms is presented in Table 2.

4.1 The ExactS Algorithm

Let T be a data trajectory and T_q be a query trajectory. The ExactS algorithm enumerates all possible subtrajectories $T[i,j]$ $(1 \le i \le j \le n)$ of the data trajectory T and computes the similarity between each $T[i,j]$ and T_q, i.e., $\Theta(T[i,j], T_q)$, and then returns the one with the greatest similarity. For better efficiency, ExactS computes the similarities between the subtrajectories and T_q *incrementally* as much as possible as follows. It involves n iterations, and in the ith iteration, it computes the similarity between each subtrajectory starting from the ith point and the query trajectory in an ascending order of the ending points, i.e., it computes $\Theta(T[i,i], T_q)$ (from scratch) first and then computes $\Theta(T[i, i+1], T_q)$, ..., $\Theta(T[i,n], T_q)$ sequentially and incrementally. During the process, it maintains the subtrajectory that is the most similar to the query one, among those that have been traversed so far. As could be verified, it would traverse all possible subtrajectories after n iterations. The ExactS algorithm with this implementation is presented in Algorithm 1.

Table 2: Time complexities of algorithms ($n_1 << n$)

Algorithms	abstract similarity measurement	t2vec	DTW	Frechet
ExactS	$O(n \cdot (\Phi_{ini} + n \cdot \Phi_{inc}))$	$O(n^2)$	$O(n^2 \cdot m)$	$O(n^2 \cdot m)$
SizeS	$O(n \cdot (\Phi_{ini} + (m + \xi) \cdot \Phi_{inc}))$	$O((\xi + m) \cdot n)$	$O((\xi + m) \cdot n \cdot m)$	$O((\xi + m) \cdot n \cdot m)$
PSS, POS, POS-D	$O(n_1 \cdot \Phi_{ini} + n \cdot \Phi_{inc})$	$O(n)$	$O(n \cdot m)$	$O(n \cdot m)$
RLS, RLS-Skip (learning-based)	$O(n_1 \cdot \Phi_{ini} + n \cdot \Phi_{inc})$	$O(n)$	$O(n \cdot m)$	$O(n \cdot m)$

Algorithm 1: ExactS

Input: A data trajectory T and query trajectory T_q;
Output: A subtrajectory of T that is the most similar to T_q;

1   $T_{best} \leftarrow \emptyset$; $\Theta_{best} \leftarrow 0$;
2   **forall** $1 \leq i \leq |T|$ **do**
3   compute $\Theta(T[i,i], T_q)$;
4   **if** $\Theta(T[i,i], T_q) > \Theta_{best}$ **then**
5       $T_{best} \leftarrow T[i,i]$; $\Theta_{best} \leftarrow \Theta(T[i,i], T_q)$;
6   **end**
7   **forall** $i + 1 \leq j \leq |T|$ **do**
8   compute $\Theta(T[i,j], T_q)$ based on $\Theta(T[i, j-1], T_q)$;
9   **if** $\Theta(T[i,j], T_q) > \Theta_{best}$ **then**
10        $T_{best} \leftarrow T[i,j]$; $\Theta_{best} \leftarrow \Theta(T[i,j], T_q)$;
11   **end**
12   **end**
13 **end**
14 **return** T_{best};

Consider the time complexity of ExactS. Since there are n iterations and in each iteration, the time complexity of computing $\Theta(T[i,i], T_q)$ is Φ_{ini} and the time complexity of computing $\Theta(T[i, i+1], T_q)$, ..., and $\Theta(T[i,n], T_q)$ is $O(n \cdot \Phi_{inc})$, we know that the overall time complexity is $O(n \cdot (\Phi_{ini} + n \cdot \Phi_{inc}))$.

We note that for some specific similarity measurement, there may exist algorithms that have better time complexity than ExactS. For example, the Spring algorithm [31], which finds the most similar subsequence of a data time series to a query one, is applicable to the SimSub problem and has the time complexity of $O(nm)$. The major idea of Spring is a dynamic programming process for computing the DTW distance between the data time series and the query one, where the latter is padded with a fictitious point that could be aligned with any point of the data time series with distance equal to 0 (so as to cover all possible suffixes of the data time series). Nevertheless, Spring is designed for the specific similarity DTW while ExactS works for an abstract one that could be instantiated to be any similarity.

4.2 The SizeS Algorithm

ExactS explores all possible $\frac{n(n+1)}{2}$ subtrajectories, many of which might be quite dissimilar from the query trajectory and could be ignored. For example, by following some existing studies on subsequence matching [15, 49], we could restrict our attention to only those subtrajectories, which have similar sizes as the query one for better efficiency. Specifically, we enumerate all subtrajectories that have their sizes within the range $[m - \xi, m + \xi]$, where $\xi \in [0, n - m]$ is a predefined parameter that controls the trade-off between the efficiency and effectiveness of the algorithm. Again, we adopt the strategy of incremental computation for the similarities

between those subtrajectories starting from the same point and the query trajectory. We call this algorithm *SizeS* and analyze its time complexity as follows. The time complexity of computing the similarities between all subtrajectories starting from a specific point and having their sizes within the range $[m-\xi, m+\xi]$ is $O(\Phi_{ini} + (m-\xi-1) \cdot \Phi_{inc} + 2\xi \cdot \Phi_{inc})$, where $\Phi_{ini} + (m - \xi - 1) \cdot \Phi_{inc}$ is cost of computing $\Theta(T[i, i+m-\xi-1], T_q)$ and $2\xi \cdot \Phi_{inc}$ is the cost of computing $\Theta(T[i,j], T_q)$ for $j \in [i + m - \xi, i + m + \xi - 1]$. It could be further reduced to $O(\Phi_{ini} + (m + \xi) \cdot \Phi_{inc})$. Therefore, the overall time complexity of SizeS is $O(n \cdot (\Phi_{ini} + (m + \xi) \cdot \Phi_{inc}))$. For example, when DTW or Frechet is used, it is $O(n \cdot (m + (m + \xi) \cdot m)) = O((\xi + m) \cdot n \cdot m)$ and when t2vec is used, it is $O(n \cdot (1 + (\xi + m) \cdot 1)) = O((\xi + m) \cdot n)$.

In summary, SizeS achieves a better efficiency than ExactS at the cost of its effectiveness. Besides, SizeS still needs to explore $O(\xi \cdot n)$ subtrajectories, which restricts its application on small and moderate datasets only. Unfortunately, SizeS may return a solution, which is arbitrarily worse than the best one. We illustrate this in the technical report version [42] due to the page limit.

4.3 Splitting-based Algorithms

The ExactS algorithm is costly since it explores $O(n^2)$ subtrajectories. The SizeS algorithm runs faster than ExactS since it explores about $O(\xi \cdot n)$ subtrajectories ($\xi << n$). Thus, an intuitive idea to push the efficiency further up is to explore fewer subtrajectories. In the following, we design a series of three approximate algorithms, which all share the idea of splitting a data trajectory into several subtrajectories and returning the one that is the most similar to the query trajectory. These algorithms differ from each other in using different heuristics for deciding where to split the data trajectory. With this splitting strategy, the number of subtrajectories that would be explored is bounded by n and in practice, much smaller than n. We describe these algorithms as follows.

(1) Prefix-Suffix Search (PSS). The PSS algorithm is a greedy one, which maintains a variable T_{best} storing the subtrajectory that is the most similar to the query trajectory found so far. Specifically, it scans the points of the data trajectory T in the order of $p_1, p_2, ..., p_n$. When it scans p_i, it computes the similarities between the two subtrajectories that would be formed if it splits T at p_i, i.e., $T[h, i]$ and $T[i, n]$, and the query trajectory T_q, where p_h is the point following the one, at which the last split was done if any and p_h is the first point p_1 otherwise. In particular, we replace the part of computing the similarity between the suffix $T[i, n]$ and the query trajectory with that between their reversed versions, denoted by $T[i,n]^R$ and T_q^R, respectively. This is because (1) $\Theta(T[i,n]^R, T_q^R)$ could be computed incrementally based on $\Theta(T[i+1, n]^R, T_q^R)$ and (2) $\Theta(T[i,n]^R, T_q^R)$ and $\Theta(T[i,n], T_q)$ are equal for some sim-
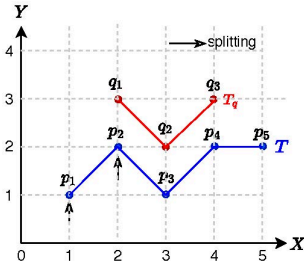
Figure 1: A problem input.

Table 3: Illustration of PSS with the DTW distance.

Initial	$h = 1$, $T_{best} = \emptyset$ and $\Theta_{best} = 0$					
Point	Prefix	Suffix	Split	h	Θ_{best}	T_{best}
p_1	$\Theta(T[1,1], T_q) = 0.124$	$\Theta(T[1,5]^R, T_q^R) = 0.150$	Yes	2	0.150	$T[1,5]$
p_2	$\Theta(T[2,2], T_q) = 0.236$	$\Theta(T[2,5]^R, T_q^R) = 0.227$	Yes	3	0.236	$T[2,2]$
p_3	$\Theta(T[3,3], T_q) = 0.183$	$\Theta(T[3,5]^R, T_q^R) = 0.215$	No	3	0.236	$T[2,2]$
p_4	$\Theta(T[3,4], T_q) = 0.236$	$\Theta(T[4,5]^R, T_q^R) = 0.215$	No	3	0.236	$T[2,2]$
p_5	$\Theta(T[3,5], T_q) = 0.215$	$\Theta(T[5,5]^R, T_q^R) = 0.152$	No	3	0.236	$T[2,2]$
Output	$T_{best} = T[2,2]$ with $\Theta_{best} = 0.236$					

ilarity measurements such as DTW and Frechet and positively correlated for others such as t2vec as we found via experiments. If any of these two similarities are larger than the best-known similarity, it performs a split operation at p_i and updates T_{best} accordingly; otherwise, it continues to scan the next point p_{i+1}. At the end, it returns T_{best}. The procedure of PSS is presented in Algorithm 2.

To illustrate, consider an example shown in Figure 1, where T is a data trajectory with 5 points $p_{1:5}$ and T_q is a query trajectory with 3 points $q_{1:3}$. Suppose that we measure the similarity between two trajectories using the ratio of 1 over the DTW distance between them. Consider the process of the PSS algorithm, which is depicted in Table 3. When it scans p_1, it considers two subtrajectories, namely $T[1,1]$ and $T[1,5]$, which have the similarities, i.e., 0.124 and 0.150, both larger than the best-known so far, i.e., 0. Therefore, it performs a split operation at p_1 and updates h, Θ_{best} and T_{best} accordingly, as shown in the 3rd row of the table. It continues to scan p_2 and considers two subtrajectories $T[2,2]$ and $T[2,5]$. Since their similairities are larger than the best-known one, it performs a split at p_2 and updates h, Θ_{best} and T_{best}. It then scans p_3, p_4, and p_5 and does not perform a split at any of them. Therefore, h, Θ_{best} and T_{best} are kept unchanged. Finally, it returns T_{best}, i.e., $T[2,2]$, which has the similarity of 0.236. Note that $T[2,4]$ has the largest similarity to T_q, which is $1/3 = 0.333$.

We analyze the time complexity of PSS as follows. When it scans a specific point p_i, the time costs include that of computing $\Theta(T[h,i], T_q)$ and also that of computing $\Theta(T[i,n]^R, T_q^R)$. Consider the former part. If $i = h$, it is Φ_{ini}. If $i \geq h+1$, it is Φ_{inc} since $\Theta(T[h,i], T_q)$ could be computed based on $\Theta(T[h,i-1], T_q)$ incrementally. Consider the latter part. It is simply $O(\Phi_{inc})$. In conclusion, the time complexity of PSS is $O(n_1 \cdot (\Phi_{ini} + \Phi_{inc}) + (n - n_1) \cdot \Phi_{inc}) = O(n_1 \cdot \Phi_{ini} + n \cdot \Phi_{inc})$, where n_1 is the number of points where splits are done. For example, when DTW or Frechet is used, the time complexity of PSS is $O(n_1 \cdot m + n \cdot m) = O(n \cdot m)$ and when t2vec is used, it is $O(n_1 \cdot 1 + n \cdot 1) = O(n)$.

(2) Prefix-Only Search (POS). In PSS, when it scans a point p_i, it considers two subtrajectories, namely $T[h,i]$ and $T[i,n]$. An alternative is to consider the prefix $T[h,i]$ only - one argument is that the suffix $T[i,n]$ might be destroyed when further splits are conducted. A consequent benefit is that the time cost of computing $\Theta(T[i,n], T_q)$ would be saved. We call this algorithm the POS algorithm. As could be verified, POS has the same time complexity as PSS though the former runs faster in practice.

(3) Prefix-Only Search with Delay (POS-D). POS performs a split operation whenever a prefix, which is bet-

---

Algorithm 2: Prefix-Suffix Search (PSS)

Input: A data trajectory T and query trajectory T_q;
Output: A subtrajectory of T that is similar to T_q;

1   $T_{best} \leftarrow \emptyset$; $\Theta_{best} \leftarrow 0$;
2   compute $\Theta(T[n,n]^R, T_q^R)$;
3   compute $\Theta(T[n-1,n]^R, T_q^R)$, $\Theta(T[n-2,n]^R, T_q^R)$, ..., $\Theta(T[1,n]^R, T_q^R)$ incrementally;
4   $h \leftarrow 1$;
5   **forall** $1 \leq i \leq |T|$ **do**
6   compute $\Theta(T[h,i], T_q)$ incrementally if possible;
7   **if** $\max\{\Theta(T[h,i], T_q), \Theta(T[i,n]^R, T_q^R)\} > \Theta_{best}$ **then**
8       $\Theta_{best} \leftarrow \max\{\Theta(T[h,i], T_q), \Theta(T[i,n]^R, T_q^R)\}$;
9   **if** $\Theta(T[h,i], T_q) > \Theta(T[i,n]^R, T_q^R)$ **then**
10        $T_{best} \leftarrow T[h,i]$;
11   **else**
12        $T_{best} \leftarrow T[i,n]$;
13   **end**
14       $h \leftarrow i + 1$;
15   **end**
16 **end**
17 **return** T_{best};

---

ter than the best subtrajectory known so far, is found. This looks a bit rush and may prevent a better subtrajectory to be formed by extending it with a few more points. Thus, we design a variant of POS, called *Prefix-Only Search with Delay* (POS-D). Whenever a prefix is found to be more similar to the query trajectory than the best subtrajectory known so far, POS-D continues to scan D more points and splits at one of these $D + 1$ points, which has the corresponding prefix the most similar to the query trajectory. It could be verified that with this delay mechanism, the time complexity of the algorithm does not change though in practice, it would be slightly higher.

While these splitting-based algorithms including PSS, POS and POS-D, return reasonably good solutions in practice, they may return solutions that are arbitrarily worse than the best one in theory. We illustrate this in the technical report version [42] due to the page limit.

5. REINFORCEMENT LEARNING BASED ALGORITHM

A splitting-based algorithm has its effectiveness rely on the quality of the process of splitting a data trajectory. In order to find a solution of high quality, it requires to perform split operations at appropriate points such that some subtrajectories that are similar to a query trajectory are

formed and then explored. The three splitting-based algorithms, namely PSS, POS and POS-D, mainly use some hand-crafted heuristics for making decisions on whether to perform a split operation at a specific point. This process of splitting a trajectory into subtrajectories is a typical *sequential decision making* process. Specifically, it scans the points sequentially and for each point, it makes a decision on whether or not to perform a split operation at the point. In this paper, we propose to model this process as a *Markov decision process* (MDP) [26] (Section 5.1), adopt a *deep-Q-network* (DQN) [21] for learning an optimal policy for the MDP (Section 5.2), and then develop an algorithm called *reinforcement learning based search* (RLS), which corresponds to a splitting-based algorithm that uses the learned policy for the process of splitting a data trajectory (Section 5.3) and an augmented version of RLS, called RLS-Skip, with better efficiency (Section 5.4)

5.1 Trajectory Splitting as a MDP

A MDP consists of four components, namely *states*, *actions*, *transitions*, and *rewards*, where (1) a state captures the *environment* that is taken into account for decision making by an *agent*; (2) an action is a possible decision that could be made by the agent; (3) a transition means that the state changes from one to another once an action is taken; and (4) a reward, which is associated with a transition, corresponds to some feedback indicating the quality of the action that causes the transition. We model the process of splitting a data trajectory as a MDP as follows.

(1) States. We denote a state by s. Suppose it is currently scanning point p_t. p_h denotes the point following the one, at which the last split operation happens if any and p_1 otherwise. We define the state of the current environment as a triplet $(\Theta_{best}, \Theta_{pre}, \Theta_{suf})$, where Θ_{best} is the largest similarity between a subtrajectory found so far and the query trajectory T_q, Θ_{pre} is $\Theta(T[h,t], T_q)$ and Θ_{suf} is $\Theta(T[t,n]^R, T_q^R)$. As could be noticed, a state captures the information about the query trajectory, the data trajectory, the point at which the last split happens, and the point that is being scanned, etc. Note that the state space is a three-dimensional continuous one.

(2) Actions. We denote an action by a. We define two actions, namely $a = 1$ and $a = 0$. The former means to perform a split operation at the point that is being scanned and the latter means to move on to scan the next point.

(3) Transitions. In the process of splitting a trajectory, given a current state s and an action a to take, the probability that we would observe a specific state s' is unknown. We note that the method that we use for solving the MDP in this paper is a *model-free* one and could solve the MDP problem even with its transition information unknown.

(4) Rewards. We denote a reward by r. We define the reward associated with the transition from state s to state s' after action a is taken as $(s'.\Theta_{best} - s.\Theta_{best})$, where the $s'.\Theta_{best}$ is the first component of state s' and $s.\Theta_{best}$ is the first component of state s. With this reward definition, the goal of the MDP problem, which is to maximize the accumulative rewards, is consistent with that of the process of splitting a data trajectory, which is to form a subtrajectory with the greatest possible similarity to the query trajectory. To see this, consider that the process goes through a sequence of states $s_1, s_2, ..., s_N$ and ends at s_N. Let $r_1, r_2, ..., r_{N-1}$

denote the rewards received at these states except for the termination state s_N. Then, when the future rewards are not discounted, we have

$$\Sigma_t r_t = \Sigma_t(s_t.\Theta_{best} - s_{t-1}.\Theta_{best}) = s_N.\Theta_{best} - s_1.\Theta_{best}$$

where $s_N.\Theta_{best}$ corresponds to the similarity between the best subtrajectory found and the query trajectory T_q and $s_1.\Theta_{best}$ corresponds to the best known similarity at the beginning, i.e., 0. Therefore, maximizing the accumulative rewards is equivalent to maximizing the similarity between the subtrajectory to be found and T_q in this case.

5.2 Deep-Q-Network (DQN) Learning

The core problem of a MDP is to find an optimal *policy* for the agent, which corresponds to a function π that specifies the action that the agent should choose when at a specific state so as to maximize the accumulative rewards. One type of methods that are commonly used is those *value-based* methods [33, 21]. The major idea is as follows. First, it defines an optimal action-value function $Q^*(s, a)$ (or Q function), which represents the maximum amount of expected accumulative rewards it would receive by following any policy after seeing the state s and taking the action a. Second, it estimates $Q^*(s, a)$ using some methods such as Q-learning [43] and deep-Q-network (DQN) [21]. Third, it returns the policy, which always chooses for a given state s the action a that maximizes $Q^*(s, a)$.

In our MDP, the state space is a three dimensional continuous one, and thus we adopt the DQN method. Specifically, we use the *deep Q learning with replay memory* [22] for learning the Q functions. This method maintains two neural networks. One is called the *main network* $Q(s, a; \theta)$, which is used to estimate the Q function. The other is called the *target network* $\hat{Q}(s, a; \theta^-)$, which is used to compute some form of loss for training the main network. Besides, it maintains a fixed-size pool called *replay memory*, which contains the latest transitions that are sampled uniformly and used for training the main network. The intuition is to avoid the correlation among consecutive transitions. The detailed procedure of DQN for our MDP is presented in Algorithm 3, which we go through as follows. We maintain a database $\mathcal{D}$ of data trajectories and a set of $\mathcal{D}_q$ of query trajectories. It first initializes the reply memory $\mathcal{M}$ with some capacity, the main network $Q(s, a; \theta)$ with random weights, and the target network $\hat{Q}(s, a; \theta^-)$ by copying $Q(s, a; \theta)$ (Lines 1 - 3). Then, it involves a sequence of many episodes. For each episode, it samples a data trajectory T from $\mathcal{D}$ and a query trajectory T_q from $\mathcal{D}_q$, both uniformly (Lines 4 - 5). It initializes a variable h such that p_h corresponds to the point following the one, at which the last split operation is performed if any and p_1 otherwise (Line 6). It also initializes the state s_1 (Lines 7 - 8). Then, it proceeds with $|T|$ time steps. At the tth time step, it scans point p_t and selects an action using the ϵ-greedy strategy based on the main network, i.e., it performs a random action a_t with the probability ϵ ($0 < \epsilon < 1$) and $a_t = \arg\max_a Q(s_t, a; \theta)$ with the probability $(1 - \epsilon)$ (Lines 9 - 10). If $a_t = 1$, it splits the trajectory at point p_t and updates h to be $t + 1$ (Lines 11 - 13). It then updates Θ_{best} if possible (Line 14). If the current point being scanned is the last point p_n, it terminates (Lines 15 - 17). Otherwise, it observes a new state s_{t+1} and the reward r_t (Lines 18 - 20). It then stores the experience (s_t, a_t, r_t, s_{t+1}) in the reply memory, samples a minibatch of

Algorithm 3: Deep-Q-Network (DQN) Learning with Experience Replay

Input: A database $\mathcal{D}$ of data trajectories and a set of $\mathcal{D}_q$ of query trajectories;
Output: Learned action-value function $Q(s, a; \theta)$;

1   initialize the reply memory M;
2   initialize the main network $Q(s, a; \theta)$ with random weights θ;
3   initialize the target network $\hat{Q}(s, a; \theta^-)$ with weights $\theta^- = \theta$;
4   **for** $episode = 1, 2, 3, ...$ **do**
5   sample a data and query trajectory T, T_q;
6     $h \leftarrow 1$;
7     $\Theta_{best} \leftarrow 0; \Theta_{pre} \leftarrow \Theta(T[h, h], T_q)$;
      $\Theta_{suf} \leftarrow \Theta(T[h, n]^R, T_q^R)$;
8   observe the first state $s_1 = (\Theta_{best}, \Theta_{pre}, \Theta_{suf})$;
9   **for** *each step* $1 \leq t \leq |T|$ **do**
10   select a random action a_t with probability ϵ and select action $a_t = \arg\max_a Q(s_t, a; \theta)$ with probablity $(1 - \epsilon)$;
11   **if** $a_t = 1$ **then**
12         $h \leftarrow t + 1$;
13   **end**
14       $\Theta_{best} \leftarrow \max\{s_t.\Theta_{best}, s_t.\Theta_{pre}, s_t.\Theta_{suf}\}$;
15   **if** $t = |T|$ **then**
16   **break**;
17   **end**
18       $\Theta_{pre} \leftarrow \Theta(T[h, t + 1], T_q)$;
        $\Theta_{suf} \leftarrow \Theta(T[t + 1, n]^R, T_q^R)$;
19   observe the next state $s_{t+1} = (\Theta_{best}, \Theta_{pre}, \Theta_{suf})$;
20   observe the reward $r_t = s_{t+1}.\Theta_{best} - s_t.\Theta_{best}$;
21   store the experience (s_t, a_t, r_t, s_{t+1}) in the replay memory $\mathcal{M}$;
22   sample a random minibatch of experiences from $\mathcal{M}$ uniformly;
23   perform a gradient descent step on the loss as computed by Equatioin (3) wrt θ;
24   **end**
25   copy the main network $Q(s, a; \theta)$ to $\hat{Q}(s, a; \theta^-)$;
26 **end**

experiences, and uses it to perform a gradient descent step for updating θ wrt a loss function (Lines 21 - 23). The loss function for one experience (s, a, r, s') is as follows.

$$L(\theta) = (y - Q(s, a; \theta))^2 \qquad (3)$$

where y is equal to r if s' is a termination step and $r + \gamma \cdot \max_{a'} \hat{Q}(s', a'; \theta^-)$ otherwise. Finally, it updates the target network $\hat{Q}(s, a; \theta^-)$ with the main network $Q(s, a; \theta)$ at the end of each episode (Line 25). A graphical illustration of the method is shown in Figure 2.

5.3 Reinforcement Learning based Search Algorithm (RLS)

Once we have estimated the Q functions $Q(s, a; \theta)$ via the deep Q learning with experience replay, we use the policy, which always takes for a given state s the action that maximizes $Q(s, a; \theta)$, for the process of splitting a data trajectory. Among all subtrajectories that are formed as a result of the process, we return the one with the greatest similarity to the query trajectory T_q. We call this algorithm *reinforcement learning based search* (RLS). Essentially, it is the same
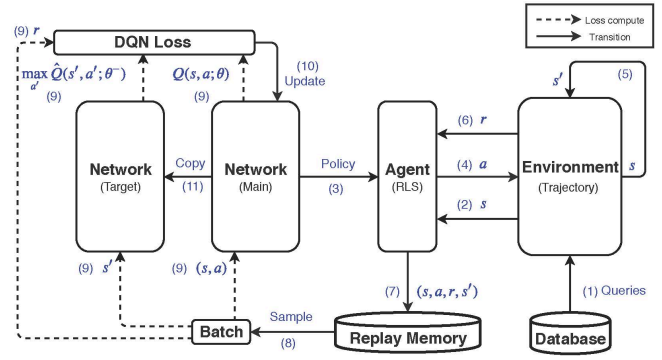

Figure 2: Deep Q learning with experience replay

as PSS except that it uses a policy learned via DQN instead of human-crafted heuristics for making decisions on how to split a data trajectory.

RLS has the same time complexity as PSS since both RLS and PSS make decisions based on the similarities of the subtrajectories that are being considered when scanning a point and the best-known similarity: (1) RLS constructs a state involving them and goes through the main network of DQN with the state information, which is $O(1)$ given that the network is small-size (e.g., a few layers); and (2) PSS simply conduct some comparisons among the similarities, which is also $O(1)$. In terms of effectiveness, RLS provides consistently better solutions than PSS as well as POS and POS-D, as will be shown in the empirical studies, and the reason is possibly that RLS is based on a learned policy, which makes decision more intelligent than simple heuristics that are human-crafted.

5.4 Reinforcement Learning based Search with Skipping (RLS-Skip)

In the RLS algorithm, each point is considered as a candidate for performing a split operation. While this helps to attain a reasonably large space of subtrajectories for exploration and hence achieving good effectiveness, it is somehow conservative and incurs some cost of decision marking for *each* point. An alternative is to go a bit more optimistic and *skip* some points from being considered as places for split operations. The benefit would be immediate, i.e., the cost of making decisions at these points is saved. Motivated by this, we propose to augment the MDP that is used by RLS by introducing k more actions (apart from two existing ones: scanning the next point and performing a split operation), namely skipping 1 point, skipping 2 points, ..., skipping k points. Here, k is a hyperparameter, and by skipping j points $(j = 1, 2, ..., k)$, it means to skip points $p_{i+1}, p_{i+2}, ..., p_{i+j}$ and scan point p_{i+j+1} next, where p_i is the point that is being scanned. All other components of the MDP are kept the same as that for RLS. Note that when $k = 0$, this MDP reduces to the original one for RLS. We call the algorithm based on this augmented MDP *RLS-Skip*.

To illustrate, consider again the example shown in Figure 1. Suppose that it has learned a policy using the DQN method, which is captured by the main network $Q(s, a; \theta)$. The process of RLS-Skip is depicted in Table 4. Suppose the parameter k is equal to 1, which implies that there are three possible actions 0 (no split), 1 (split), and 2 (no split and skip of 1 point). In addition, we write $\Theta_{pre} = \Theta(T[i, j], T_q)$

Table 4: Illustration of RLS-Skip with the DTW distance.

Initial	$h = 1$, $T_{best} = \emptyset$ and $\Theta_{best} = 0$				
Point	State	Action	h	Θ_{best}	T_{best}
p_1	$s_1 = (\Theta_{best} = 0, \Theta_{pre=T[1,1]} = 0.124, \Theta_{suf=T[1,5]} = 0.150)$	$a_1 = \arg\max_a Q(s_1, a; \theta) = 1 : split$	2	0.150	$T[1,5]$
p_2	$s_2 = (\Theta_{best} = 0.150, \Theta_{pre=T[2,2]} = 0.236, \Theta_{suf=T[2,5]} = 0.227)$	$a_2 = \arg\max_a Q(s_2, a; \theta) = 2 : skip$	2	0.236	$T[2,2]$
p_3 (skipped)	-	-	-	-	-
p_4	$s_3 = (\Theta_{best} = 0.236, \Theta_{pre=T[2,4]} = 0.333, \Theta_{suf=T[4,5]} = 0.215)$	$a_3 = \arg\max_a Q(s_3, a; \theta) = 1 : split$	5	0.333	$T[2,4]$
p_5	$s_4 = (\Theta_{best} = 0.333, \Theta_{pre=T[5,5]} = 0.152, \Theta_{suf=T[5,5]} = 0.152)$	$a_4 = \arg\max_a Q(s_4, a; \theta) = 0 : no-split$	5	0.333	$T[2,4]$
Output	$T_{best} = T[2,4]$ with $\Theta_{best} = 0.333$				

as $\Theta_{pre=T[i,j]}$ and $\Theta_{suf} = \Theta(T[i,j], T_q)$ as $\Theta_{suf=T[i,j]}$ for simplicity. At the very beginning, it initializes h, Θ_{best} and T_{best}. It then scans point p_1, observes the first state s_1 as $(\Theta_{best} = 0, \Theta_{pre=T[1,1]} = 0.124, \Theta_{suf=T[1,5]} = 0.150)$ and finds the action $a_1 = \arg\max_a Q(s_1, a; \theta) = 1$, meaning to perform a split operation at p_1. It then updates h, Θ_{best}, and T_{best} as shown in the 3rd row of the table. It continues to scan point p_2, observes the second state s_2 as $(\Theta_{best} = 0.150, \Theta_{pre=T[2,2]} = 0.236, \Theta_{suf=T[2,5]} = 0.227)$ and finds the action $a_2 = \arg\max_a Q(s_2, a; \theta) = 2$, meaning to skip the next 1 point, i.e., p_3. It keeps h unchanged (since no splits are done) but updates Θ_{best} and T_{best} to be 0.236 and $T[2,2]$, respectively, since $T[2,2]$ is the subtrajectory with the largest similarity among all subtrajectories that have been considered. As a result of the skipping, it scans point p_4 next and proceeds similarly. It performs a split operation when scanning point p_4 and terminates after scanning point p_5. At the end, it returns $T[2,4]$, which has the similarity of 0.333.

While the cost of making decisions at those points that are skipped (i.e., that of going through the main network of the DQN) could be saved in RLS-Skip, the cost of constructing the states at those points that are not skipped would be more or less that of constructing the states at *all* points since the state at a point involves some similarities, which are computed incrementally based on the similarities computed at those points before the point. Thus, by applying the skipping strategy alone would not help much in reducing the time cost since the cost of maintaining the states dominates that of making decisions. To fully unleash the power of the skipping strategy, we propose to ignore those points that have been skipped when maintaining the states. That is, to maintain the state $(\Theta_{best}, \Theta_{pre}, \Theta_{suf})$ at a point p_i, we compute Θ_{best} and Θ_{suf} in the same way as we do in RLS and Θ_{pre} as the similarity between the query trajectory and the subtrajectory consisting of those points that are before p_i and *have not been skipped*. Here, the prefix subtrajectory corresponds to a *simplification* of that used in RLS [19]. While RLS-Skip has the same worse-case time complexity as RLS, e.g., it reduces to RLS when no skipping operations happen, the cost of maintaining the states for RLS-Skip would be much smaller. As shown in our empirical studies, RLS-Skip runs significantly faster than RLS as well as PSS, POS and POS-D. In addition, RLS-Skip and RLS do not provide theoretical guarantees on the approximation quality due to their learning nature. The proofs can be found in the appendix of the technical report [42]. Nevertheless, they work quite well in practice (e.g., RLS has the approximation ratio smaller than 1.1 for all similarity measurements and on all datasets (Figure 3)). In addition, the problem instances that we constructed for proving the negative results in fact rarely happen in practice, which are confirmed by the effectiveness results on real datasets.

6. EXPERIMENTS

We present the experimental set-up in Section 6.1 and then the experimental results in Section 6.2.

6.1 Experimental Setup

Dataset. Our experiments are conducted on three real-world trajectory datasets. The first dataset, denoted by Porto, is collected from the city of Porto [1], Portugal, which consists around 1.7 million taxi trajectories over 18 months with a sampling interval of 15 seconds and a mean length around 60. The second dataset, denoted by Harbin, involves around 1.2 million taxi trajectories collected from 13,000 taxis over 8 months in Harbin, China with non-uniform sampling rates and a mean length around 120. The third dataset, denoted by Sports, involves around 0.2 million soccer player and ball trajectories collected from STATS Sports [2] with a uniform sampling rate of 10 times per second and a mean length around 170.

Parameter Setting. For training t2vec model, we follow the original paper [18] by excluding those trajectories that are short and use their parameter settings. For SizeS, we use the setting $\xi = 5$ (with the results of its effect shown later on). For POS-D, we vary the parameter D from 4 to 7, and since the results are similar, we use the setting $D = 5$. For the neural networks involved in the RL-based algorithms, i.e., RLS and RLS-Skip, we use a feedforward neural network with 2 layers. In the first layer, we use the ReLu function with 20 neurons, and in the second layer, we use the sigmoid function with $2 + k$ neurons as the output corresponding to different actions, where for RLS we use $k = 0$ and for RLS-Skip, we use $k = 3$ by default. In the training process, the size of replay memory $\mathcal{M}$ is set at 2000. We train our model on 25k random trajectory pairs, using Adam stochastic gradient descent with an initial learning rate of 0.001. The minimal ϵ is set at 0.05 with decay 0.99 for the ϵ-greedy strategy, and the reward discount rate γ is set at 0.95.

Compared Methods. We compare RL-based Search (RLS), RL-based Search with skipping (RLS-Skip) and the proposed non-learning based algorithms (Section 4), namely ExactS, SizeS, PSS, POS, and POS-D. For RLS and RLS-Skip, when t2vec is adopted, we ignore the Θ_{suf} component of a state based on empirical findings.

In addition, we consider three competitor methods, namely UCR [24, 27, 28], Spring [31], and Random-S. UCR was originally developed for searching subsequences of a time series, which are the most similar to a query time series and the similarity is based on the DTW distance. UCR
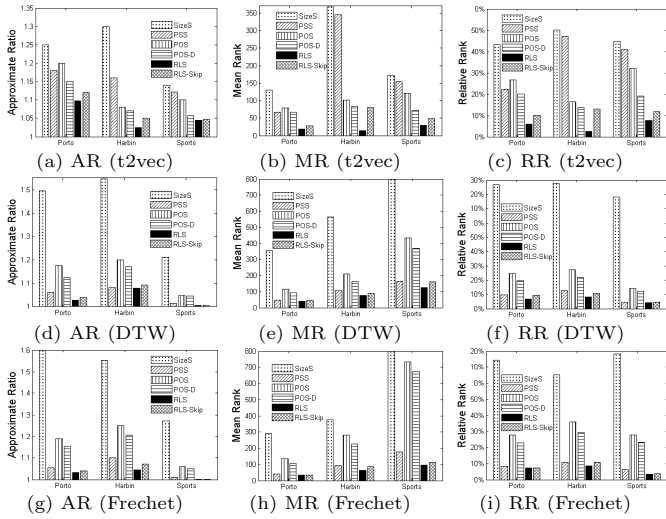
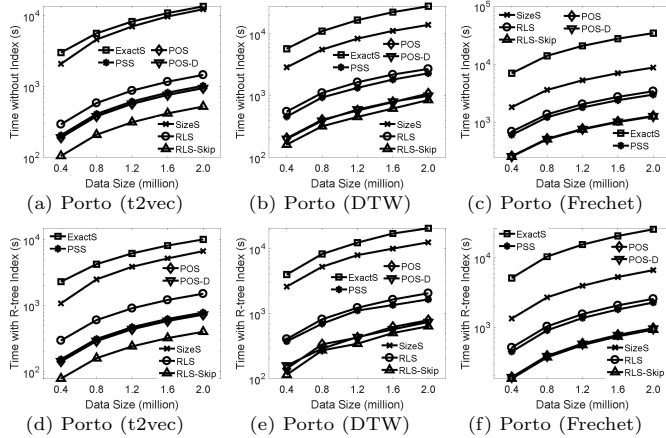Figure 3: Effectiveness for t2vec (a)-(c), DTW (d)-(f) and Frechet (g)-(i).

Figure 4: Efficiency without index (a)-(c) and with R-tree index (d)-(f) on Porto.

enumerates all subsequences that are of the same length of the query time series and employs a rich set of techniques for pruning many of the subsequences. We adapt UCR for our similar subtrajectory search problem (details of adaptions are provided in the appendix of the technical report version [42]). We note that UCR only works for DTW, but not for Frechet or t2vec. Spring is an existing algorithm for searching a subsequence of a time series, which is the most similar to a query time series. It is designed for DTW. Random-S randomly samples a certain number of subtrajectories of data trajectory and among them, returns the one with the highest similarity to the query trajectory. Since these methods are either not general for all similarity measurements (e.g., UCR) or involve some parameter that is difficult to set (e.g., Random-S with a parameter of sample size), we compare these two competitor methods with our RLS-Skip algorithm only in terms of effectiveness and efficiency.

Evaluation Metrics. We use three metrics to evaluate the effectiveness of an approximate algorithm. (1) Approximate

Ratio (AR): It is defined as the ratio between the dissimilarity of the solution wrt a query trajectory, which is returned by an approximate algorithm, and that of the solution returned by an exact algorithm. A smaller AR indicates a better algorithm. (2) Mean Rank (MR): We sort all the subtrajectories of a data trajectory in ascending order of their dissimilarities wrt a query trajectory. MR is defined as the rank of the solution returned by an approximate algorithm. (3) Relative Rank (RR): RR is a normalized version of MR by the total number subtrajectories of a data trajectory. A smaller MR or RR indicates a better algorithm.

Evaluation Platform. All the methods are implemented in Python 3.6. The implementation of RLS is based on Keras 2.2.0. The experiments are conducted on a server with 32-cores of Intel(R) Xeon(R) Gold 6150 CPU @ 2.70GHz 768.00GB RAM and one Nvidia Tesla V100-SXM2 GPU.

6.2 Experimental Results

(1) Effectiveness results. We randomly sample 10,000 trajectory pairs from a dataset, and for each pair we use one trajectory as the query trajectory to search the most similar subtrajectory from the other one. Figure 3 shows the results. The results clearly show that RLS and RLS-Skip consistently outperform all other non-learning based approximate algorithms in terms of all three metrics on both datasets and under all three trajectory similarity measurements. For example, RLS outperforms POS-D, the best non-learning algorithm when using t2vec, by 70% (resp. 83%) in terms of RR on Porto (resp. Harbin); RLS outperforms PSS, the best non-learning based algorithm when using DTW, by 25% (resp. 20%) in terms of MR on Porto (resp. Harbin); RLS outperforms PSS, the best non-learning based algorithm when using Frechet, by 25% (resp. 20%) in terms of MR on Porto (resp. Harbin). Among PSS, POS, and POS-D, PSS performs the best for DTW and Frechet; However, for t2vec, PSS provides similar accuracy as POS and POS-D on Porto, but performs much worse on Harbin. The reason is that for DTW and Frechet, PSS computes exact similarity values for suffix subtrajectories, while for t2vec, it computes only approximate ones. Therefore, PSS has a relatively worse accuracy when used for t2vec. We also observe that SizeS is not competitive compared with other approximate algorithms. In addition, RLS-Skip has its effectiveness a bit worse than RLS, but still better than those non-learning based algorithms due to the fact that it is based on a learned policy for decision making.

(2) Efficiency results. We prepare different databases of data trajectories by including different amounts of trajectories from a dataset and vary the total number of points in a databases. For each database, we randomly sample 10 query trajectories from the dataset, run for each query trajectory a query for finding the top-50 similar subtrajectories, and then collect the average running time over 10 queries. The results of running time on the Porto dataset are shown in Figure 4, and those on the other datasets could be found in the technical report [42]. RLS-Skip runs the fastest since on those points that have been skipped, the cost of maintaining the states and making decisions is saved. In contrast, none of the other algorithms skip points. ExactS has the longest running time, e.g., ExactS is usually around 7-15 times slower than PSS, POS, POS-D, RLS and 20-30 times slower than RLS-Skip. RLS is slightly slower than PSS,
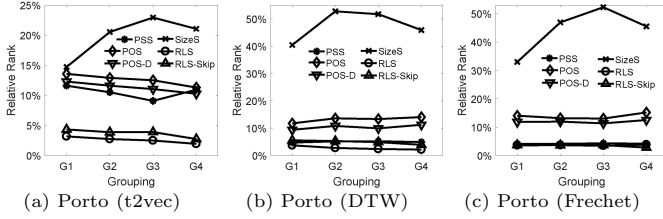
(a) Porto (t2vec) (b) Porto (DTW) (c) Porto (Frechet)

Figure 5: Effectiveness with varying query lengths.

(a) Porto (t2vec) (b) Porto (DTW) (c) Porto (Frechet)
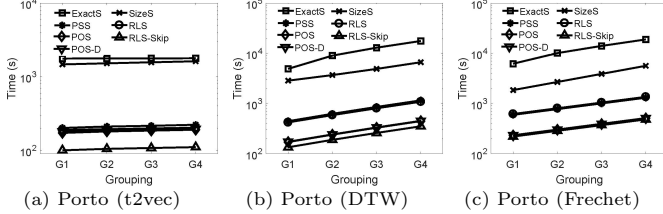
Figure 6: Efficiency with varying query lengths.

Table 5: The effect of skipping steps k for RLS-Skip.

Metrics	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
AR	1.028	1.039	1.042	1.044	1.055	1.069
MR	41.138	56.633	58.077	64.741	70.281	94.356
RR	3.5%	5.4%	5.6%	5.8%	6.3%	8.9%
Time (ms)	55.2	39.8	38.5	35.8	31.8	22.9
Skip Pts	0%	3.1%	13.1%	17.7%	29.5%	47.6%

(a) Relative Rank (DTW) (b) Time Cost (DTW)

Figure 7: The effect of soft margin ξ for SizeS.

POS, POS-D. This is because RLS makes the splitting decision via a learning model while the other three use a simple similarity comparison.

(3) Scalability. We investigate the scalability of all the algorithms based on the results reported in Figure 4. All those splitting-based algorithms including PSS, POS, POS-D, RLS and RLS-Skip scale well.

(4) Working with indexes. Following two recent studies [45, 39] on trajectory similarity search, we employ the Bounding Box R-tree Index for boosting the efficiency. It indexes the MBRs of data trajectories and prunes all those data trajectories whose MBRs do not interact with the MBR of a given query trajectory. We note that in theory, exact solutions might be filtered out by the index (e.g., the most similar subtrajectory may be part of a trajectory, whose MBR does not interact with that of a query one), but in practice, this rarely happen. For example, as found in our experiments on the Porto dataset, when DTW and Frechet are used, the results returned when using the index and those when using no indexes are exactly the same, i.e., no results are missed out. When t2vec is used, at most 20% results are missed. Furthermore, for cases of finding approximate solutions, as most of the proposed algorithms do, missing some potential solutions for better efficiency is acceptable. Compared with the results without indexes in Figure 4(a)-(c), the results using the R-tree index as shown in Figure 4(d)-(f) are lower by around 20–30%.

(5) The effect of query trajectory length. We prepare four groups of query trajectories from a dataset, namely $G1$, $G2$, $G3$, and $G4$, each with 10,000 trajectories, such that the lengths of the trajectories in a group are as follows: $G1 = [30, 45)$, $G2 = [45, 60)$, $G3 = [60, 75)$ and $G4 = [75, 90)$. Then, for each query trajectory, we prepare a data trajectory from the dataset. Note that for a query trajectory and a corresponding data trajectory, the latter may be longer than or shorter than the former. For each group, we report the average results. The results of RR on Porto are shown in Figure 5. We observe that the RRs of all algorithms except for SizeS remain stable when the query length grows. For SizeS, the RRs fluctuate with the change of query length.

This is because the length of the most similar subtrajectory may not have similar length as the query trajectory, and thus it may miss high-quality results when the search space is constrained by the parameter ξ. The results of running time on Porto are shown in Figure 6. We notice that for t2vec, the running times of all the algorithms are almost not affected by the query length. This is because for t2vec, the time complexity of computing a similarity is constant once the vector of the query trajectory is learned. For DTW, Frechet, the time complexity of computing a similarity increase with the query length, as shown in Table 2.

(6) The effect of skipping steps k. According to the results, a general trend is that with larger settings of k, RLS-Skip has its effectiveness drop but its efficiency grow because RLS-Skip tends to skip more points. We present in Table 5 the results on Porto for DTW only due to the page limit. We also report the portion of skipped points in the Porto dataset with 10,000 trajectories. Note that when k is set to 0, RLS-Skip degrades to RLS. For other experiments, we choose $k = 3$ as a reasonable trade-off between effectiveness and efficiency.

(7) The effect of parameter ξ. Figure 7 shows SizeS's RR and running time averaged on 10,000 trajectory pairs from the Porto dataset. As expected, as ξ grows, the RR of SizeS becomes better, but running time increases and approaches to that of ExactS.

(8) Comparison with similar trajectory search (SimTra). The solution of the similar trajectory search (SimTra) could be regarded as an approximate solution of the SimSub problem because a data trajectory by itself is a subtrajectory. We compare this approximate solution by SimTra and that by the RLS algorithm. We report the average results over 10,000 trajectory pairs. The results are shown in Table 6. The MR and RR of SimTra are around 10 times larger than those of SimSub for t2vec and 20 times for DTW and Frechet, which shows that SimTra is not a good approximation for SimSub, though SimTra runs faster than SimSub.

(9) Comparison with algorithms for specific measurements (UCR and Spring). In UCR and Spring, a point q_i from the query trajectory can be aligned with only those points p_j from the data trajectory T with $j \in [i - R \cdot |T|, i + R \cdot |T|]$. We vary the parameter R in this

Table 6: Comparison with Trajectory Similarity Computation and Subtrajectory Similarity Computation.

Similarity		t2vec				DTW				Frechet			
Dataset	Problem	AR	MR	RR	Time (ms)	AR	MR	RR	Time (ms)	AR	MR	RR	Time (ms)
Porto	SimTra	1.313	156.153	23.3%	28.5	2.100	752.831	70.7%	18.1	1.883	559.462	56.5%	19.2
	SimSub	1.098	18.323	3.0%	39.6	1.028	41.138	3.5%	55.2	1.034	34.162	3.6%	69.6
Harbin	SimTra	1.293	678.311	46.9%	31.7	2.326	1218.908	72.2%	27.1	1.891	854.042	53.9%	28.6
	SimSub	1.025	14.945	1.3%	62.6	1.081	75.324	4.1%	114.4	1.045	64.729	4.4%	130.6
Sports	SimTra	1.221	345.488	43.4%	46.1	1.659	4291.666	59.8%	107.5	1.403	3272.743	48.2%	133.3
	SimSub	1.045	28.761	3.8%	210.3	1.005	126.334	2.1%	254.7	1.002	95.280	1.7%	302.3

(a) Relative Rank (DTW) (b) Time Cost (DTW)
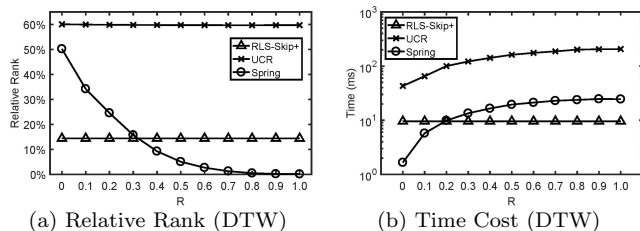
Figure 8: Comparison with UCR and Spring.

(a) Relative Rank (DTW) (b) Time Cost (DTW)

Figure 9: Comparison with Random-S.

Table 7: Training time (hours).

Similarity	t2vec		DTW		Frechet	
Algorithms	RLS	RLS-Skip	RLS	RLS-Skip	RLS	RLS-Skip
Porto	7.2	4.4	10.1	4.8	10.6	5.5
Harbin	9.7	5.4	13.9	5.7	14.2	8.3
Sports	12.5	7.6	83.3	46.6	104.1	52.4

experiment. When $R = 1$, it reduces to the unconstrained DTW that is used in this paper. Essentially, R controls how accurately the DTW distance is computed: the higher R is, the more accurate (but also more costly) the computation is. We note that even when $R = 1$, UCR does not return exact solutions since it considers subtrajectories of the same size of the query one only. For this part of experiment, we drop the component Θ_{suf} when defining the MDP of RLS-Skip for better efficiency and call the resulting algorithm *RLS-Skip+*. The results are shown in Figure 8, where we vary the parameter R from 0 to 1. We notice that (1) RLS-Skip+ dominates UCR in terms of both efficiency and effectiveness; (2) the RR of UCR changes slightly from 60.1% (when $R = 0$) to 59.7% (when $R = 1$), which shows that the performance of UCR is insensitive to the parameter R; (3) under settings of $0.2 \le R \le 0.3$, RLS-Skip+ dominates Spring in terms of both effectiveness and efficiency; and (4) under other settings, RLS-Skip+ and Spring provide different trade-offs between effectiveness and efficiency.

(10) Comparison with Random-S. The results are shown in Figure 9, where we vary the sample size from 10 to 100 and for each sample size, we run the algorithm 100 times and collect the average and standard deviations of the metrics of RR and running time. We notice that for a relatively small sample size, e.g., 100, the running time of Random-S is almost that of ExactS and significantly larger than that of RLS-Skip (25 times higher). This is because for Random-S, the subtrajectories that are considered could be quite different, and thus it is not possible to compute their similarities *incrementally* as it does for ExactS. Whereas when the sample size is small, e.g., below 20, Random-S has its effectiveness significantly degraded, which is clearly worse than that of RLS-Skip.

(11) Training time. The training times of the RLS and RLS-Skip models on different datasets are shown in Table 7. It normally takes a couple of hours to train a reinforcement learning model for RLS and RLS-Skip. It takes less time to train RLS-Skip than RLS since we use the same number of trajectory pairs and epochs for training both algorithms and RLS-Skip runs faster.
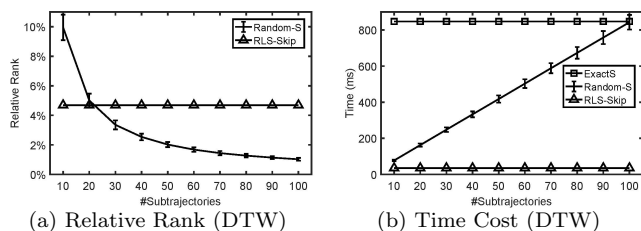
7. CONCLUSION

In this paper, we study the similar subtrajectory search (SimSub) problem and develop a suite of algorithms including an exact algorithm, an approximate algorithm providing a controllable trade-off between efficiency and effectiveness, and a few splitting-based algorithms, among which some are based on pre-defined heuristics and some are based on deep reinforcement learning called RLS and RLS-Skip. We conducted extensive experiments on real datasets, which verified that among the approximate algorithms, learning based algorithms achieve the best effectiveness and efficiency. In the future, we plan to explore some more similarity measurements for the SimSub problem, e.g., the constrained DTW distance and other similarity measurements reviewed in Section 2.

8. REFERENCES

[1] P. K. Agarwal, K. Fox, K. Munagala, A. Nath, J. Pan, and E. Taylor. Subtrajectory clustering: Models and algorithms. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 75–87. ACM, 2018.

[2] H. Alt and M. Godau. Computing the fréchet distance between two polygonal curves. *International Journal of Computational Geometry & Applications*, 5(01n02):75–91, 1995.

[3] V. Athitsos, P. Papapetrou, M. Potamias, G. Kollios, and D. Gunopulos. Approximate embedding-based subsequence matching of time series. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 365–378. ACM, 2008.

[4] R. I. Brafman and M. Tennenholtz. R-max-a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3(Oct):213–231, 2002.

[5] K. Buchin, M. Buchin, J. Gudmundsson, M. Löffler, and J. Luo. Detecting commuting patterns by clustering subtrajectories. *International Journal of Computational Geometry & Applications*, 21(03):253–282, 2011.

[6] L. Chen and R. Ng. On the marriage of lp-norms and edit distance. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 792–803. VLDB Endowment, 2004.

[7] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 491–502. ACM, 2005.

[8] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[9] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. *Fast subsequence matching in time-series databases*, volume 23. ACM, 1994.

[10] X. Gong, S. Fong, and Y.-W. Si. Fast fuzzy subsequence matching algorithms on time-series. *Expert Systems with Applications*, 116:275–284, 2019.

[11] W.-S. Han, J. Lee, Y.-S. Moon, and H. Jiang. Ranked subsequence matching in time-series databases. In *Proceedings of the 33rd international conference on Very large data bases*, pages 423–434. VLDB Endowment, 2007.

[12] M. Kearns and S. Singh. Near-optimal reinforcement learning in polynomial time. *Machine learning*, 49(2-3):209–232, 2002.

[13] E. Keogh and C. A. Ratanamahatana. Exact indexing of dynamic time warping. *Knowledge and information systems*, 7(3):358–386, 2005.

[14] S.-W. Kim, S. Park, and W. W. Chu. An index-based approach for similarity search supporting time warping in large sequence databases. In *Proceedings 17th International Conference on Data Engineering*, pages 607–614. IEEE, 2001.

[15] Y. Kim and K. Shim. Efficient top-k algorithms for approximate substring matching. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 385–396. ACM, 2013.

[16] J.-G. Lee, J. Han, and K.-Y. Whang. Trajectory clustering: a partition-and-group framework. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 593–604. ACM, 2007.

[17] G. Li, X. Zhou, S. Li, and B. Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment*, 12(12):2118–2130, 2019.

[18] X. Li, K. Zhao, G. Cong, C. S. Jensen, and W. Wei. Deep representation learning for trajectory similarity computation. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 617–628. IEEE, 2018.

[19] C. Long, R. C.-W. Wong, and H. Jagadish. Direction-preserving trajectory simplification. *Proceedings of the VLDB Endowment*, 6(10):949–960, 2013.

[20] C. Ma, H. Lu, L. Shou, and G. Chen. Ksq: Top-k similarity query on uncertain trajectories. *IEEE Transactions on Knowledge and Data Engineering*, 25(9):2049–2062, 2012.

[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[22] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[23] Y.-S. Moon, K.-Y. Whang, and W.-K. Loh. Duality-based subsequence matching in time-series databases. In *Proceedings 17th International Conference on Data Engineering*, pages 263–272. IEEE, 2001.

[24] A. Mueen and E. Keogh. Extracting optimal performance from dynamic time warping. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2129–2130, 2016.

[25] S. Park, W. W. Chu, J. Yoon, and C. Hsu. Efficient searches for similar subsequences of different lengths in sequence databases. In *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*, pages 23–32. IEEE, 2000.

[26] M. L. Puterman. *Markov Decision Processes.: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 2014.

[27] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 262–270. ACM, 2012.

[28] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. Addressing big data time series: Mining

trillions of time series subsequences under dynamic time warping. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 7(3):10, 2013.

[29] S. Ranu, P. Deepak, A. D. Telang, P. Deshpande, and S. Raghavan. Indexing and matching trajectories under inconsistent sampling rates. In *2015 IEEE 31st International Conference on Data Engineering*, pages 999–1010. IEEE, 2015.

[30] D. E. Rumelhart, G. E. Hinton, R. J. Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[31] Y. Sakurai, C. Faloutsos, and M. Yamamuro. Stream monitoring under the time warping distance. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 1046–1055. IEEE, 2007.

[32] L. Sha, P. Lucey, Y. Yue, P. Carr, C. Rohlf, and I. Matthews. Chalkboarding: A new spatiotemporal query paradigm for sports play retrieval. In *Proceedings of the 21st International Conference on Intelligent User Interfaces*, pages 336–347, 2016.

[33] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[34] P. Tampakis, C. Doulkeridis, N. Pelekis, and Y. Theodoridis. Distributed subtrajectory join on massive datasets. *arXiv preprint arXiv:1903.07748*, 2019.

[35] P. Tampakis, N. Pelekis, C. Doulkeridis, and Y. Theodoridis. Scalable distributed subtrajectory clustering. *arXiv preprint arXiv:1906.06956*, 2019.

[36] I. Trummer, S. Moseley, D. Maram, S. Jo, and J. Antonakakis. Skinnerdb: regret-bounded query evaluation via reinforcement learning. *Proceedings of the VLDB Endowment*, 11(12):2074–2077, 2018.

[37] M. Vlachos, G. Kollios, and D. Gunopulos. Discovering similar multidimensional trajectories. In *Proceedings 18th international conference on data engineering*, pages 673–684. IEEE, 2002.

[38] S. Wang, Z. Bao, J. S. Culpepper, T. Sellis, and X. Qin. Fast large-scale trajectory clustering. *Proceedings of the VLDB Endowment*, 13(1):29–42, 2019.

[39] S. Wang, Z. Bao, J. S. Culpepper, Z. Xie, Q. Liu, and X. Qin. Torch: A search engine for trajectory data. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 535–544. ACM, 2018.

[40] Y. Wang, Y. Tong, C. Long, P. Xu, K. Xu, and W. Lv. Adaptive dynamic bipartite graph matching: A reinforcement learning approach. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1478–1489. IEEE, 2019.

[41] Z. Wang, C. Long, G. Cong, and C. Ju. Effective and efficient sports play retrieval with deep representation learning. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 499–509, 2019.

[42] Z. Wang, C. Long, G. Cong, and Y. Liu. Efficient and effective similar subtrajectory search with deep reinforcement learning (technical report). https://www.ntu.edu.sg/home/wang_zheng/paper/TR-subtraj.pdf.

[43] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[44] M. Xie. Eds: a segment-based distance measure for sub-trajectory similarity search. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 1609–1610. ACM, 2014.

[45] D. Yao, G. Cong, C. Zhang, and J. Bi. Computing trajectory similarity in linear time: A generic seed-guided neural metric learning approach. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1358–1369. IEEE, 2019.

[46] B.-K. Yi, H. V. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *Proceedings 14th International Conference on Data Engineering*, pages 201–208. IEEE, 1998.

[47] H. Yuan and G. Li. Distributed in-memory trajectory similarity search and join on road network. In *2019 IEEE 35th international conference on data engineering (ICDE)*, pages 1262–1273. IEEE, 2019.

[48] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, pages 415–432. ACM, 2019.

[49] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 915–926. ACM, 2010.

APPENDIX

A. QUALITY ANALYSIS OF THE SIZES ALGORITHM

There usually exists a space, within which the objects move. Therefore, we assume the points of trajectories are all located in a $d_{max} \times d_{max}$ rectangle, where d_{max} is a large number that captures the extent of the space. We use a coordinate system whose origin is at the middle of this rectangle.

Case 1: DTW. Consider a problem input with a query trajectory with m points $T_q = < p'_1, p'_2, ..., p'_m >$ and a data trajectory with $n = m^2$ points $T = < p_{1,1}, p_{1,2}, ..., p_{1,m}, p_{2,1}, p_{2,2}, ..., p_{2,m}, ..., p_{m,1}, p_{m,2}, p_{m,m} >$. We assume that m is an even number and $l = m/2$. In addition, we let $d = d_{max}/m$. The locations of the points in these trajectories are provided as follows.

- $p'_i = (-(l-i+1/2) \cdot d, 0)$ for $i = 1, 2, ..., l$;
- $p'_i = ((i-l-1/2) \cdot d, 0)$ for $i = l+1, l+2, ..., m$;
- $p_{i,j}$'s $(1 \leq j \leq m)$ are evenly located at the circle with its center at p'_i and its radius equal to ϵ, where ϵ is a very small real number for $i = 1, 2, ..., m$;

Consider the optimal solution. Its DTW distance, denoted by D_o, is at most the distance between T and T_q, which is equal to $m^2 \cdot \epsilon$ (points $p_{i,j}$ for $1 \leq j \leq m$ are aligned with point p'_i for $1 \leq i \leq m$). That is, we have $D_o \leq m^2 \cdot \epsilon$.

Consider the approximate solution returned by SizeS. Suppose that $\xi = 0$. That is, we only consider subtrajectories with the length exactly equal to m. We further know that each such sub-trajectory consists of points, either all located at a circle with the center at one of the point of T_q or some located at a circle with the center at a point p'_i and others located at a circle with the center at a point p'_{i+1} $(1 \leq i < m)$. It could be verified that among these subtrajectories, the one with some located at the circle with the center at p'_l and others at p'_{l+1} has the smallest DTW distance from T_q and would be returned. We denote its DTW distance to T_q by D_a. It could be verified that $D_a > 2 \times (\Sigma_{i=1}^{l-1}((l-i) \cdot d - \epsilon) + \epsilon)$, where the lower bound of the Euclidean distance from a point of T_q to its aligned point of the returned subtrajectory is (1) $((l-i) \cdot d - \epsilon)$ for $1 \leq i \leq l-1$ and (2) ϵ for $i = l$; and (3) that of the point symmetric to p'_i (w.r.t. the origin) for $l+1 \leq i \leq m$ due to the symmetry. Therefore, we have

$$\frac{D_a}{D_o} \geq \frac{2 \times (\Sigma_{i=1}^{l-1}((l-i) \cdot d - \epsilon) + \epsilon)}{m^2 \cdot \epsilon}$$
$$= \frac{m^2/4 \cdot d - m/2 \cdot d - m \cdot \epsilon + 4 \cdot \epsilon}{m^2 \cdot \epsilon}$$
$$= \frac{1/4 \cdot d - 1/(2m) \cdot d - \epsilon/m + 4/m^2 \cdot \epsilon}{\epsilon}$$

which approaches infinity when m approaches infinity and ϵ approaches 0. In summary, the solution returned by SizeS could be arbitrarily worse than the optimal one when DTW distance is used.

Case 2: Frechet and t2vec. Consider the case when Frechet distance is used. The optimal solution and the approximate solution returned would be the sames as the case when DTW distance is used, but the distances would be different. D_o would be equal to ϵ and D_a would be at least $((l-1) \cdot d - \epsilon)$. Therefore, we have

$$\frac{D_a}{D_o} \geq \frac{(l-1) \cdot d - \epsilon}{\epsilon}$$

which approaches infinity when ϵ approaches 0.

Consider the case when t2vec is used. Since it is a learning-based distance - in theory, it may reduce to any possible distance metric such as DTW and Frechet. Thus, the analysis for DTW or Frechet could be carried over for t2vec.

B. QUALITY ANALYSIS OF SPLITING-BASED ALGORITHMS

Case 1: PSS with DTW. Consider a problem input with a data trajectory with $n + 3$ points $T = < p'_1, p'_2, p_1, p_2, ..., p_n, p'_3 >$ (n is a positive integer) and a query trajectory $T_q = < p' >$. Let $d = d_{max}/2$. The locations of the points in these trajectories are provided as follows.

- $p'_1 = (-d/2, 0)$, $p'_2 = (-d, 0)$;
- $p_i = (0, 0)$ for $i = 1, 2, ..., n$;
- $p'_3 = (d, 0)$;
- $p' = (0, \epsilon)$, where ϵ is a very small non-negative real number;

Consider the optimal solution. It could be any subtrajectory $< p_i > (1 \leq i \leq n)$ and the corresponding DTW distance is equal to ϵ, which we denote by D_o.

Consider the approximate solution returned by PSS. It is the subtrajectory $< p'_1 >$, which is explained as follows. When it scans the first point p'_1, it would split the trajectory at p'_1 and update the best-known subtrajectory to be $< p'_1 >$ with the DTW distance equal to $\sqrt{d^2/4 + \epsilon^2}$. It then continues to scan the following points $p'_2, p_1, ..., p_n, p'_3$ and would not perform any split operations at these points due to the fact that p'_2 and p'_3 are farther away from p' than p'_1. As a result, $< p'_1 >$ would be returned as a solution. We denote the DTW distance of this solution by D_a, i.e., $D_a = \sqrt{d^2/4 + \epsilon^2}$.

Consider the approximation ratio (AR). We have

$$\frac{D_a}{D_o} = \frac{\sqrt{d^2/4 + \epsilon^2}}{\epsilon} > \frac{d/2}{\epsilon}$$

which approaches infinity when ϵ approaches zero.

Consider the mean rank (MR). We know that the rank of the approximate solution $< p'_1 >$ is at least $\frac{n(n+1)}{2} + 1$ since any subtrajectory of $< p_1, p_2, ..., p_n >$ has a smaller DTW distance than $< p'_1 >$ (assuming $\epsilon = 0$). Therefore, the mean rank would approach infinity when n approaches infinity.

Consider the relative rank (RR). Based on the analysis of mean rank, we know that the relative rank of the approximate solution $< p'_1 >$ is at least $\frac{\frac{n(n+1)}{2} + 1}{\frac{(n+3)(n+4)}{2}}$, which approaches 1 when n approaches infinity.

In conclusion, the solution returned by PSS could be arbitrarily worse than the optimal one in terms of AR, MR, and RR, when the DTW distance is used.

Case 2: Other Algorithms and Similarity Measurements Consider the other algorithms, i.e., POS and POS-D. It could be verified that they would run exactly in the same way as PSS on the problem input provided in the Case 1.

Therefore, the conclusion would be carried over. Consider the other measurements, namely, Frechet and t2vec. For Frechet, it could be verified that the optimal solution and the approximate solution returned by the algorithms are the same as those in the Case 1 and their Frechet distances and DTW distances to T_q are equal since both solutions and T_q involve one single point. As a result, the conclusion for DTW distance could be carried over for Frechet distance. For t2vec, since it is a learning-based distance - in theory, it may reduce to any possible distance metric such as DTW and Frechet. Thus, the analysis for DTW in the Case 1 could be carried over for t2vec.

C. ADAPTION OF UCR

UCR [27] was originally developed for searching subsequences of a time series, which are the most similar to a query time series and the similarity is based on the DTW distance. Specifically, UCR enumerates all subsequences that are of the same length of the query time series and employs a rich set of techniques for pruning many of the subsequences. We use UCR for our "similar subtrajectory search" problem by adapting UCR's pruning techniques for trajectories. UCR involves seven techniques used for pruning, organized in two groups. Let $T =< p_1, p_2, ..., p_n >$ be a data trajectory and $T_q =< q_1, q_2, ..., q_m >$ be a query trajectory. Suppose we are considering a subtrajectory of T, denoted by $T' =< p_1, p_2, ..., p_m >$ without loss of generality. We describe the adaptions of the techniques involved in UCR, which are used to prune T' from being considered if possible as follows.

Group 1: Known Optimizations

- Early Abandoning of LB_{Keogh}. This is to compute a lower bound called LB_{Keogh} of the DTW distance between T' and T_q, denoted by $LB_{Keogh}(T', T_q)$, and prune T' if the lower bound is larger than the best-known DTW distance. Specifically, let (1) $R \in [0, 1]$ be a real number, (2) $q_{i-\lfloor R \cdot m \rfloor : i + \lfloor R \cdot m \rfloor}$ be the set involving $\lfloor R \cdot m \rfloor$ points before q_i, $\lfloor R \cdot m \rfloor$ points after q_i, and point p_i, and (3) $MBR(\cdot)$ be the minimum bounding box of a set of points. We compute $LB_{Keogh}(T', T_q)$ as follows.

$$LB_{Keogh}(T', T_q) = \sum_{i=1}^{m} \begin{cases} d(p_i, MBR(q_{i-\lfloor R \cdot m \rfloor : i + \lfloor R \cdot m \rfloor})) \\ \text{if } p_i \text{ outside } MBR(q_{i-\lfloor R \cdot m \rfloor : i + \lfloor R \cdot m \rfloor}) \\ 0 \text{ otherwise} \end{cases}$$

where $d(p_i, MBR(\cdot))$ denotes shortest distance between p_i and $MBR(\cdot)$.

- Early Abandoning of DTW. During the process of computing the DTW distance between T' and T_q, when the accumulated DTW distance exceeds the best-known DTW distance, we abandon the computation.

- Earlier Early Abandoning of DTW using LB_{Keogh}. For $i = 1, 2, ..., m$, if the sum of the DTW distance between $T_q[1 : i]$ and any prefix of $T'[1 : i]$ (or between $T'[1 : i]$ and any prefix of $T_q[1 : i]$) and the LB_{Keogh} bound between $T_q[i : m]$ and $T'[i : m]$ is larger than the best-known DTW distance, we prune T'. Note that DTW distances used in this pruning are maintained in the process of computing the DTW distance between T' and T_q.

Group 2: Novel Optimizations in UCR Suite.

- Just-in-time Z-normalizations. We do not adapt the Z-normalization technique since it is designed for one-dimensional data and cannot be used for trajectory data that is two-dimensional.

- Reordering Early Abandoning. We consider the points of T_q in a descending order of their distances to the y-axis for computing the bound of LB_{Keogh}.

- Reversing LB_{Keogh}. We reverse the roles of T' and T_q and compute another LB_{Koegh} bound. We then use the larger one among the two LB_{Koegh} bounds to act as a tighter LB_{Keogh} bound when necessary.

- Cascading Lower Bounds. We first compute the $LB_{Kim}FL$ bound, which is another simple lower bound of the DTW distance between T' and T_q, denoted by $LB_{Kim}FL(T', T_q)$. Specifically, $LB_{Kim}FL(T', T_q)$ is defined as $(d(T_q[1], T'[1]) + d(T_q[m], T'[m]))$. The time complexity of this step is simply $O(1)$. If this bound does not help to prune T', we cascade the techniques of early abandoning of LB_{Keogh}, early abandoning of DTW, and earlier early abandoning of DTW using LB_{Keogh}. Finally, if T' has still not been pruned, we compute the DTW distance between T' and T_q.

We note that UCR is designed specifically for the DTW distance and cannot be used for the problem when the Frechet or t2vec distance is used.
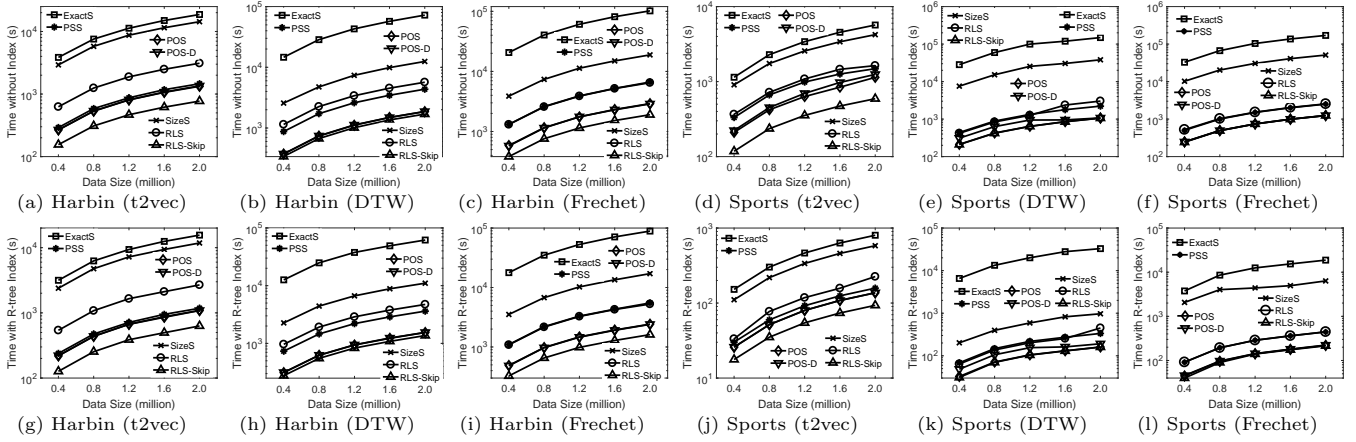
D. ADDITIONAL EXPERIMENTAL RESULTS

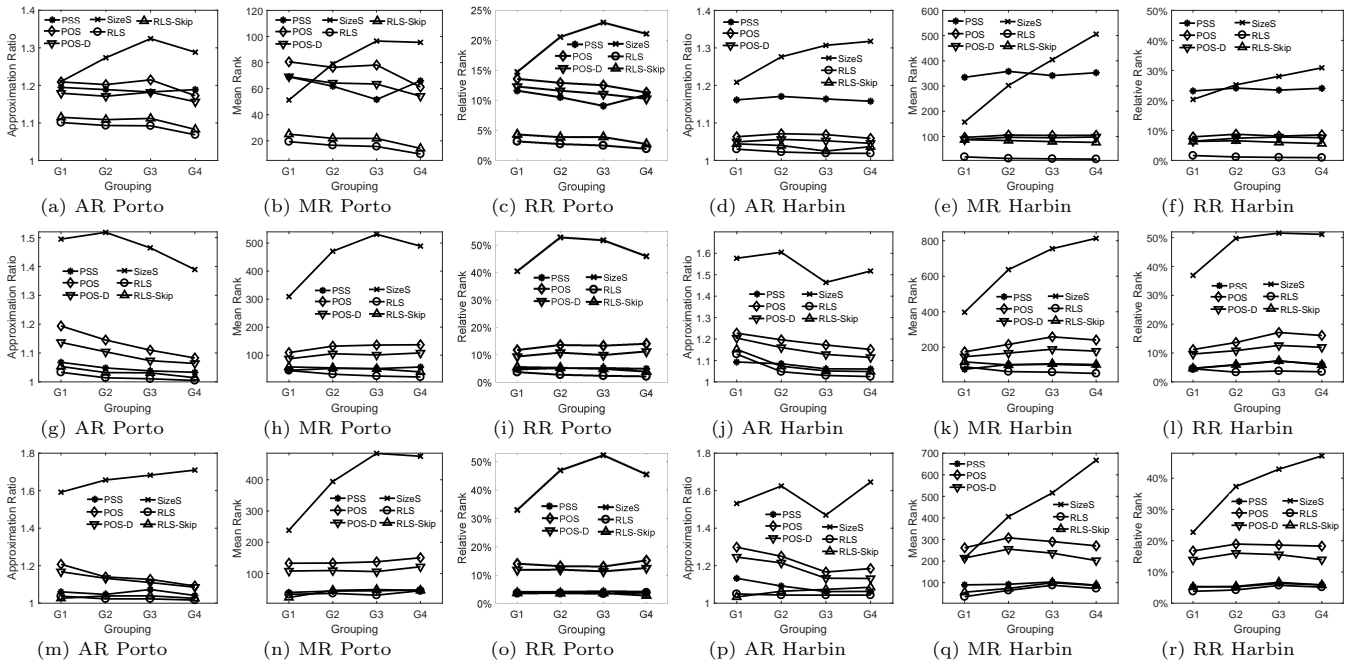Figure 10: Efficiency without index (a)-(f) and with R-tree index (g)-(l) on Harbin and Sports.

Figure 11: Results of grouping evaluation for t2vec (a)-(f), DTW (g)-(l) and Frechet (m)-(r).
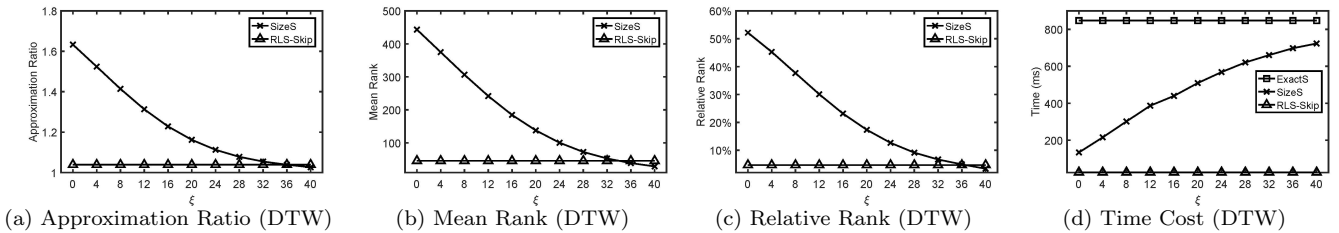

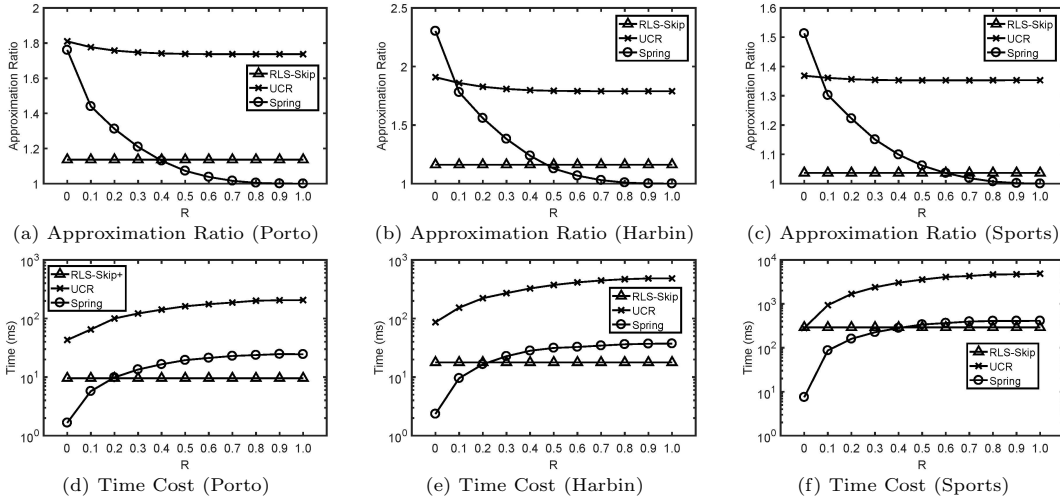Figure 12: The effect of soft margin ξ for SizeS.

(a) Approximation Ratio (Porto)   (b) Approximation Ratio (Harbin)   (c) Approximation Ratio (Sports)

(d) Time Cost (Porto)   (e) Time Cost (Harbin)   (f) Time Cost (Sports)

Figure 13: Comparison with UCR and Spring.

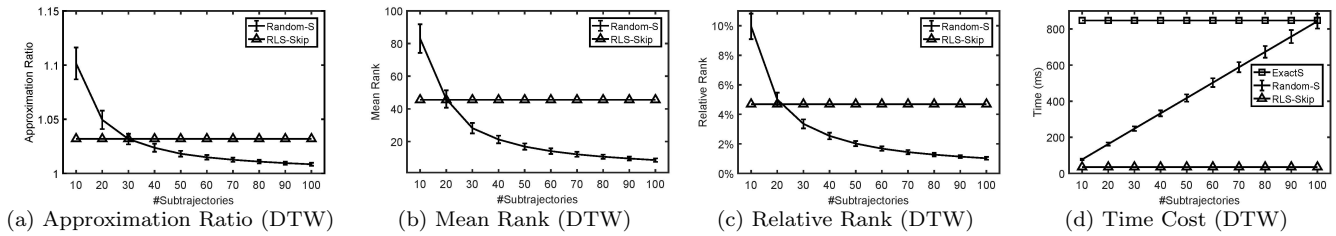(a) Approximation Ratio (DTW)   (b) Mean Rank (DTW)   (c) Relative Rank (DTW)   (d) Time Cost (DTW)

Figure 14: Comparison with Random-S.