Trajectory Similarity Measurement: An Efficiency Perspective

YANCHUAN CHANG, The University of Melbourne, Australia
EGEMEN TANIN, The University of Melbourne, Australia
GAO CONG, Nanyang Technological University, Singapore
CHRISTIAN S. JENSEN, Aalborg University, Denmark
JIANZHONG QI, The University of Melbourne, Australia

Trajectories that capture object movement have numerous applications, in which similarity computation between trajectories often plays a key role. Traditionally, the similarity between two trajectories is quantified by means of heuristic measures, e.g., Hausdorff or ERP, that operate directly on the trajectories. In contrast, recent studies exploit deep learning to map trajectories to d-dimensional vectors, called embeddings. Then, some distance measure, e.g., Manhattan or Euclidean, is applied to the embeddings to quantify trajectory similarity. The resulting similarities are inaccurate: they only approximate the similarities obtained using the heuristic measures. As distance computation on embeddings is efficient, focus has been on achieving embeddings yielding high accuracy.

Adopting an efficiency perspective, we analyze the time complexities of both the heuristic and the learning-based approaches, finding that the time complexities of the former approaches are not necessarily higher. Through extensive experiments on open datasets, we find that, on both CPUs and GPUs, only a few learning-based approaches can deliver the promised higher efficiency, when the embeddings can be pre-computed, while heuristic approaches are more efficient for one-off computations. Among the learning-based approaches, the self-attention-based ones are the fastest to learn embeddings that also yield the highest accuracy for similarity queries. These results have implications for the use of trajectory similarity approaches given different application requirements. Code is available at https://github.com/changyanchuan/TrajectorySimilarity-GPU.

1 INTRODUCTION

A trajectory is a sequence of timestamped point locations that captures the movement of an object such as a vehicle or a person. The ability to quantify the similarity between two trajectories is essential in spatio-temporal data mining [27, 49, 58, 59, 66]. Due to the rich location and movement information encoded in trajectories and many application settings, no single universal trajectory similarity measure exists. Rather, different trajectory similarity measures have been proposed for different settings. These can be largely classified into two categories: *heuristic measures* and *learned measures*.

Early studies focus on heuristic measures [5, 6, 16, 17, 36, 47, 56]. They typically work by matching the points between two trajectories (see Figure 1) and are hand-crafted to capture similarity. For example, the *Hausdorff* measure [5] computes a point matching that minimizes the sum of point-to-trajectory distances of two trajectories, and the *ERP* measure [16] computes a point matching by adapting the edit distance [43]. Popular heuristic measures like the above have quadratic time complexity in the number of points in trajectories to examine. This complexity is considered a drawback in the literature that studies learned measures [11, 19, 22, 68, 69, 74].

Learned-measure approaches [18, 28, 32, 67–70] mainly aim to improve the computational efficiency by exploiting deep learning and have recently attracted substantial interest. They generally follow the steps: (1) encoding trajectories as vectors (called *trajectory embeddings*), and (2) computing the vector distance (e.g., the Manhattan distance) between the embeddings of two trajectories then serves as the trajectory distance (see Figure 2). For example, *t2vec* [39], *NEUTRAJ* [69], and

Authors' addresses: Yanchuan Chang, The University of Melbourne, Australia, yanchuanc@student.unimelb.edu.au; Egemen Tanin, The University of Melbourne, Australia, etanin@unimelb.edu.au; Gao Cong, Nanyang Technological University, Singapore, gaocong@ntu.edu.sg; Christian S. Jensen, Aalborg University, Denmark, csj@cs.aau.dk; Jianzhong Qi, The University of Melbourne, Australia, jianzhong.qi@unimelb.edu.au.

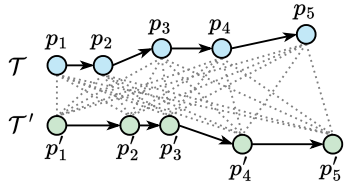


Fig. 1. Computation of heuristic measures (dotted lines indicate point-to-point distance computation)

TMN [67] use recurrent neural networks (RNNs) [20, 34] to create trajectory embeddings, while *T3S* [68] and *TrajCL* [11] use self-attention models [55].

Among learned measures, some (e.g., NEUTRAJ and T3S) "learn" to *approximate existing heuristic measures* (e.g., Hausdorff or ERP), i.e., the training signals are the ground-truth trajectory similarities provided by heuristic measures. Such sacrifice in accuracy is expected to be rewarded by higher computational efficiency.

Another series of the learned measures (e.g., t2vec and TrajCL) not only improves computational efficiency but also *improves the measurement effectiveness*. Such methods typically use self-supervised learning techniques to learn robust measures from unlabeled trajectories directly without relying on any heuristic measure. Once trained, they generally have better effectiveness, especially on measuring low-quality trajectories, than the heuristic ones.

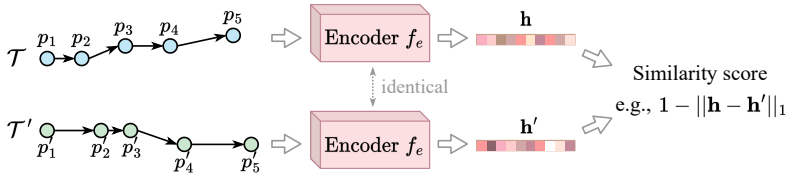


Fig. 2. Computation of learned measures

A general perception of the learned measures in either category is that the learning-based approach yields superior efficiency, due to the simple vector format of embeddings and the highly optimized implementations of deep learning processes.

However, we observe that the time complexities of the learned measures are not necessarily lower than those of the heuristic ones. For example, T3S [68] and TrajCL [11] also have quadratic time complexity in the number of trajectory points (covered in Section 3.2). Although previous studies [11, 19, 22, 67–69] report that the learned measures take less time to compute, they do not share detailed experimental settings, e.g., as to whether GPUs or CPUs are used for measuring trajectory similarity.

These observations prompt two questions: *(1) Do learned measures have better time and space complexities than heuristic measures, and are they more efficient? (2) How do the learned measures compare with each other in terms of accuracy given training time constraints, e.g., due to different application requirements?*

This study provides the first answers to these questions based on comprehensive experiments on the efficiency of heuristic and learned measures. This way, our study aims to provide a foundation for the community to pursue promising research directions and to provide guidance on selecting a suitable similarity measure for an application.

We start by covering the computational complexity of representative heuristic and learned measures and discuss their strengths and limitations (Section 3). We find that most of the existing learned measures do not necessarily have lower time complexities.

Then, we design three meta-algorithms for parallelizing the computation of heuristic measures (Section 4). We implement nine heuristic measures that leverage these algorithms to use CUDA

streaming cores, thus enabling GPU-based empirical comparisons with learned measures. The meta-algorithms provide a comprehensive basis for future studies to implement new heuristic measures.

We compare the efficiency of both types of measures using GPUs and CPUs on real datasets in a variety of settings (Section 5). We use the measures for trajectory similarity computation, trajectory clustering [15, 37], and trajectory kNN queries [33, 66, 75]. Previous studies [39, 69, 70] use spatial indices like R-trees [7] for trajectory kNN queries, which does not fit the learned measures. We use a kNN query framework designed for high-dimensional vectors to better realize the potential of the learned measures. We also investigate the impact of training time constraints on the accuracy of kNN queries for the learned measures, to provide guidance on selecting a suitable similarity measure for an application.

To sum up, we make the following contributions:

- We review the existing heuristic and learned measures and provide their computational complexity, finding that the learned measures do not necessarily have the lowest time complexities.
- We report on an extensive evaluation of the efficiency of both types of measures on GPUs and CPUs, finding that:
  (1) The heuristic measures are most efficient for one-off trajectory similarity computation (online matching of incoming trajectories or kNN queries with data updates, e.g., for ride-sharing).
  (2) The learned measures are most efficient for offline trajectory clustering and kNN queries, although they only offer approximate results.
  (3) Further, among the learned measures, the self-attention-based ones are the fastest to train and also offer the highest accuracy for kNN queries.
- We cover a simulated experiment to show a learned measure that outperforms heuristic measures at one-off computation, and we provide future directions for achieving such a learned measure.

Several survey papers [51, 52, 57] cover heuristic measures, one of which [57] also mentions a few early studies of learned measures. None of these papers include a comprehensive comparison between the two types of measures, which is our focus.

2 PRELIMINARIES

We start by defining core concepts. Frequently used notion is listed in Table 1.

Table 1. Frequently used notion

Symbol	Description
$\mathcal{T}$	A trajectory
p_i	The i-th point in a trajectory
n	The number of points in a trajectory
$\mathbf{h}$	A trajectory embedding
d	The dimensionality of trajectory embeddings
D	A trajectory dataset

Trajectory. A trajectory $\mathcal{T} = [p_1, p_2, \ldots, p_n]$ is a sequence of n points, where point p_i is either given by a pair of coordinates (x_i, y_i) or a pair of timestamped coordinates (x_i, y_i, t_i). Trajectories without timestamps are also called paths.

Heuristic trajectory similarity measure. Given a trajectory dataset D, the similarity between two trajectories is defined by a function $f\colon D \times D \to \mathbb{R}_{\geqslant 0}$.

A heuristic measure f_h: $D \times D \rightarrow \mathbb{R}_{\geqslant 0}$ is a handcrafted function that aims to capture the similarity between two trajectories. Examples include EDR and Hausdorff.

Distance measures can easily be converted into similarity measures, and thus we also consider them as similarity measures.

Learned trajectory similarity measure. A learned measure f_l involves a two-step process. First, an encoding function f_e: $D \rightarrow \mathbb{R}^d$ is applied to map each trajectory into a d-dimensional embedding space. Second, the distance between the embeddings $\mathbf{h}$ and $\mathbf{h}'$ of trajectories $\mathcal{T}$ and $\mathcal{T}'$ is used to quantify the similarity between $\mathcal{T}$ and $\mathcal{T}'$, e.g., $f_l(\mathcal{T}, \mathcal{T}') = 1 - ||f_e(\mathcal{T}) - f_e(\mathcal{T}')||_1 = 1 - ||\mathbf{h} - \mathbf{h}'||_1$, using the Manhattan distance. Here, f_e is a learned function. Examples include NEUTRAJ and T3S.

Trajectory similarity query. Given a trajectory dataset D, a query trajectory $\mathcal{T}_q$, a trajectory similarity measure f, and a positive integer k, a *trajectory similarity query* returns a set $S \subset D$ with $|S| = k$ such that $\forall \mathcal{T} \in S, \mathcal{T}' \in D \setminus S \;\; (f(\mathcal{T}_q, \mathcal{T}) \geqslant f(\mathcal{T}_q, \mathcal{T}'))$. This query is also called a *trajectory kNN query*.

Trajectory clustering. Given a trajectory dataset D and a trajectory similarity measure f, *trajectory clustering* groups the trajectories in D into subsets (i.e., clusters) based on their similarity.

Both heuristic and learned measures can be applied in trajectory similarity queries and clustering. A learned measure typically approximates some heuristic measures. The approximation error is referred to as the *inaccuracy* of the learned measure.

3 TRAJECTORY SIMILARITY MEASURES

Next, we cover existing studies on *trajectory similarity measures*, including both *heuristic* and *learned* ones, and trajectory queries, including *trajectory similarity queries*, and *trajectory clustering*. Figure 3 and Table 2 summarize the representative measures based on Euclidean space that we focus on. As the figure shows, the learned measures dominate the recent literature.

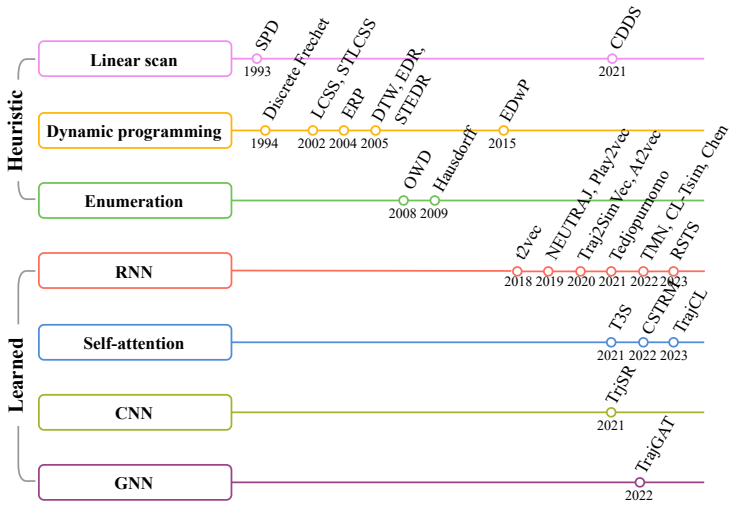


Fig. 3. Representative trajectory similarity measures plotted in chronological order

3.1 Heuristic Trajectory Similarity Measures

Heuristic measures are generally based on point matches between two trajectories that are computed following hand-crafted rules [5, 6, 12, 16, 17, 47]. The similarity between two trajectories is derived from the distances between the matched point pairs. Based on how the point matches are

Table 2. Categorization of representative trajectory similarity measures in Euclidean space

Category	Methodology	Measure	Time Complexity	Space Complexity
Heuristic measures	Linear scan	SPD [4], CDDS [12]	$O(n)$	$\Theta(1)$
	Dynamic programming	EDR [17], ERP [16], EDwP [47], LCSS [56], DTW [36], Discrete Fréchet [26], STEDR [53], STLCSS [53]	$O(n^2)$	$\Theta(n)$
	Enumeration	OWD [40], Hausdorff [5]	$O(n^2)$	$\Theta(1)$
Learned measures	Recurrent neural network	t2vec [39], NEUTRAJ [69], Play2vec [62], At2vec [41], Traj2SimVec [74], Tedjopurnomo [53], Chen [18], CL-Tsim [22], TMN [67], RSTS [19]	$\Omega(nd^2)$	$\Omega(d^2)$
	Self-attention neural network	T3S [68], CSTRM [42], TrajCL [11]	$\Omega(n^2d)$	$\Omega(d^2 + nd + n^2)$
	Convolutional neural network	TrjSR [9]	$\Omega(mk^2n_kc)$	$\Omega(k^2n_kc + mc)$
	Graph neural network	TrajGAT [70]	$\Omega(nn_ed)$	$\Omega(d^2 + nd + nn_e)$

computed, we categorize the heuristic measures into three classes: (i) *linear scan-based*, (ii) *dynamic programming-based* and (iii) *enumeration-based* measures.

Linear scan-based measures. Linear scan-based measures [4, 12] take only a single scan over two trajectories to compute their similarity. An example is shown in Figure 4, where each gray dotted line denotes a pair of matched points. Such measures take $O(n)$ time to compute, assuming n points per trajectory. The measures take $\Theta(1)$ space, to store the partial similarity results, excluding the $O(n)$ space to hold the input trajectories (same below).

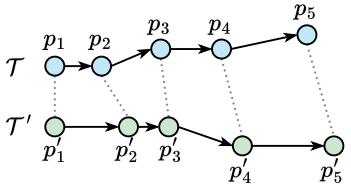


Fig. 4. Computation of linear scan-based measures

For example, the *Sum-of-Pair Distance* (SPD) [4] simply matches the points on two trajectories following the order of the points and sums up the pairwise point distances. It requires the trajectories to have the same number of points. The *Close-Distance Duration Similarity* (CDDS) [12] further considers the time dimension. It scans the points on two trajectories and sums up the time span when the points on both trajectories are within a given spatial distance threshold. The resulting sum is used as the similarity.

Dynamic programming-based measures. While being simple and efficient, the linear scan-based measures may compute sub-optimal point matches and hence result in falsely large trajectory similarity values. For example, SPD performs sequential matching that does not consider the spatial proximity between the points. To address the issue, dynamic programming (DP)-based measures are proposed to explore a larger point-matching space while confining the computation costs. As Figure 1 shows, such measures examine all point pairs incrementally with DP in $O(n^2)$ time, taking $\Theta(n)$ space to store the intermediate similarity (distance) values.

For example, *Dynamic Time Warping* (DTW) [36], which was designed for time series similarity computation, has been extended to trajectory data. DTW computes a set of point alignments between two trajectories that achieves a global minimum sum of point-to-point distances. It allows many-to-one point matching, and hence it does not require trajectories of the same length. *Discrete Fréchet* [26] also allows many-to-one point matching. It returns the maximum distance between any matched point pairs.

Longest Common Sub-Sequence (LCSS) [56] adapts a string similarity measure for trajectory data. It computes the longest common sub-sequence of two trajectories. Here, a common sub-sequence refers to consecutive pairs of points that are within a given spatial distance threshold. Another series of studies adapt edit distance. They compute the cost to "edit" (insert, delete, or substitute)

points on a trajectory to match those in the other trajectory. For example, *Edit Distance on Real sequence* (EDR) [17] considers a same unit cost for each edit. Next, *Edit Distance with Real Penalty* (ERP) [16] factors the point distances into the edit costs. Further, *Edit Distance with Projections* (EDwP) [47] uses an interpolation-style insertion operation. It adds points on the line between two adjacent points of a trajectory when point insertions are needed to form matches with another trajectory. A few other measures consider both spatial and temporal distances, such as STEDR and STLCSS [53], which extend EDR and LCSS by adding a temporal distance threshold when matching the points, respectively.

Enumeration-based measures. These measures compute all pairwise point distances directly and aggregate them to form a trajectory similarity (cf. Figure 1). Such measures take $O(n^2)$ time and $O(1)$ space, as they do not need to store intermediate results as do the DP-based measures.

For example, the *One Way Distance* (OWD) [40] uses the average point-to-trajectory distance as the distance between two trajectories. The point-to-trajectory distance here is the minimum distance between a point to any point on a trajectory. *Hausdorff* [5] uses the maximum point-to-trajectory distance instead of the average.

Discussion. We note that some heuristic measures have approximate algorithms (e.g., *Approx-DTW* [72] and *aprxFréchetI* [25]) with lower running times. The comparison between non-learning-based and learning-based approximations is not the focus of our study. We leave such a comparison for future work.

3.2 Learned Trajectory Similarity Measures

Studies in the past five years focused on deep learning models to encode trajectories and subsequently learn trajectory similarity [9, 11, 39, 67–70, 74]. These studies can be categorized into two classes according to their design purposes: (i) to learn and approximate existing heuristic measures [68–70], and (ii) to learn latent similarity measures that are independent from any heuristic measures [9, 11, 39]. The former class of studies use supervised learning, where some existing heuristic measure is used to provide the supervision signals. The latter class of studies, on the other hand, leverages self-supervised learning to learn trajectory embeddings. The similarity between two trajectories are calculated based on their learned embeddings, e.g., using the L_1 distance.

In both classes of studies, a core component is the *backbone trajectory encoder*, which takes a trajectory as input and converts it into an embedding in a latent space. The backbone encoders play a central role in the learned trajectory similarity measures. Thus, we review the learned measures based on the backbone encoders used.

Recurrent neural network (RNN)-based measures. RNNs are popular for sequential data modeling. Since trajectories are sequences, RNNs form a natural backbone trajectory encoder. RNNs encode each trajectory point recurrently by considering both the historical states (i.e., aggregated information from preceding points) and the current state (i.e., the current point to be processed). When an RNN completes its computation, the final output state entails aggregated information from all points on a trajectory, which is used as the trajectory embedding, i.e., $\mathbf{h}$ in Figure 5.

RNN-based trajectory similarity measures (using Long Short-Term Memory (LSTM) [34] or Gated Recurrent Unit (GRU) [20]) take at least $\Omega(nd^2)$ time to encode a trajectory because they need to compute the hidden representation of each of the n states (for the n points on a trajectory), while each hidden representation takes $\Theta(d^2)$ time to compute (i.e., hidden feature mapping). Here, d is the embedding dimensionality. Some approaches may take more time than $\Theta(nd^2)$ time, e.g., $\Theta(nd^2 + n^2d)$ time for TMN [67]. The RNN-based measures take $\Omega(d^2)$ space for the model parameters (i.e., $d \times d$ weight matrices) and another $\Omega(d)$ space to store the intermediate results during the embedding computation process.

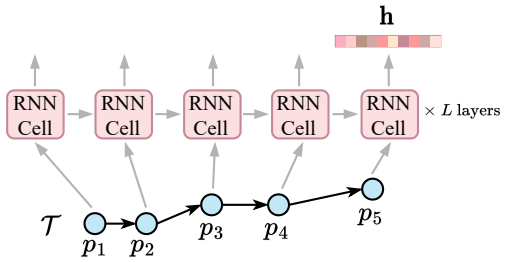


Fig. 5. RNN-based learned measures

Most RNN-based studies use a grid cell-based input representation. They partition the data space with a grid and transform a trajectory from a sequence of points to a sequence of grid cells enclosing the points. This transformation has two benefits: (1) The cell-based representation helps reduce the input space from points in a continuous space to a small number of discrete cells. This creates an input data distribution that is easier to be learned. (2) As a side effect, the cell-based representation helps alleviate the impact of GPS errors and varying trajectory sampling rates. This helps learn more robust embeddings.

The first RNN-based trajectory similarity model is *t2vec* [39]. The model adapts GRU by introducing a spatial proximity aware loss function that penalizes the model when it generates large similarity prediction errors on trajectories that are spatially close to each other. *NEUTRAJ* [69] uses GRU (according to its released code) and adds a spatial attention memory unit, such that the representation learning process for a trajectory can refer to spatially close trajectories seen by the GRU before. Further, NEUTRAJ has a weighted ranking loss function to encourage model learning from the most similar trajectory pairs. *Traj2SimVec* [74] improves upon NEUTRAJ in both training sample construction and spatial distance learning: (1) Traj2SimVec leverages a k-d tree [8] to select a set of most similar trajectories as the positive training samples, rather than randomly sampling from the whole trajectory dataset as in NEUTRAJ. Hence, it achieves better training efficiency. (2) Traj2SimVec uses a sub-trajectory-based loss to learn the detailed alignment and distances between points, while the loss function of NEUTRAJ is based on the embeddings of full trajectories. Chen et al. [18] also build upon NEUTRAJ. They use an interpolation-based trajectory calibration process to generate smoother trajectories for model training. *CL-Tsim* [22] adopts contrastive learning to help generate more diverse training samples so as to obtain more robust embeddings.

Unlike the models above that learn to encode each trajectory separately, *TMN* [67] introduces a dual-branch model to learn trajectory embeddings and point matches at the same time. Its matching module aims to simulate the computation process of the heuristic measures. The backbone encoder model of TMN requires two input trajectories. It cannot be used as a standalone trajectory backbone encoder to encode individual trajectories. Thus, this model cannot be used for the kNN queries experiments in Section 5.3.

Besides, Tedjopurnomo et al. [53] and Li et al. (i.e., *RSTS*) [19] adapt t2vec to measure spatio-temporal trajectory similarity. RSTS simply introduces a three-dimensional grid cell where the third dimension models the time. Tedjopurnomo et al. introduce three loss functions to jointly learn trajectory similarity at different levels, i.e., trajectory, point, and pattern levels. Finally, there are several studies adopting RNN models to learn similarity for special types of trajectories, such as point of interest (POI) trajectories (*At2vec*) [41] and sports play trajectories (*Play2vec*) [61, 62].

Self-attention-based measures. *Multi-head Self-attention Mechanism* ("self-attention" in short) [55] is a more recent sequential model that addresses the catastrophic forgetting issue of the RNNs (i.e., forgetting previously learned information as the input sequence gets longer). It learns the hidden

correlation between every two elements in an input sequence (cf. Figure 6). It takes $\Omega(n^2d)$ time to encode a trajectory of n points. While this seems to be higher than the time taken by RNN models, self-attention models may run much faster than RNN models on GPU. This is because self-attention models run in only one round to compute a sequence embedding, which is highly parallelizable. In contrasts, RNNs need to run n rounds of computation iteratively due to their recurrent structures. Self-attention-based measures take $\Omega(d^2 + nd + n^2)$ space, where the model parameters (i.e., weight matrices) take $\Omega(d^2)$ space, and the intermediate results for computing the attention coefficients (between the n^2 pairs of points) take $\Omega(nd + n^2)$ space.

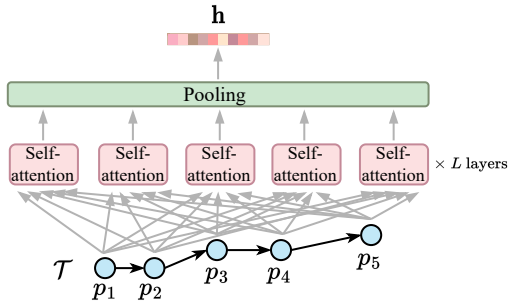


Fig. 6. Self-attention-based learned measures

A few studies have applied self-attention to trajectory similarity learning. *CSTRM* [42] leverages Masked Language Modeling [23] to learn trajectory embeddings in a self-supervised manner. The trained trajectory encoder can be further fine-tuned to approximate a given heuristic measure. *T3S* [68] combines self-attention and LSTM to capture the topological and the spatial features of trajectories, respectively. Later, *TrajCL* [11] introduces a fully self-attention-based trajectory encoder that adaptively learns the topological and spatial features. It lifts the dependence on recurrent structures.

Convolutional neural network (CNN)-based measures. CNN models are widely used in image representation learning. They stack convolution kernel layers and pooling layers to capture image features. Trajectories can be converted to images. To convert a trajectory to a fixed-size image, a blank image corresponding to the data space enclosing the trajectory is first created. Then, the points on the trajectory are mapped to pixels of the image, where a pixel value indicates the number of points mapped to the pixel (cf. Figure 7). Such a conversion resembles the grid-cell based trajectory representation described earlier and shares similar benefits.

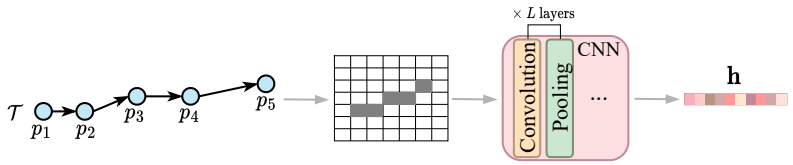


Fig. 7. CNN-based learned measures

TrjSR [9] is the only model using a CNN-based backbone encoder. This model is trained by reconstructing a super-resolution trajectory image from a low-resolution one. It takes $\Omega(mk^2n_kc)$ time to encode a trajectory, where k is the side length of the convolution kernels, n_k is the number of kernels, c is the channel size, and $m \gg d$ is the image size (number of pixels). It takes $\Omega(k^2n_kc + mc)$ space, where the model parameters (i.e., weight matrices) take $\Omega(k^2n_kc)$ space, and the intermediate results take $\Omega(mc)$ space.

An issue with the CNN-based measures is that they lose the sequence information of the trajectory points. They cannot distinguish two trajectories traveling towards the opposite directions.

Graph neural network (GNN)-based measures. GNNs are designed for graph representation learning. State-of-the-art GNN models are built upon the message passing mechanism. In each GNN layer, every graph node receives and aggregates information (typically node embeddings) from its neighbors (aggregation). Then, the aggregated information is combined with the embedding of the node to form its updated embedding (combination).

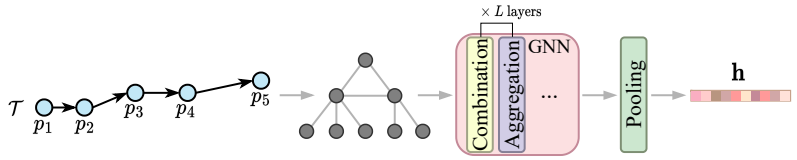


Fig. 8. GNN-based learned measures

TrajGAT [70] is the only GNN-based measure (cf. Figure 8). It builds a multi-level quadtree [48] that partitions the space into cells with different sizes to help create graphs with multi-granularity views of trajectories. To construct a graph from a trajectory, it queries each trajectory point in the quadtree and adds the tree nodes on the traversed path into the graph as graph nodes. The edges between the added tree nodes are kept as graph edges. To encode a trajectory graph, TrajGAT takes $\Omega(nn_ed)$ time, where n_e is the number of neighbors per node in the graph. TrajGAT takes $\Omega(d^2+nd+nn_e)$ space, where the model parameters take $\Omega(d^2)$ space, and the intermediate results for computing the node embeddings and the attention coefficients between nodes take $\Omega(nd+nn_e)$ space.

Discussion. As mentioned earlier, we focus on measuring trajectory similarity in Euclidean space. We note a few other studies considering road network settings [13, 28, 29, 32, 77]. We leave an empirical analysis on these studies for future work.

3.3 Trajectory Similarity Queries

Trajectory similarity queries typically refer to trajectory k nearest neighbor (kNN) queries (Section 2). All existing trajectory kNN query algorithms are designed for the heuristic measures. They query data trajectories from tree-based indies (e.g., R-trees [31]), with spatial distance-based pruning to speed up the search.

Recent studies [49, 66] introduce generic data structures and query algorithms for trajectory kNN queries. *DFT* [66] is a distributed segment-based index, which supports the Hausdorff and Fréchet measures. It indexes trajectory segments with an R-tree, where each tree node is associated with a bitmap [10] that records the trajectory IDs corresponding to the segments in the node. The kNN algorithm of DFT leverages a small set of sampled data trajectories (more than k) to obtain a distance threshold ϵ that bounds the largest distance between the query trajectory $\mathcal{T}_q$ and its k-th NN. Then, the DFT index is queried. If a data segment has a distance to $\mathcal{T}_q$ greater than ϵ, the corresponding trajectory is pruned.

DITA [49] is another R-tree-like index, which supports DTW and ERP. It indexes points sampled at equal intervals on each data trajectory, for space efficiency. The kNN algorithm of DITA also starts by computing a k-th NN distance threshold ϵ. Unlike DFT, DITA can shrink ϵ during its search process, by subtracting the distance between the currently matched points between a data trajectory and the query trajectory. This is because DTW and ERP compute trajectory distance by summing up the distances between the matched points. In contrast, Fréchet and Hausdorff compute the global maximum distance of the matched points, such that ϵ cannot be updated progressively as more points are seen for a trajectory.

3.4 Trajectory Clustering

Earlier studies on trajectory clustering are based on heuristics [15, 21, 37, 38]. They apply classic clustering algorithms, e.g., k-medoids, with heuristic similarity measures (e.g., LCSS), For example, Lee et al. [37] presents a partition-and-group framework for trajectory clustering. Li et al. [38] focus on finding top-k clusters considering the cluster cardinality. Besides, a series of studies [15, 21, 38] consider online trajectory clustering settings.

Recent studies [14, 27, 30, 45, 71, 73] exploit deep learning to improve clustering efficiency and effectiveness. Yao et al. [71] first leverage an RNN auto-encoder for trajectory clustering. They first learn trajectory embeddings and then group the embeddings into clusters by the k-medoids algorithm. This clustering paradigm is followed by the later studies. *DETECT* [73] and *Trip2Vec* [14] further consider POIs on trajectories to study the mobility pattern of trajectories. *E2DTC* [27] adopts t2vec [39] as the backbone encoder and fine-tunes the encoder with a multi-task loss function that considers both trajectory similarity and cluster distribution. Besides, several studies [45, 60] extend deep trajectory clustering to different data domains, e.g., aircraft or vessel trajectories.

4 GPU-BASED HEURISTIC MEASURE IMPLEMENTATION

We present three *meta-algorithms* for GPU-based implementations of the three types of heuristic trajectory similarity measures as described in Section 3.1, to enable empirical comparisons with the learned measures on GPUs.

4.1 Parallelizing Linear Scan-Based Measures

Linear scan-based heuristic measures first pair up points from two trajectories sequentially, e.g., by point indices (in SPD) or timestamps (in CDDS). Then, they compute a similarity score (distance) for each point pair and aggregate the scores to obtain the overall trajectory similarity score. As the similarity scores of the point pairs are independent from each other, we can simply partition each trajectory into sub-trajectories and parallelize the processing of the sub-trajectories. We name such an algorithm **`par-scan`** and illustrate it with Figure 9, where the two input trajectories $\mathcal{T}$ and $\mathcal{T}'$ are divided into three pairs of sub-trajectories sequentially. We distribute each pair onto a computation core (i.e., a GPU core, same below), where $core_i$ denotes the i-th core, and each pair is processed concurrently. Each core computes the distances for at most $\max(|\mathcal{T}|, |\mathcal{T}'|)/n_c$ point pairs, assuming n_c cores on the GPU. In the end, the result from each core is aggregated sequentially. We omit the pseudo-code as it is straightforward.

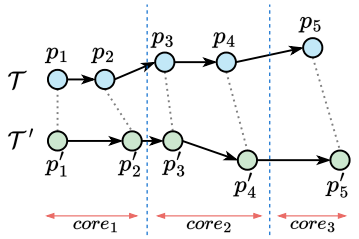


Fig. 9. Parallelization of linear scan-based measures

4.2 Parallelizing DP-Based Measures

Dynamic programming (DP)-based measures need to compute an optimal matching of the points from two trajectories and their similarity scores by filling up a DP *score matrix*. In general, the score matrix is computed row by row, and from the left to the right in each row. The last computed score in the matrix is returned as the final similarity score. Figure 10a illustrates the process,

assuming two trajectories of five points each, i.e., p_1 to p_5 and p'_1 to p'_5, respectively. A dot at row i, column j represents the intermediate similarity score between two partial trajectories $[p_1, p_2, \ldots, p_i]$ and $[p'_1, p'_2, \ldots, p'_j]$ (the actual similarity value has been omitted for simplicity). The blue arrow denotes the order of computation, and the bottom right dot represents the final similarity score between the two trajectories. Note the gray arrows between the dots. They show the sequential computational dependency between the intermediate similarity scores, which are the key challenge in parallelization.

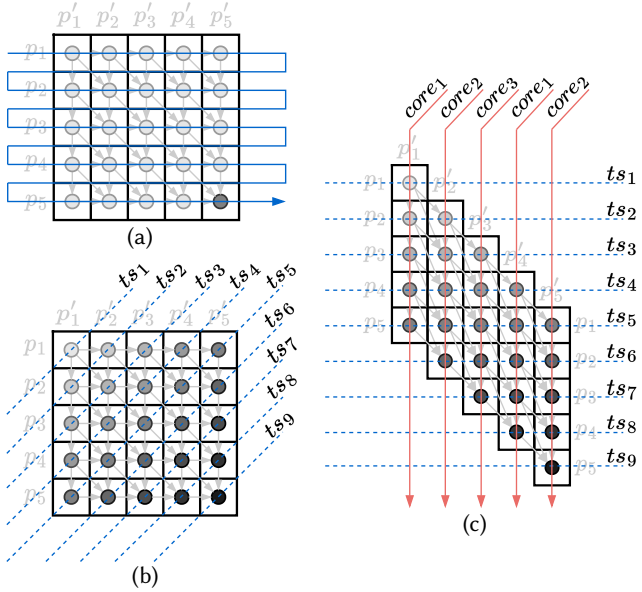


Fig. 10. Score matrix computation in DP-based measures: (a) single core strategy, (b) anti-diagonal computational dependencies, (c) parallelized strategy.

The parallelization strategy. To break the sequential computational dependency of the score matrix computation, we adapt the *anti-diagonal wavefront parallelization* strategy [24, 44, 50, 54, 65]. The core idea is to isolate the elements in the score matrix that are independent from the others, which enable parallel processing.

We use Figure 10b to illustrate the parallelization strategy. It is intuitive that the intermediate scores on the same blue dash line (an anti-diagonal, e.g., ts_2) are independent from each other, while they all depend on the scores to their upper left. Thus, the score matrix can be computed one anti-diagonal at a time, from the top left to the bottom right, i.e., from anti-diagonal ts_1 to anti-diagonal ts_9.

Figure 10c re-plots the score matrix following the processing order of the anti-diagonals strategy. The dots and the gray arrows still represent the scores and their computational dependencies, respectively. Only the positions of the intermediate scores have been reorganized such that their independent relationships become more obvious. Each column in this figure (which corresponds to partial trajectory $[p'_1, p'_2, \ldots, p'_j]$) can be computed by a different computation core in parallel, as long as the core for the j-th column starts one time step earlier than that for the $(j+1)$-th column.

When there are more cores than points on $\mathcal{T}'$, each point (i.e., a partial trajectory up to the point) is simply assigned to a different core. Otherwise, we take an interleaving strategy and assign one point on $\mathcal{T}'$ to a core sequentially until all cores have been assigned. We then repeat this process from the first core again, as the figure shows (there are three cores). This strategy is better than

assigning a continuous segment of $\mathcal{T}'$ to a core, as it reduces the wait time for the cores assigned with the later points on $\mathcal{T}'$.

The parallel algorithm. We name such an parallel algorithm based on the anti-diagonals strategy **`par-DP`** and summarize it in Algorithm 1. Since each row of the DP score matrix in Figure 10c depends on the two rows above, we create a two-dimensional matrix of three rows and $|\mathcal{T}'|+1$ columns for the computation, denoted by M (Line 2). Here, an extra column has been added for ease of iterative computation. Note that sequential DP implementations only require two rows, as one of the read rows can be reused for writing. In parallel implementation, three rows are needed to avoid reading dirty data when multiple cores read and write the matrix concurrently. Matrix M is initialized as per the requirement of a target heuristic measure (Line 3), e.g., zeroing M for EDR.

Algorithm 1: Parallel DP-based measure (`par-DP`)

Input: $\mathcal{T}$ and $\mathcal{T}'$: two trajectories; n_c: the number of cores.
Output: trajectory similarity score

1 $n_1 \leftarrow |\mathcal{T}|, n_2 \leftarrow |\mathcal{T}'|$;
2 $M \leftarrow array([3, n_2+1])$; `// DP score matrix`
3 Initialize M as required by the target heuristic measure;
4 $n_{ts} \leftarrow n_1 + n_2 - 1$; `// No. of time slots (Figure 10c)`
5 **for** $ts \in [0, n_{ts})$ **do**
6 $\quad r\_row_1 \leftarrow ts\%3, r\_row_2 \leftarrow (ts+1)\%3$;
7 $\quad w\_row \leftarrow (ts+2)\%3$;
$\quad$ `/* The following` *while* `block is executed on` n_c `cores in parallel, where` $cid \in [0, n_c)$ `is the sequence number of the current core. */`
8 $\quad$ **while** $cid \in [0, n_2)$ **do**
9 $\quad\quad$ **if** $cid \leqslant ts < cid + n_1$ **then**
10 $\quad\quad\quad$ Compute *subcost* based on $M[r\_row_1, cid]$, $M[r\_row_2, cid]$, and $M[r\_row_2, cid+1]$ following rules of the target heuristic measure;
11 $\quad\quad\quad M[w\_row, cid+1] \leftarrow subcost$;
12 $\quad\quad cid \leftarrow cid + n_c$;
13 $\quad$ Synchronize;
14 **return** $M[(n_{ts}+1)\%3, n_2]$;

The score matrix is computed in $n_{ts} = |\mathcal{T}| + |\mathcal{T}'| + 1$ time slots (for the n_{ts} anti-diagonals, cf. Figure 10c). At time slot ts, scores in the $(ts\%3)$-th and $[(ts+1)\%3]$-th rows of matrix M are read, which are used to compute the scores for the next anti-diagonal, to be stored in the w_row-th row (Lines 5 to 7). For each time slot, n_c cores compute in parallel for the up to $|\mathcal{T}'|$ values of an anti-diagonal. The i-th value is computed by the $(i\%n_c)$-th core based on the i-th value in the second read row and the $(i-1)$-th values of both read rows (Lines 8 to 12). The exact computation depends on the target heuristic measure (e.g., maximization for LCSS) and is not our focus. After the computation, we need a synchronization step to ensure that all cores have completed their computations (Line 13). When all ts time slots are computed, the element at the last column of the last write row is the target similarity score (Line 14).

4.3 Parallelizing Enumeration-Based Measures

Enumeration-based measures check all combinations of point pairs. Like in the linear scan-based measures, there is no computational dependency between the similarity scores of any two pairs

of points. The computation of the scores is thus embarrassingly parallel. We name the parallel algorithm **par-enum** and illustrate it with Figure 11. The pseudo-code is omitted as it is straightforward. As the figure shows, we distribute the computation of the similarity scores onto $n_c = 3$ cores where each core computes the scores for $|\mathcal{T}'|/n_c$ points on $\mathcal{T}'$. Here, the interleaving strategy is used again, while parallizing by continuous segments will work just the same. All similarity cores obtained are harvested to compute the final trajectory similarity on a single core (e.g., the CPU) following the definition of a target enumeration-based measure.

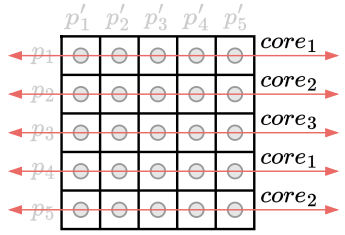


Fig. 11. Parallelization of enumeration-based measures

5 EXPERIMENTS

We study the performance of both the heuristic and the learned measures empirically with identical hardware settings, for three representative tasks: trajectory similarity computation, clustering, and similarity querying. We also explore an empirical lower bound of the computational time required by learning-based measures, to provide a reference for future studies.

5.1 Experimental Setup

5.1.1 Datasets. We use three real-world trajectory datasets, which are widely used in recent studies [9, 11, 22, 67–70]. We pre-process the datasets by discarding short trajectories with less than 20 points, as done in previous studies [9, 11, 19, 39, 67, 69]. Table 3 summarizes dataset statistics after pre-processing.

Table 3. Dataset statistics

	Porto	Geolife	Xi'an
#trajectories	1,380,777	15,972	1,009,693
#points per trajectory	20 ~ 3,836	20 ~ 56,780	20 ~ 8,730
Avg. #points per trajectory	50	1,201	262
Trajectory length (meters)	8 ~ 110,227	1 ~ 556,814	23 ~ 219,824
Avg. trajectory length (meters)	6,445	25,175	6,204
Spatial area (km^2)	16.0 × 20.1	235.9 × 232.5	9.9 × 9.6
Time span	-	five years	a week

Porto [1] contains 1.3 million taxi trajectories collected from Porto, Portugal, between July 2013 and June 2014. The trajectory points do not come with timestamps. The average number of points per trajectory is 50, which is the smallest among the three datasets.

Geolife [76] contains 15,972 user trajectories collected in Beijing, China, from April 2007 to August 2012. We have further discarded the trajectories recorded outside Beijing, which take up a very small portion (6%) of the data. Geolife records trajectories of different types of user movements, e.g., walking, cycling, and driving. The dataset also covers the largest spatial area among the three datasets, i.e., over 500 km^2. Its trajectories have the largest average number of points, i.e., 1,201, and the longest average length, i.e., 25 km.

Xi'an [2] contains 1 million ride-hailing trajectories collected mainly within the Second Ring Road of Xi'an, China, in the first week of October 2018. This dataset covers the smallest area among the three datasets but is also the most current dataset. Its sampling rate is the lowest, i.e., roughly 30 meters per sampled point.

5.1.2 Similarity Measures. For each category of measures as summarized in Table 2, we study the state-of-the-art measure as well as measures that take substantially different computation strategies, such as NEUTRAJ and TMN, which are both in the RNN-based category but use single- and dual-branch models, respectively. The similarity measures tested are listed below, where "ST" refers to measures that consider both spatial distances and time differences.

(1) Heuristic measures: ① Linear scan: **CDDS** (ST) ② DP: **DTW**, **ERP**, **Fréchet**, and **STEDR** (ST) ③ Enumeration: **Hausdorff**.

(2) Learned measures: ① RNN: **NEUTRAJ** (single-branch), **TMN** (dual-branch), **RSTS** (ST) ② Self-attention: **T3S** (RNN + self-attention), **TrajCL** (self-attention) ③ CNN: **TrjSR** ④ GNN: **TrajGAT**.

5.1.3 Implementation Details. All experiments are run on a virtual machine with an 8-core Intel Xeon CPU, 128GB RAM and an NVIDIA Tesla V100 GPU (5,120 FP32 cores and 16GB VRAM). We repeat each experiment five times and report the average results.

For the heuristic measures, we use *traj-dist* [3], a commonly used CPU-based sequential implementation of trajectory similarity computation in Python. It supports DTW, ERP, Fréchet, and Hausdorff, while we add support for STEDR and CDDS.

We implement the GPU-based heuristic measures by following the meta-algorithms in Section 4 on CUDA cores of GPUs in Python with the Numba 0.53.1 library supporting GPU programming. We note that parallel implementations of some of the heuristic measures exist (e.g., [64]). We use our meta-GPU-based algorithms instead because these do not contain special optimizations for any specific measures, enabling us to capture better the speedups achievable by a simple GPU-based adaptation.

For the learned measures, we use their released source code which are written in PyTorch, except for T3S and RSTS for which no code is available. We implement T3S and RSTS with PyTorch 1.8.1 following their papers. PyTorch allows the models to be run on either CPU or GPU, with the same implementation.

Following previous studies [67–69], we set the dimensionality d of trajectory embeddings to 128. The side length of the grid cells is 100 meters for the grid-based learned measures. The image size m of TrjSR is set to 162×128 following the default setting in its paper. The number of encoder layers stacked in each learned measure is 2, except for TrjSR, which has 10 CNN layers following its paper. We set the number of GPU cores, n_c, for computing the similarity between a pair of trajectories to 64. The batch size is set to 512 by default, to fit every measure in memory.

5.2 Trajectory Similarity Computation

We first report the results on trajectory similarity computation.

5.2.1 Setup. For each dataset, we randomly sample 100,000 pairs of trajectories and compute their similarity using each measure. Each trajectory is limited to at most 200 points, following previous studies [11, 19, 67, 69, 70], as the learned measures may fail on longer trajectories due to out-of-memory errors (cf. Section 5.2.3). We run experiments by varying the data size of each run (i.e., **single** or **batched**) and the computation unit (i.e., **CPU** or **GPU**), to simulate different application settings. Here, "single" refers to computing the similarity for a single pair of trajectories (with a single core), while "batched" refers computing for multiple pairs (i.e., 512) of trajectories using multiple cores of a computation unit.

Table 4. Elapsed time of trajectory similarity computation (best results are in bold)

Dataset	Measure		Single		Batched	
			GPU	CPU	GPU	CPU
Porto	Heuristic	DTW	146.23	36.04	**3.96**	10.07
		ERP	190.76	78.46	4.09	12.88
		Fréchet	146.90	32.59	4.00	7.44
		Hausdorff	**135.71**	**26.64**	4.07	**6.57**
	Learned	NEUTRAJ	4907.14	3088.88	94.69	229.21
		TMN	389.63	535.48	54.67	254.23
		T3S	465.75	861.54	49.49	649.17
		TrajCL	561.94	666.82	59.49	276.30
		TrjSR	584.21	OT	301.74	4409.75
		TrajGAT	3319.11	2580.19	494.34	603.32
Geolife	Heuristic	DTW	165.68	130.19	**5.68**	24.70
		ERP	212.14	283.55	5.94	48.91
		Fréchet	165.91	91.40	5.82	23.02
		Hausdorff	**140.97**	**47.29**	5.76	**16.95**
	Learned	NEUTRAJ	4182.19	2682.02	118.14	306.60
		TMN	524.18	608.16	68.64	414.70
		T3S	618.42	1242.94	93.35	833.46
		TrajCL	613.15	711.60	88.28	425.05
		TrjSR	588.39	OT	299.55	4417.78
		TrajGAT	4375.38	3799.01	1335.27	1815.51
	Heuristic (ST)	STEDR	165.46	221.35	**6.50**	43.94
		CDDS	**138.17**	**24.74**	6.65	**17.35**
	Learned (ST)	RSTS	2970.58	3745.92	805.44	939.33
Xi'an	Heuristic	DTW	168.24	133.42	6.21	28.11
		ERP	215.28	362.92	6.34	54.04
		Fréchet	169.60	117.20	6.49	25.66
		Hausdorff	**145.41**	**70.63**	**5.83**	**17.75**
	Learned	NEUTRAJ	4226.71	2609.39	102.85	215.03
		TMN	394.89	463.79	62.08	230.18
		T3S	639.53	1958.81	83.69	778.61
		TrajCL	622.56	733.51	71.37	298.11
		TrjSR	625.54	OT	318.97	4629.79
		TrajGAT	4439.46	4114.52	1112.02	1560.24
	Heuristic (ST)	STEDR	171.03	280.66	7.25	48.09
		CDDS	**142.51**	**33.77**	**7.06**	**22.66**
	Learned (ST)	RSTS	3736.67	6030.93	1257.75	1473.71

Note that, when we compute trajectory similarity in batches, different "batched" computation paradigms are applied on GPU and CPU, respectively. On GPU, we use n_c cores for parallel processing of each trajectory pair following our meta-GPU-based algorithms. On CPU, we simply use a computation core for each trajectory pair, i.e., no parallel processing is done on individual trajectory level. This different setting is because there are much more cores on GPU than on CPU [63]. In batch processing mode, all CPU cores can be fully utilized by the trajectory pairs in a batch already, while the cores on GPU can be shared at the individual trajectory level.

We vary the number of points in trajectories, the dimensionality of trajectory embeddings, and the number of trajectory pairs.

We report the **elapsed time** (in seconds) and the **space cost** (in GB). Next, we report the hit ratios **HR@50** to study how accurate the learned measures are when approximating a heuristic measure, i.e., the fraction of the ground-truth top-50 most similar trajectories in the predicted top-50 trajectories [11, 68–70].

All input data is aligned to the same form, i.e., raw 2-dimensional trajectory points, and the outputs are similarity scores. The cutoff running time is 7,200 seconds. We use "**OT**" to denote overtime errors and "**OOM**" to denote out-of-memory errors.

5.2.2 Results under Online Computation Settings. Table 4 shows the computation times when all computation is done online, including trajectory embedding. *Overall, the heuristic measures take less time to compute than the learned ones when they are run online on the same computation units.* Even in the setting where the learned ones are supposed to be at their best, i.e., batched computation on GPU, the heuristic measures are still at least an order of magnitude faster. The performance gap is as large as 234 times (5.68 vs. 1335.27 for DTW vs. TrajGAT on Geolife).

A few detailed observations can be made from the table:

(1) **Both heuristic and learned measures take the least time for batched computation on GPU.** This setting best exploits the parallelization power of GPU. Such a setting thus should be used to compute the similarity of a large number of trajectories (e.g., for offline trajectory mining tasks such as contact tracing).

(2) **When computing the similarity between trajectories on per pair basis ("single"), e.g., for an ad hoc similarity computation requirement, the heuristic measures prefer CPU while the learned ones prefer GPU.** The heuristic measures are relatively lightweight, such that the savings achieved by parallelization for a single pair of trajectories is not worth the extra data transfer costs between CPU and GPU. The learned measures, on the other hand, take less time on GPU than on CPU, as matrix multiplications in the trajectory encoders are better suited to GPUs.

(3) **Among the heuristic measures, the enumeration-based one, Hausdorff, is the fastest in general** (excluding the spatio-temporal measures), due to its simple computation rules. When computed on GPU, Hausdorff further benefits from its independent calculation of the pairwise point similarity scores, which can be fully parallelized. Among the DP-based measures, ERP generally takes the most time, as it has the most complex computation rules. We note that all heuristic measures are very fast and have similar elapsed times under the GPU-batched mode, while DTW reported marginally faster elapsed time on Porto and Geolife. When spatio-temporal measures are considered, the linear scan-based method CDDS is the fastest on GPU for its simple calculation, while it is still slower than Hausdorff for batched computation, because it needs to take the time factors into computation.

(4) **Among the learned measures, the attention-based ones T3S and TrajCL are more efficient in time** – T3S is the slower among the two as it also uses an RNN component. This is because the computation steps of the attention module can be fully paralellized as discussed in Section 3. An RNN-based measure, TMN, is the fastest among the learned measures, due to its simple model (a vanilla LSTM). NEUTRAJ and RSTS are also RNN-based, while they suffer in efficiency especially on the "single" mode. NEUTRAJ has an expensive spatial module to compute the attention coefficients between the current and the seen training trajectories (cf. Section 3), while RSTS has an expensive input pre-processing step (cf. Table 5). The CNN-based measure TrjSR and the GNN-based measure TrajGAT are also slow. TrjSR suffers in the large amount of computation for its CNN component, while TrajGAT spends much time on converting a trajectory into a graph.

Table 5. Detailed time and space costs of batched trajectory similarity computation on Xi'an

Measure	Time on GPU				Time on CPU				Space
	Pre.	Emb.	Cmp.	Total	Pre.	Emb.	Cmp.	Total	(GB)
DTW	5.64	-	0.56	6.21	0.65	-	27.45	28.11	**0.02**
ERP	5.72	-	0.62	6.34	0.52	-	53.51	54.04	**0.02**
Fréchet	5.91	-	0.58	6.49	0.50	-	25.16	25.66	**0.02**
Hausdorff	5.28	-	0.55	**5.83**	0.44	-	17.30	**17.75**	**0.02**
NEUTRAJ	68.15	34.69	1e-4	102.85	67.83	147.20	2e-3	215.03	0.13
TMN	56.10	5.97	1e-4	62.08	56.18	174.00	2e-3	230.18	6.37
T3S	59.73	23.95	1e-4	83.69	58.27	720.34	2e-3	778.61	3.02
TrajCL	60.38	10.99	1e-4	71.37	57.79	240.30	2e-3	298.11	5.81
TrjSR	86.56	232.40	1e-4	318.97	90.08	4539.71	2e-3	4629.79	9.12
TrajGAT	1035.67	76.35	1e-4	1112.02	1028.66	531.58	2e-3	1560.24	13.89
STEDR (ST)	6.72	-	0.53	7.25	0.83	-	47.26	48.09	**0.02**
CDDS (ST)	6.49	-	0.57	**7.06**	0.60	-	22.06	**22.66**	**0.02**
RSTS (ST)	1239.22	18.53	1e-4	1257.75	1245.93	227.78	2e-3	1473.71	0.76

Computation time decomposition. We further "zoom in" on the time costs. The time of a similarity computation (i.e., those reported in Table 4, denoted by "**Total**") mainly consists of three parts: (1) the input pre-processing time (denoted by "**Pre.**") to convert an input raw trajectory into the format required by a measure (e.g., a graph for TrajGAT), (2) the embedding time (denoted by "**Emb.**") to compute trajectory embeddings (inapplicable to the heuristic measures), and (3) the similarity computation time (denoted by "**Cmp.**") after the data preparation steps above. There is also time for transferring the results and other minor inter-step processing, which is very small and hence omitted from the table. We only show the results on Xi'an in Table 5, as similar result patterns are observed on the other two datasets. *Overall, although the heuristic measures take more time than the learned measures on similarity computation, they have better efficiency on input pre-processing and do not require trajectory embedding, which explains for their smaller total elapsed times.*

The heuristic measures take more time for input pre-processing on GPU than on CPU. This is because when preparing data on GPU, we need to pad the trajectories in a batch to the same length and group them into a matrix, which is required by CUDA.

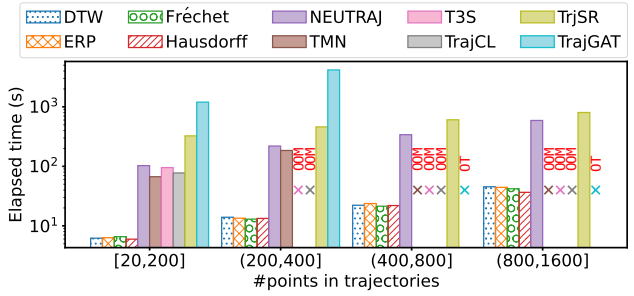
Table 6. Maximum batch sizes for different measures on Xi'an

DTW:	100,000	NEUTRAJ:	100,000	STEDR:	100,000
ERP:	100,000	TMN:	2,048	CDDS:	100,000
Fréchet:	100,000	T3S:	2,048	RSTS:	16,384
Hausdorff:	100,000	TrajCL:	2,048		
		TrjSR:	512		
		TrajGAT:	512		

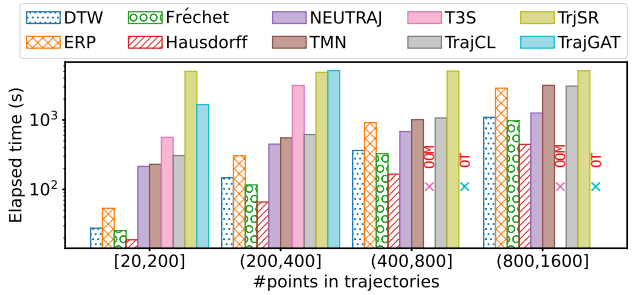
Memory costs. From Table 5, we also see that the heuristic measures take less memory space, as they do not need to store the large weight matrices.

We further show the maximum number of trajectory pairs that each measure can process in parallel given the same GPU memory (16 GB VRAM in our experiments). We vary the batch size from 256 and double it until reaching 100,000 which is the number of trajectories in the dataset. Table 6 shows the results on Xi'an. Overall, the heuristic measures allow a larger batch size than the learned ones except NEUTRAJ which also has a good space efficiency (for its simple RNN structure). Even so, the heuristic measures take less memory than NEUTRAJ, i.e., 1 GB vs. 12 GB.

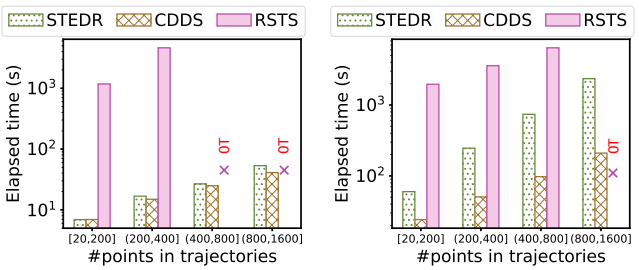
TMN, T3S, TrajCL, and TrajGAT need to compute the correlation between every two points, while TrjSR suffers from much intermediate computation, which explain for their higher memory costs.



(a) Spatial measures on GPU



(b) Spatial measures on CPU



(c) ST measures on GPU (d) ST measures on CPU

Fig. 12. Elapsed time vs. #points in trajectories

5.2.3 Impact of the Number of Points in Trajectories n. We vary the number of points in trajectories, n, from 20 to 1,600. We note that most existing studies on learned measures only used trajectories with up to 200 points. Ours is the first set of results on trajectories with over 1,000 points.

We focus on the batched setting on GPU and CPU as this is a more realistic setting in practice, and we omit the "single"-mode results as similar result patterns are observed as before. We again present the results on the largest dataset, Xi'an, as the other two datasets report results of similar comparative patterns.

Figure 12 shows the results, where each configuration, e.g., "[20, 200]" means to compute trajectory similarity for 100,000 pairs of randomly sampled trajectories each with $n = 20$ to 200. All measures except TrjSR have increasing computation times as n increases. The heuristic measures (denoted by empty bars with hatches) show clear advantage on GPU (Figures 12a and 12c), while the performance of both types of measures becomes closer on CPU when n becomes larger (Figures 12b

and 12d). TrjSR uses fixed-size images to represent trajectories, which is independent from n. TMN, T3S, and TrajCL trigger out of memory errors when n becomes large, especially on GPU. This is because TMN computes the attention coefficients between the points of two trajectories, while T3S and TrajCL compute the self-attention coefficients between every two points on each trajectory. Both types of attention coefficient computation lead to a quadratic space overhead with respect to n.

5.2.4 Impact of the Trajectory Embedding Dimensionality d. We vary d from 32 to 256 following the literature and report the results in Figure 13. As expected, the heuristic measures are not impacted by d, as they do not use embeddings. The computation time of the learned measures presents only a slightly increasing trend, as their matrix operations have been well parallelized by the PyTorch packages. The learned measures are still slower even when $d = 32$, which is the smallest d value in the literature [70]. We show only the results for the spatial measures on GPU for conciseness as the comparative patterns on CPU and the ST measures are similar.

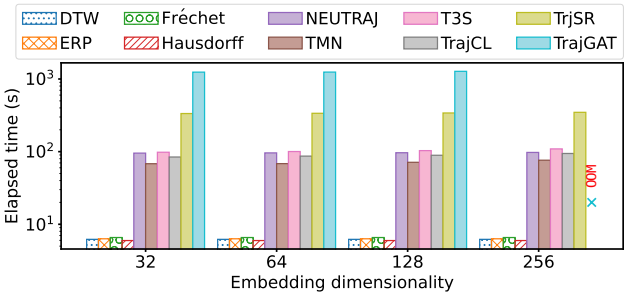


Fig. 13. Elapsed time vs. embedding dimensionality (spatial measures on GPU)

5.2.5 Impact of the Number of Trajectory Pairs. We further vary the number of trajectory pairs from 1,000 to 1,000,000. Again, the computation time grows with the number of trajectory pairs, and the heuristic measures outperform the learned ones. These results reinforce the advantage of the heuristic measures in computation efficiency. We omit the result figures due to space limit.

5.2.6 Results with Embedding Reuse. Existing studies [11, 68, 69] assume two sets of trajectories and report the time to compute the similarity between each pair of trajectories, one from each dataset. They can compute the embedding of each trajectory only once and reuse it for all similarity computations, which saves the computation time substantially (the pre-processing and embedding steps dominate their computation times, cf. Table 5). We repeat such a set of experiments to show that, when the embeddings can be pre-computed, the learned measures indeed can delivery their promised higher computation efficiency.

On each of the three trajectory datseats, we randomly sample and form two sets of 1,000 and 100 trajectories, respectively, such that we have 100,000 trajectory pairs to compute as before. We report the results of batched computation on Xi'an in Table 7. Now the learned measures TrajCL, T3S, and NEUTRAJ outperform the heuristic ones, benefiting from the one-off input pre-processing and embedding times. TrjSR and TrajGAT also benefit substantially. However, they are still slower than the heuristic ones because even the one-off preparations times are too expensive. These results are consistent with the literature [11]. TMN does not apply in this experiment, as it is a pairwise model and cannot be run on individual trajectories to pre-compute their embeddings.

Table 7. Detailed time and space costs of batched trajectory similarity computation between two trajectory sets on Xi'an

Measure	Time on GPU				Time on CPU				Space
	Pre.	Emb.	Cmp.	Total	Pre.	Emb.	Cmp.	Total	(GB)
DTW	0.10	-	0.58	0.68	0.01	-	28.32	28.33	**0.02**
ERP	0.10	-	0.70	0.80	0.01	-	55.10	55.11	**0.02**
Fréchet	0.10	-	0.57	0.67	0.01	-	26.03	26.04	**0.02**
Hausdorff	0.10	-	0.63	0.73	0.01	-	18.71	18.72	**0.02**
NEUTRAJ	0.49	0.46	1e-4	0.96	0.51	0.78	2e-3	**1.29**	0.13
TMN	-	-	-	-	-	-	-	-	-
T3S	0.48	0.03	1e-4	0.51	0.49	5.09	2e-3	5.63	3.02
TrajCL	0.47	0.02	1e-4	**0.49**	0.46	1.75	2e-3	2.21	5.81
TrjSR	0.62	0.90	1e-4	1.52	0.63	36.89	2e-3	37.52	9.12
TrajGAT	17.96	1.81	1e-4	19.77	18.23	17.30	2e-3	35.53	13.89
STEDR (ST)	0.14	-	0.59	0.73	0.02	-	41.85	41.87	**0.02**
CDDS (ST)	0.14	-	0.50	**0.64**	0.02	-	20.97	20.99	**0.02**
RSTS (ST)	4.98	0.08	1e-4	5.06	5.03	1.18	2e-3	**6.30**	0.76

5.3 Trajectory Similarity Queries

Next, we study the efficiency and effectiveness of trajectory similarity queries, i.e, trajectory kNN queries.

5.3.1 Setup. The default trajectory dataset D contains 100,000 randomly sampled trajectories of which the number of points is between 20 and 200. The query set Q contains 1,000 trajectories with the same length range as those in D that are randomly sampled outside D. The query parameter k is set as 50 by default. We vary the size of D and the number n of points in query trajectories.

We deploy *dedicated indices* for the heuristic and the learned measures, respectively, in order to provide a fairer comparison. This differs from the existing studies [39, 69] that simply use R-trees [7] to index the raw trajectories for query pruning and compute the trajectory similarities by the embeddings. For the heuristic measures, we use two recent indices, i.e., DITA [49] for DTW and ERP, and DFT [66] for Hausdorff and Fréchet, as detailed in Section 3.3. For the learned measures, we use Faiss [35] to index the embeddings and process kNN queries. Faiss is a widely used similarity search library for vectorized data. It is used because there are no existing indices for embedding-based trajectory kNN queries.

We report results on both query efficiency and effectiveness, as the effectiveness of the learned measures impact their applicability. Following the literature, we use hit ratios (HR@50) to measure the query accuracy of the learned measures. In addition, we report the cost of index construction. We use all learned measures expect TMN which cannot be applied in index-based kNN queries because it does not produce embeddings in advance. Like before, we focus on batched processing on Xi'an.

5.3.2 Overall Results. Table 8 summarizes the index building and kNN query time results. Overall, while the learned measures take more time on index building, they are faster at query processing.

Index building costs. The index build times of the learned measures are higher because they need to first encode raw trajectories into embeddings. In comparison, the heuristic measures index raw trajectory points or segments, and their indices are faster to build. DTW and ERP both use DITA indices, while Fréchet and Hausdorff both use DFT. Thus, the two pairs of measures share similar index build times. The DITA indices are the fastest to build, because they are built on only a

Table 8. *k*NN index building and query performance results

Measure	Index building		Query time (GPU)	Query time (CPU)
	Time	**Space** (GB)		
DTW	0.71	**1.60**	1193.12	5441.06
ERP	**0.69**	**1.60**	1304.40	6792.25
Fréchet	28.46	2.63	878.89	1783.95
Hausdorff	28.46	2.63	903.41	3410.28
NEUTRAJ	49.15	2.55	0.98	14.30
T3S	58.39	2.55	0.96	8.58
TrajCL	47.88	2.55	**0.89**	**4.33**
TrjSR	171.15	2.55	2.42	23.43
TrajGAT	661.69	2.55	6.62	60.45

few pivot points of each trajectory. Their space costs (i.e., the index size) are hence also the smallest. DFT indexes all trajectory segments which has even higher space costs than the embedding-based Faiss indices.

***k*NN query costs.** As for *k*NN query processing, the learned measures outperform the heuristic ones significantly, by some two orders of magnitude. For example, the fastest learned measure TrajCL is 984 times and 410 times faster than the fastest heuristic measure Fréchet when querying on GPU and CPU, respectively. This performance gap can be explained as follows. Both DFT and DITA are spatial indices, which suffer in their pruning capability when indexing objects with highly skewed aspect ratios, such as trajectories which are long and thin. The trajectories that cannot be pruned require expensive similarity computations with the heuristic measures. In contrast, the learned measures use Faiss which is a highly optimized system for vector-based similarity searches. The embeddings are pre-computed in index building, and the similarity scores are now computed by simple embedding scans, which is highly efficient. *This set of results confirm an important advantage of the learned measures, i.e., their embeddings can be indexed that enable extremely fast kNN queries.*

Table 9. *k*NN query accuracy (HR@50 of learned measures to approximate heuristic measures)

	DTW	**ERP**	**Fréchet**	**Hausdorff**
NEUTRAJ	**0.629**	0.417	0.671	0.678
T3S	0.514	**0.756**	0.756	0.657
TrajCL	0.528	0.421	**0.814**	**0.831**
TrjSR	0.521	0.377	0.630	0.757
TrajGAT	0.561	0.287	0.320	0.286

Overall *k*NN query accuracy. Table 9 shows HR@50 of the learned measures for *k*NN queries ($k = 50$). Intuitively, the values indicate how accurately the top 50 trajectories returned by the learned measures approximate those returned by the heuristic measures (recall that the learned measures are trained to approximate the heuristic ones). While none of the learned measures return fully accurate results, they can approximate Fréchet and Hausdorff reasonably well (i.e., HR@50 > 0.8). T3S is particularly strong at approximating ERP, outperforming the second best method TrajCL by 33%, while TrajCL is best for Fréchet and Hausdorff. The learned measures struggle at approximating DTW, where the highest HR@50 is only 0.637 (NEUTRAJ), which was also observed in previous studies [68, 70].

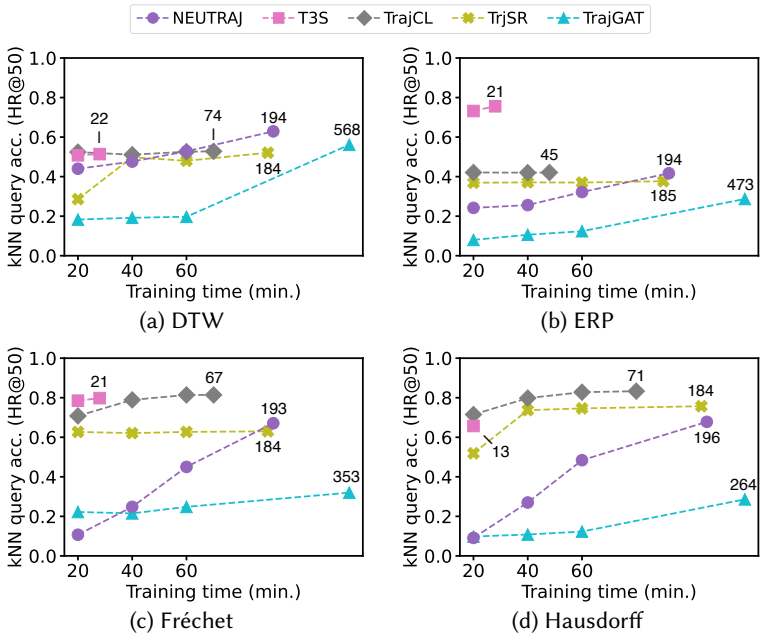


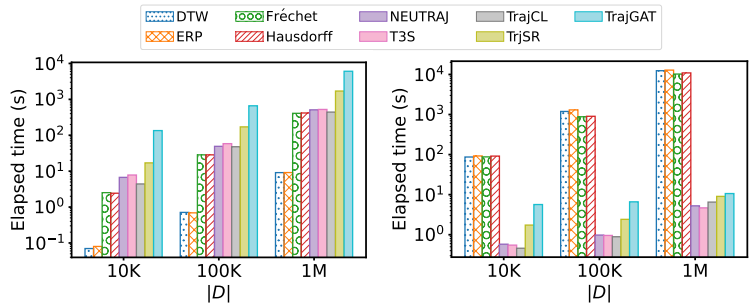
Fig. 14. Training time of the learned measures to approximate heuristic measures vs. kNN query accuracy (model convergence time is labeled in the figures)

***k*NN query accuracy vs. trajectory similarity model training time.** Figure 14 reports HR@50 of the learned measures for kNN queries ($k = 50$), where the model training time is constrained from 20 to 60 minutes. Note that the trajectory similarity model training time is excluded from the index building times reported above (and is excluded from all the other sets of experimental results), i.e., the last set of experiments assumes trained similarity measures. This set of results further provides guidance on trajectory similarity learning model selection for applications with time constraints on system preparation from scratch or model retraining.

The figure shows that when the model training time is very limited, e.g., 20 minutes, T3S is generally the best option to approximate most heuristic measures, except that TrajCL is 6% more accurate than T3S when approximating Hausdorff. When the training time increases to 40 or 60 minutes, TrajCL performs the best to approximate the heuristic measures except for the ERP (for which T3S remains the best). This is because T3S uses a vanilla attention model that is faster to train, while TrajCL uses a more complex multi-view attention model that takes more time to train but may produce higher accuracy given enough time. *These results also confirm the superiority of using attention models to learn trajectory similarity, especially when the model training time is limited.*

5.3.3 Impact of the Size of the Trajectory Dataset $|D|$. We vary the size of the trajectory dataset $|D|$ from 10^4 to 10^6. Here, we only report the results on GPU, as similar patterns are observed on CPU. As Figure 15a shows, the index build times of all measures grow roughly linearly with $|D|$. The indices of the heuristic measures are consistently faster to build, by up to three orders of magnitude (0.074 vs. 134.996 on DTW and TrajGAT when $|D| = 10^4$).

Figure 15b shows the kNN query times. Like in Table 8, the heuristic measures are one to three orders of magnitude slower than the learned ones. Importantly, as $|D|$ increases, the performance gap increases, attributing to the better pruning techniques of Faiss used for the learned measures. *Since there are no universally applicable trajectory similarity measures, unless accurate results based*



(a) Index build time (b) kNN query time

Fig. 15. kNN index building and query time vs. the size of the trajectory dataset

on a specific heuristic measure are required, the learned measures are the more practical choice for kNN queries.

5.3.4 Impact of the Number of Points in the Query Trajectories. We vary the number of points in the query trajectories from 20 to 1,600. The comparative performance between the heuristic and learned measures again resembles those reported earlier. We omit the result figure for conciseness.

5.4 Trajectory Clustering

5.4.1 Setup. We follow the commonly used trajectory clustering paradigm in the literature [71, 73]. For the heuristic measures, we apply the k-medoids algorithm [46] to cluster the trajectories based on their heuristic similarity. For the learned measures, we apply k-medoids on the learned trajectory embeddings.

The default trajectory dataset D consists of 1,000 trajectories that are randomly sampled from a dataset, where the number of points in trajectories n is between 20 and 200. The number of clusters is set to 10 by default (i.e., $k = 10$ in k-medoids).

We report the experimental results in terms of both clustering efficiency and effectiveness. Similar to kNN queries, the trajectory clustering based on the learned measures is an approximation of the clustering on the heuristic measures. We follow the literature and use the *rand index* (RI) to evaluate clustering accuracy. RI computes the percentage of the ground-truth similar trajectory pairs (derived based on heuristics measures) that are assigned to the same cluster. We again report results on Xi'an.

Table 10. Detailed time and space costs of batched trajectory clustering on Xi'an

Measure	Time on GPU				Time on CPU				Space
	Pre.	Emb.	Clst.	Total	Pre.	Emb.	Clst.	Total	(GB)
DTW	0.60	-	3.22	3.82	0.24	-	226.09	226.33	**0.02**
ERP	0.60	-	3.70	4.29	0.25	-	436.48	436.73	**0.02**
Fréchet	0.60	-	3.26	3.86	0.24	-	223.68	223.92	**0.02**
Hausdorff	0.61	-	3.37	3.98	0.25	-	167.60	167.85	**0.02**
NEUTRAJ	0.26	0.17	0.27	**0.70**	0.35	0.87	0.21	**1.43**	0.88
T3S	0.40	0.02	0.31	0.74	0.47	2.26	0.34	3.06	2.74
TrajCL	0.36	0.01	0.33	**0.70**	0.40	1.40	0.31	2.11	2.35
TrjSR	0.52	0.05	0.41	0.97	0.58	17.12	0.31	18.01	9.12
TrajGAT	6.82	0.35	0.22	7.39	7.56	6.87	0.32	14.75	2.08

5.4.2 Overall Results. Table 10 and Table 11 report the trajectory clustering efficiency and accuracy, respectively. *Overall, the learned measures achieve better time efficiency than the heuristic measures, while they take more memory space and serve inaccurate results.*

Clustering costs. We use the clustering time (denoted by **"Clst."**) to measure the time to cluster the raw trajectories with the heuristic measures or to cluster the trajectory embeddings for the learned measures. As Table 10 shows, clustering with the heuristic measures takes more time. The clustering process allows computing the embedding for just once, thus it is easy for the learning measures to offer a high computation efficiency (which also requires a higher space cost to store the embeddings).

Table 11. Clustering accuracy (RI of learned measures to approximate heuristic measures)

	DTW	ERP	Fréchet	Hausdorff
NEUTRAJ	**0.885**	0.811	0.811	**0.874**
T3S	0.832	0.816	0.823	0.820
TrajCL	0.816	**0.817**	**0.827**	0.811
TrjSR	0.708	0.718	0.746	0.799
TrajGAT	0.808	0.646	0.712	0.433

Clustering accuracy. Table 11 shows the clustering accuracy (i.e., RI) for the learned measures. Similar to the kNN results in Table 9, NEUTRAJ is strong for DTW, while TrajCL approximates ERP and Fréchet well, and no learned measures can offer exactly the same clustering results as those of the heuristic measures.

5.4.3 Impact of the Size of Trajectory Dataset $|D|$. We vary the size of the trajectory dataset $|D|$ from 100 to 10,000. Figure 16 shows the clustering time. With $|D|$ increasing, clustering takes more time with both types of measures. Clustering with heuristic measures generally consumes more time than that with the learned ones. For dataset as small as $|D| = 100$, the heuristic measures can be more efficient, outperforming TrjSR, and TrajGAT, because the embedding time dominates in this setting.

We also varied the number of points in trajectories. Since the comparative patterns have not changed, we omit the result figure.

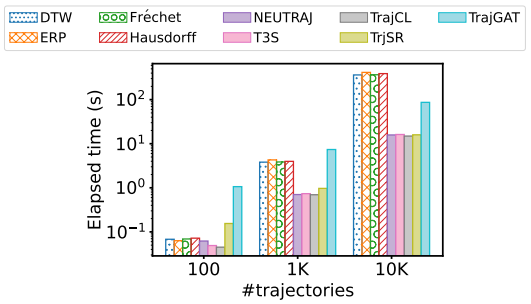


Fig. 16. Clustering time vs. the number of trajectories

5.5 Research Opportunities

The experimental results show that: (1) The learned measures are slower than the heuristic ones when computing trajectory similarity on the fly. (2) The learned measures are faster than the heuristic ones for kNN queries and clustering when the embeddings can be pre-computed, although

there is no accuracy guarantee. (3) Self-attention network-based measures are relatively fast to train to obtain a high accuracy of trajectory similarity.

These results motivate important research questions:

RQ1: Can learned measures also outperform the heuristic ones for online trajectory similarity computation, for wider applicability?

RQ2: Can learned measures approximate the heuristic ones with accuracy guarantees?

RQ3: Can a learned measure approximate a wide range of different heuristic measures all with a high accuracy?

RQ4: Can heuristic measures be indexed with a structure to achieve higher kNN query efficiency?

Here, we present an attempt to answer the first question. We make use a *feedforward neural network* (FFN) as the trajectory encoder, which is arguably the simplest and most efficient deep learning model. We use a two-layer FFN, where the size of the intermediate vectors is d and the activation function is *ReLU*. The inputs are raw two-dimensional coordinates of the points on a trajectory, for efficiency considerations.

Table 12. Comparison between the existing representative measures and an FFN-based measure

	Similarity comp.	**kNN query**		**Clustering**	
	Time (s)	**Time (s)**	**HR@50**	**Time (s)**	**RI**
Hausdorff	5.97	903.41	**100.0%**	3.98	**100.0%**
TrajCL	77.40	0.89	83.1%	0.70	81.1%
FFN	**6.20**	**0.41**	59.1%	**0.41**	50.2%

We follow the previous default experimental settings of batched computation on GPU and repeat the experiments with FFN to approximate Hausdorff. We compare the results with the most efficient heuristic and learned measures, i.e., Hausdorff and TrajCL, respectively. As Table 12 shows, FFN is much more efficient than TrajCL for trajectory similarity computation, and it even achieves a comparable time to that of Hausdorff. Such results are achieved because the time complexity of FFN for trajectory similarity computation is $O(nd)$, which is lower than that of TrajCL and similar to that of Hausdorff. However, this high efficiency comes with a substantial cost in the kNN query and clustering accuracy.

Such results show the potential of learned measures to obtain a high efficiency for trajectory similarity computation, thus meeting their original promise, while they also highlight the challenges in obtaining high query and clustering accuracy at the same time.

6 CONCLUSION

We revisited both heuristic and deep learning-based trajectory similarity measures and studied their empirical efficiency comprehensively. We found that the learned measures outperform the heuristic measures as promised in literature, only when the trajectory embeddings can be pre-computed. Meanwhile, such measures lack accuracy when approximating the heuristic measures. Among the learned measures, the self-attention-based ones are also the fastest to train and offer the highest accuracy. In comparison, the heuristic measures do not require pre-computations and are more suitable for one-off trajectory similarity computations. These results open up research opportunities in designing advanced learned measures with even higher efficiency and accuracy.

REFERENCES

[1] 2015. Porto Taxi Trajectory Dataset. https://www.kaggle.com/c/pkdd-15-predict-taxi-service-trajectory-i.
[2] 2018. DiDi GAIA Open Dataset. https://outreach.didichuxing.com/.
[3] 2020. Trajectory Distance Library. https://github.com/bguillouet/traj-dist.

[4] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. 1993. Efficient Similarity Search in Sequence Databases. In *International Conference on Foundations of Data Organization and Algorithms*. 69–84.

[5] Helmut Alt. 2009. The Computational Geometry of Comparing Shapes. In *Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*. 235–248.

[6] Helmut Alt and Michael Godau. 1995. Computing the Fréchet Distance between Two Polygonal Curves. *International Journal of Computational Geometry & Applications* 5, 01n02 (1995), 75–91.

[7] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*. 322–331.

[8] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517.

[9] Hanlin Cao, Haina Tang, Yulei Wu, Fei Wang, and Yongjun Xu. 2021. On Accurate Computation of Trajectory Similarity via Single Image Super-resolution. In *IJCNN*. 1–9.

[10] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2016. Better Bitmap Performance with Roaring Bitmaps. *Software: Practice and Experience* 46, 5 (2016), 709–719.

[11] Yanchuan Chang, Jianzhong Qi, Yuxuan Liang, and Egemen Tanin. 2023. Contrastive Trajectory Similarity Learning with Dual-Feature Attention. In *ICDE*. 2933–2945.

[12] Yanchuan Chang, Jianzhong Qi, Egemen Tanin, Xingjun Ma, and Hanan Samet. 2021. Sub-Trajectory Similarity Join with Obfuscation. In *SSDBM*. 181–192.

[13] Yanchuan Chang, Egemen Tanin, Xin Cao, and Jianzhong Qi. 2023. Spatial Structure-Aware Road Network Embedding via Graph Contrastive Learning. In *EDBT*. 144–156.

[14] Chao Chen, Chengwu Liao, Xuefeng Xie, Yasha Wang, and Junfeng Zhao. 2019. Trip2Vec: A Deep Embedding Approach for Clustering and Profiling Taxi Trip Purposes. *Personal and Ubiquitous Computing* 23 (2019), 53–66.

[15] Lu Chen, Yunjun Gao, Ziquan Fang, Xiaoye Miao, Christian S Jensen, and Chenjuan Guo. 2019. Real-time Distributed Co-movement Pattern Detection on Streaming Trajectories. *PVLDB* 12, 10 (2019), 1208–1220.

[16] Lei Chen and Raymond Ng. 2004. On the Marriage of LP-norms and Edit Distance. In *PVLDB*. 792–803.

[17] Lei Chen, M Tamer Özsu, and Vincent Oria. 2005. Robust and Fast Similarity Search for Moving Object Trajectories. In *SIGMOD*. 491–502.

[18] Yuanyi Chen, Peng Yu, Wenwang Chen, Zengwei Zheng, and Minyi Guo. 2021. Embedding-based Similarity Computation for Massive Vehicle Trajectory Data. *IEEE Internet of Things Journal* 9, 6 (2021), 4650–4660.

[19] Ziwen Chen, Ke Li, Silin Zhou, Lisi Chen, and Shuo Shang. 2023. Towards Robust Trajectory Similarity Computation: Representation-based Spatio-temporal Similarity Quantification. *World Wide Web* 26 (2023), 1271–1294.

[20] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. In *NIPS Workshop on Deep Learning*.

[21] Ticiana L Coelho Da Silva, Karine Zeitouni, and José AF de Macêdo. 2016. Online Clustering of Trajectory Data Stream. In *MDM*. 112–121.

[22] Liwei Deng, Yan Zhao, Zidan Fu, Hao Sun, Shuncheng Liu, and Kai Zheng. 2022. Efficient Trajectory Similarity Computation with Contrastive Learning. In *CIKM*. 365–374.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL*. 4171–4186.

[24] Amine Dhraief, Raik Issaoui, and Abdelfettah Belghith. 2011. Parallel Computing the Longest Common Subsequence (LCS) on GPUs: Efficiency and Language Suitability. In *International Conference on Advanced Communications and Computation*. 143–148.

[25] Anne Driemel, Sariel Har-Peled, and Carola Wenk. 2010. Approximating the Fréchet Distance for Realistic Curves in Near Linear Time. In *SoCG*. 365–374.

[26] Thomas Eiter and Heikki Mannila. 1994. *Computing Discrete Fréchet Distance*. Technical Report. Technical University of Vienna.

[27] Ziquan Fang, Yuntao Du, Lu Chen, Yujia Hu, Yunjun Gao, and Gang Chen. 2021. E2DTC: An End to End Deep Trajectory Clustering Framework via Self-training. In *ICDE*. 696–707.

[28] Ziquan Fang, Yuntao Du, Xinjun Zhu, Lu Chen, Yunjun Gao, and Christian S. Jensen. 2022. Spatio-temporal Trajectory Similarity Learning in Road Networks. In *KDD*. 347–356.

[29] Tao-Yang Fu and Wang-Chien Lee. 2020. Trembr: Exploring Road Networks for Trajectory Representation Learning. *ACM Transactions on Intelligent Systems and Technology* 11, 1 (2020), 10:1–25.

[30] Maxime Gariel, Ashok N Srivastava, and Eric Feron. 2011. Trajectory Clustering and an Application to Airspace Monitoring. *IEEE Transactions on Intelligent Transportation Systems* 12, 4 (2011), 1511–1524.

[31] Antonin Guttman. 1984. R-trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*. 47–57.

[32] Peng Han, Jin Wang, Di Yao, Shuo Shang, and Xiangliang Zhang. 2021. A Graph-based Approach for Trajectory Similarity Computation in Spatial Networks. In *KDD*. 556–564.

[33] Huajun He, Ruiyuan Li, Sijie Ruan, Tianfu He, Jie Bao, Tianrui Li, and Yu Zheng. 2022. TraSS: Efficient Trajectory Similarity Search Based on Key-Value Data Stores. In *ICDE*. 2306–2318.
[34] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780.
[35] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.
[36] Eamonn Keogh and Chotirat Ann Ratanamahatana. 2005. Exact Indexing of Dynamic Time Warping. *Knowledge and Information Systems* 7, 3 (2005), 358–386.
[37] Jae-Gil Lee, Jiawei Han, and Kyu-Young Whang. 2007. Trajectory Clustering: A Partition-and-group Framework. In *SIGMOD*. 593–604.
[38] Xiaohui Li, Vaida Ceikute, Christian S Jensen, and Kian-Lee Tan. 2012. Effective Online Group Discovery in Trajectory Databases. *IEEE Transactions on Knowledge and Data Engineering* 25, 12 (2012), 2752–2766.
[39] Xiucheng Li, Kaiqi Zhao, Gao Cong, Christian S. Jensen, and Wei Wei. 2018. Deep Representation Learning for Trajectory Similarity Computation. In *ICDE*. 617–628.
[40] Bin Lin and Jianwen Su. 2008. One Way Distance: For Shape based Similarity Search of Moving Object Trajectories. *GeoInformatica* 12 (2008), 117–142.
[41] An Liu, Yifan Zhang, Xiangliang Zhang, Guanfeng Liu, Yanan Zhang, Zhixu Li, Lei Zhao, Qing Li, and Xiaofang Zhou. 2022. Representation Learning with Multi-level Attention for Activity Trajectory Similarity Computation. *IEEE Transactions on Knowledge and Data Engineering* 34, 5 (2022), 2387–2400.
[42] Xiang Liu, Xiaoying Tan, Yuchun Guo, Yishuai Chen, and Zhe Zhang. 2022. CSTRM: Contrastive Self-Supervised Trajectory Representation Model for Trajectory Similarity Computation. *Computer Communications* 185 (2022), 159–167.
[43] Gonzalo Navarro. 2001. A Guided Tour to Approximate String Matching. *Comput. Surveys* 33, 1 (2001), 31–88.
[44] Jonathan F O'Connell and Christine L Mumford. 2014. An Exact Dynamic Programming Based Method to Solve Optimisation Problems Using GPUs. In *International Symposium on Computing and Networking*. 347–353.
[45] Xavier Olive, Luis Basora, Benoit Viry, and Richard Alligier. 2020. Deep Trajectory Clustering with Autoencoders. In *International Conference for Research in Air Transportation*.
[46] Hae-Sang Park and Chi-Hyuck Jun. 2009. A Simple and Fast Algorithm for K-medoids Clustering. *Expert Systems with Applications* 36, 2 (2009), 3336–3341.
[47] Sayan Ranu, Padmanabhan Deepak, Aditya D. Telang, Prasad Deshpande, and Sriram Raghavan. 2015. Indexing and Matching Trajectories under Inconsistent Sampling Rates. In *ICDE*. 999–1010.
[48] Hanan Samet. 1984. The Quadtree and Related Hierarchical Data Structures. *Comput. Surveys* 16, 2 (1984), 187–260.
[49] Zeyuan Shang, Guoliang Li, and Zhifeng Bao. 2018. DITA: Distributed In-memory Trajectory Analytics. In *SIGMOD*. 725–740.
[50] Sharanyan Srikanthan, Arvind Kumar, and Rajeev Gupta. 2011. Implementing the Dynamic Time Warping Algorithm in Multithreaded Environments for Real Time and Unsupervised Pattern Discovery. In *International Conference on Computer and Communication Technology*. 394–398.
[51] Han Su, Shuncheng Liu, Bolong Zheng, Xiaofang Zhou, and Kai Zheng. 2020. A Survey of Trajectory Distance Measures and Performance Evaluation. *The VLDB Journal* 29 (2020), 3–32.
[52] Yaguang Tao, Alan Both, Rodrigo I. Silveira, Kevin Buchin, Stef Sijben, Ross S. Purves, Patrick Laube, Dongliang Peng, Kevin Toohey, and Matt Duckham. 2021. A Comparative Analysis of Trajectory Similarity Measures. *GIScience & Remote Sensing* 58, 5 (2021), 643–669.
[53] David Alexander Tedjopurnomo, Xiucheng Li, Zhifeng Bao, Gao Cong, Farhana Choudhury, and A. Kai Qin. 2021. Similar Trajectory Search with Spatio-temporal Deep Representation Learning. *ACM Transactions on Intelligent Systems and Technology* 12, 6 (2021), 77:1–26.
[54] Christopher Tralie and Elizabeth Dempsey. 2020. Exact, Parallelizable Dynamic Time Warping Alignment with Linear Memory. *arXiv preprint arXiv:2008.02734* (2020).
[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *NIPS*. 6000–6010.
[56] Michail Vlachos, George Kollios, and Dimitrios Gunopulos. 2002. Discovering Similar Multidimensional Trajectories. In *ICDE*. 673–684.
[57] Sheng Wang, Zhifeng Bao, J. Shane Culpepper, and Gao Cong. 2021. A Survey on Trajectory Data Management, Analytics, and Learning. *Comput. Surveys* 54, 2 (2021), 39:1–36.
[58] Sheng Wang, Zhifeng Bao, J. Shane Culpepper, Zizhe Xie, Qizhi Liu, and Xiaolin Qin. 2018. Torch: A Search Engine for Trajectory Data. In *SIGIR*. 535–544.
[59] Tingting Wang, Shixun Huang, Zhifeng Bao, J. Shane Culpepper, and Reza Arablouei. 2022. Representative Routes Discovery from Massive Trajectories. In *KDD*. 4059–4069.

[60] Taizheng Wang, Chunyang Ye, Hui Zhou, Mingwang Ou, and Bo Cheng. 2021. AIS Ship Trajectory Clustering based on Convolutional Auto-encoder. In *Intelligent Systems and Applications*. 529–546.
[61] Zheng Wang, Cheng Long, and Gao Cong. 2023. Similar Sports Play Retrieval with Deep Reinforcement Learning. *IEEE Transactions on Knowledge and Data Engineering* 35, 4 (2023), 4253–4266.
[62] Zheng Wang, Cheng Long, Gao Cong, and Ce Ju. 2019. Effective and Efficient Sports Play Retrieval with Deep Representation Learning. In *KDD*. 499–509.
[63] Phillip G. D. Ward, Zhen He, Rui Zhang, and Jianzhong Qi. [n.d.]. ([n. d.]).
[64] Limin Xiao, Yao Zheng, Wenqi Tang, Guangchao Yao, and Li Ruan. 2013. Parallelizing Dynamic Time Warping Algorithm Using Prefix Computations on GPU. In *IEEE International Conference on High Performance Computing and Communications & IEEE International Conference on Embedded and Ubiquitous Computing*. 294–299.
[65] Shucai Xiao, Ashwin M Aji, and Wu-chun Feng. 2009. On the Robust Mapping of Dynamic Programming onto a Graphics Processing Unit. In *ICPADS*. 26–33.
[66] Dong Xie, Feifei Li, and Jeff M. Phillips. 2017. Distributed Trajectory Similarity Search. *PVLDB* 10, 11 (2017), 1478–1489.
[67] Peilun Yang, Hanchen Wang, Defu Lian, Ying Zhang, Lu Qin, and Wenjie Zhang. 2022. TMN: Trajectory Matching Networks for Predicting Similarity. In *ICDE*. 1700–1713.
[68] Peilun Yang, Hanchen Wang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2021. T3S: Effective Representation Learning for Trajectory Similarity Computation. In *ICDE*. 2183–2188.
[69] Di Yao, Gao Cong, Chao Zhang, and Jingping Bi. 2019. Computing Trajectory Similarity in Linear Time: A Generic Seed-guided Neural Netric learning approach. In *ICDE*. 1358–1369.
[70] Di Yao, Haonan Hu, Lun Du, Gao Cong, Shi Han, and Jingping Bi. 2022. TrajGAT: A Graph-based Long-term Dependency Modeling Approach for Trajectory Similarity Computation. In *KDD*. 2275–2285.
[71] Di Yao, Chao Zhang, Zhihua Zhu, Jianhui Huang, and Jingping Bi. 2017. Trajectory Clustering via Deep Representation Learning. In *IJCNN*. 3880–3887.
[72] Rex Ying, Jiangwei Pan, Kyle Fox, and Pankaj K. Agarwal. 2016. A Simple Efficient Approximation Algorithm for Dynamic Time Warping. In *SIGSPATIAL*. 21:1–10.
[73] Mingxuan Yue, Yaguang Li, Haoze Yang, Ritesh Ahuja, Yao-Yi Chiang, and Cyrus Shahabi. 2019. DETECT: Deep Trajectory Clustering for Mobility-behavior Analysis. In *IEEE International Conference on Big Data*. 988–997.
[74] Hanyuan Zhang, Xingyu Zhang, Qize Jiang, Baihua Zheng, Zhenbang Sun, Weiwei Sun, and Changhu Wang. 2020. Trajectory Similarity Learning with Auxiliary Supervision and Optimal Matching. In *IJCAI*. 11–17.
[75] Bolong Zheng, Lianggui Weng, Xi Zhao, Kai Zeng, Xiaofang Zhou, and Christian S Jensen. 2021. REPOSE: Distributed Top-k Trajectory Similarity Search with Local Reference Point Tries. In *ICDE*. 708–719.
[76] Yu Zheng, Xing Xie, and Wei-Ying Ma. 2010. Geolife: A Collaborative Social Networking Service among User, Location and Trajectory. *IEEE Data Engineering Bulletin* 33, 2 (2010), 32–39.
[77] Silin Zhou, Jing Li, Hao Wang, Shuo Shang, and Peng Han. 2023. GRLSTM: Trajectory Similarity Computation with Graph-based Residual LSTM. In *AAAI*. 4972–4980.