

Collectively Simplifying Trajectories in a Database: A Query Accuracy Driven Approach (Technical Report)

Zheng Wang¹, Cheng Long¹, Gao Cong¹, Qianru Zhang², Christian S. Jensen³

¹School of Computer Science and Engineering, Nanyang Technological University, Singapore

²The University of Hong Kong, Hong Kong SAR

³Department of Computer Science, Aalborg University, Denmark

zheng011@e.ntu.edu.sg, {c.long, gaocong}@ntu.edu.sg, qrzhang@cs.hku.hk, csj@cs.aau.dk

ABSTRACT

Increasing and massive volumes of trajectory data are being accumulated that may serve a variety of applications, such as mining popular routes or identifying ridesharing candidates. As storing and querying massive trajectory data is costly, trajectory simplification techniques have been introduced that intuitively aim to reduce the sizes of trajectories, thus reducing storage and speeding up querying, while preserving as much information as possible. Existing techniques rely mainly on hand-crafted error measures when deciding which point to drop when simplifying a trajectory. While the hope may be that such simplification affects the subsequent usability of the data only minimally, the usability of the simplified data remains largely unexplored. Instead of using error measures that indirectly may to some extent yield simplified trajectories with high usability, we adopt a direct approach to simplification and present the first study of query accuracy driven trajectory simplification, where the direct objective is to achieve a simplified trajectory database that preserves the query accuracy of the original database as much as possible. Specifically, we propose a multi-agent reinforcement learning based solution with two agents working cooperatively to collectively simplify trajectories in a database while optimizing query usability. Extensive experiments on three real-world trajectory datasets show that the solution is capable of consistently outperforming baseline solutions over various types of queries with different dynamics.

PVLDB Reference Format:

Zheng Wang¹, Cheng Long¹, Gao Cong¹, Qianru Zhang², Christian S. Jensen³. Collectively Simplifying Trajectories in a Database: A Query Accuracy Driven Approach (Technical Report). PVLDB, 14(1): XXX-XXX, 2023.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

1 INTRODUCTION

A trajectory is a sequence of time-stamped locations that describe the movement of an object over time. Massive amounts of trajectory data are being accumulated and used in diverse applications, such as discovering popular routes in a city [6], analyzing animal migration patterns [19], and performing sports analytics [39]. The accumulation of trajectory data [41] introduces at least two challenges [22, 43]: (1) storing the data is expensive, and (2) querying the data is time-consuming. These challenges can be addressed by conducting trajectory simplification, which aims to drop points from trajectories to save the storage cost and speed up query processing. The underlying rationale is that not all points in a trajectory carry equally important information, so that dropping unimportant ones may be acceptable. For example, if the location of an object is sampled regularly and the object does not move for a while then only the first and last positions during the period of inactivity are important, and those in-between may be dropped without losing information. Next, the efficiency of query processing is improved, at the expense of query results becoming approximate.

Indeed, the extent to which a collection of simplified trajectories enables accurate query results has been used widely as a measure of the quality of a simplification technique in empirical studies of trajectory simplification [3, 22, 43]. For example, Zhang et al. [43] evaluate existing simplification techniques in terms of their ability to produce simplified trajectories that affect the accuracy of range, k NN, and join queries as well as clustering minimally. While there are many proposals for trajectory simplification [10, 14, 27, 30–32, 38], they all assume a storage budget and aim to produce simplified trajectories that minimize the difference from the original trajectories according to a given difference notion. No proposals aim to optimize directly the query accuracy offered by the simplified trajectories. We call this line of study *Error-Driven Trajectory Simplification (EDTS)*.

In addition, existing simplification techniques are local in nature and operate on a per-trajectory basis as opposed to being global in nature and operating on a database of trajectories. Specifically, they aim to simplify a given trajectory T within a budget $r \cdot |T|$, where $r \in (0, 1]$ is the compression ratio. When these techniques are used to simplify a database of trajectories according to a compression ratio r , they simplify each trajectory in the database *separately* according to compression ratio r . This is likely sub-optimal in cases where trajectories have different sampling rates or different complexities. Intuitively, trajectories with higher sampling rates or lower complexity are candidates for simplification with larger compression ratios.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Therefore, we propose a new trajectory simplification problem, called *Query accuracy Driven Trajectory Simplification (QDTS)*. Given a trajectory database D and a storage budget, the problem is to find a simplified trajectory database D' that preserves the accuracy of query results as much as possible, compared to the query results on D . QDTS differs from the existing EDTS problem. First, it considers a different objective of trajectory simplification, namely that of preserving query accuracy directly, rather than through minimizing an error measure, as in the EDTS problem. Second, it is global in nature and takes a trajectory database as input and outputs a simplified trajectory database that satisfies a specified storage budget as a whole, instead of simplifying each trajectory in isolation according to a budget, as do existing EDTS techniques [10, 14, 41].

Challenges. An immediate solution to the QDTS problem is to reduce it (i.e., a *database-level* simplification problem) to a *trajectory-level* problem by applying simplification to each trajectory *separately* with a proportional budget. Specifically, it simplifies each trajectory T with the budget of $r \cdot |T|$. It is obvious that the resulting simplified database would contain M simplified trajectories and have at most $r \cdot \sum_{T \in D} |T| = r \cdot N$ points, where N denotes the total number points in the database. While this solution needs minimal design efforts, it suffers from two main issues. Issue 1: Uniform compression ratio: It applies the same compression ratio to each trajectory, which would be sub-optimal for trajectories with different sampling rates and/or different complexities. Issue 2: Query accuracy unawareness: Existing algorithms [10, 14, 27, 30–32, 38] (all of which operate at trajectory-level) aim to optimize some form of error metric that quantifies the difference between an original and a simplified trajectory. As a consequence, simply using any of these algorithms cannot help to optimize directly the query accuracy of the simplified database.

Another solution is to consider all trajectories in the database D *collectively* during the course of simplification. Consider a top-down approach, in which we start with the most simplified database, i.e., each simplified trajectory T' of an original trajectory T contains only the first and last points of T . We then iteratively introduce points from the original database according to some selection criterion until the budget is exhausted. Alternatively, we can adopt a bottom-up approach, in which we start from D and iteratively drop points until the remaining points are within the budget. This solution considers all trajectories *collectively* when simplifying trajectories, enabling different trajectories to be simplified with different compression ratios depending on their complexities. Therefore, it avoids the first issue mentioned above. However, this solution still does not contend with the second issue. Furthermore, the solution operates at the database level, whose scale is typically much larger than that of a single trajectory. For example, the Geolife dataset used in our experiment contains millions of points, while individual trajectories contain only around one thousand points. This brings up a third issue. Issue 3: Lack of scalability: Operating at the database level, the solution needs to repeatedly choose a point from among a very large set of points.

New Solution. Motivated by the above discussion, we propose a new solution called RL4QDTS for trajectory database simplification, which avoids all the three issues. RL4QDTS has two core ideas. First,

it considers all trajectories in the database *collectively* for simplification. Specifically, it starts with the most simplified database, then introduces original points into the simplified database iteratively until its budget is exhausted. For better efficiency, it uses an index that partitions the database into sub-spaces, called *spatio-temporal cubes*. Whenever it needs to choose a point to introduce into the database, it first chooses a cube based on the index and then chooses a point in the cube. To partition a database of trajectories, one immediate idea is to partition along the spatial and temporal dimensions with a predefined granularity, e.g., setting a grid size for the spatial dimensions and a time duration for the temporal dimension. Nevertheless, the granularity is hard to set appropriately and is unlikely to work across databases. Small cubes (corresponding to a fine granularity) contain few candidate points, making it difficult to find good points to introduce. Large cubes contain many points, making it costly to choose one point within a cube. Therefore, RL4QDTS builds an octree (a three-dimensional variant of the quadtree for spatio-temporal points) to partition the database into cubes. The octree provides different resolutions of data cubes organized in a tree structure, making it is possible to choose cubes with different sizes flexibly and adaptively by traversing the tree from the root node to an appropriate node.

Second, RL4QDTS leverages multi-agent reinforcement learning to choose a point iteratively such that the query accuracy based on the simplified database involving the chosen points is optimized. Specifically, it employs an agent (called Agent-Cube) to find an octree node with a cube of an appropriate size. Then, it employs another agent (called Agent-Point) to choose a point in the cube chosen by Agent-Cube and introduces the point into the simplified database. Specifically, the two agents employ Markov decision processes (MDP) [36] that are designed such that the two agents optimize cooperatively the query accuracy on the simplified database. The RL4QDTS algorithm then leverages the learned policies of the two agents for simplifying a trajectory database. In summary, the first idea enables a solution that avoids the first and third issues, and the second idea is to address the second issue.

Overall, we make the following contributions.

- We propose the QDTS problem that aims to find a simplified trajectory database within a given storage budget that preserves the query accuracy on the simplified database as much as possible. This is the first systematic study of this line of trajectory simplification.
- We develop multi-agent reinforcement learning based solution called RL4QDTS to the problem. It simplifies a database of trajectories collectively with the aim of optimizing the query accuracy on the simplified database, while leveraging an index on the trajectory data for better efficiency. We show that the objective of the QDTS problem is well aligned with that of RL4QDTS.
- We conduct experiments on three real-world trajectory datasets, which offer evidence that RL4QDTS is able to outperform existing EDTS solutions across *two* types of adaptations, *four* error measures, and *varying* storage budgets for *five* query operators. For example, it achieves the improvement of up to 35% for range query, 41% and 28% for two

kinds of k NN query, 35% for similarity query and 40% for clustering than the best baselines.

The rest of paper is organized as follows. We review the literature in Section 2. We cover preliminaries and define the QDTS problem in Section 3. We introduce the RL4QDTS solution in Section 4 and report on the experimental study in Section 5. We conclude the paper in Section 6.

2 RELATED WORK

Error-Driven Trajectory Simplification. The error-driven trajectory simplification aims to simplify a trajectory within a given storage budget and to minimize an error measure of the simplified trajectory. Many studies have been conducted on this problem, among which some focus on the batch mode (where full access to a trajectory is attained throughout the process) [10, 14, 27, 38] and others on the online mode (where a trajectory is inputted in an online fashion and those points that have been dropped are no longer accessible) [30–32]. We review the studies on the batch mode, which is the focus of this paper, as follows. Specifically, Top-Down [10] adapts the traditional Douglas-Peucker algorithm [7]. The algorithm starts with two points (the first and last) of a trajectory. Then, it repeatedly inserts a point with the largest error until the size of the simplified trajectory reaches the storage budget. Bottom-Up [14] adopts a reverse strategy. It scans all points of the input trajectory and repeatedly drops the point with the smallest error until the number of remaining points is within the storage budget. Long et al. [27] propose Span-Search, which is designed specifically to preserve direction information in trajectory simplification. Recently, Wang et al. [38] propose a reinforcement learning based method called RLTS+ for trajectory simplification. It adopts the Bottom-Up strategy and drops points based on a learned policy instead of using the heuristic rules seen in previous studies.

Overall, the above studies aim to minimize a given error measure while simplifying a trajectory. However, they largely disregard data usability as an objective of simplification algorithms. As a matter of fact, one of the main motivations for simplification is to improve query efficiency. Consequently, data usability should be treated as a key factor to indicate the quality of trajectory simplification. Indeed, data usability has been used to compare existing simplification algorithms in several empirical studies [3, 22, 43]. An early study [3] evaluates several error measures in trajectory simplification and analyzes the soundness of the measures for queries. Zhang et al. [43] consider four spatio-temporal queries (i.e., range query, k NN query, join query, and clustering) on a trajectory database and design the corresponding measures to evaluate the quality of existing trajectory simplification algorithms. A recent evaluation study [22] verifies the query qualities of error-bounded trajectory simplification algorithms [21, 23, 24, 28] that simplify a trajectory with a given error tolerance and aim to minimize the size of a simplified trajectory. However, data usability is only used as evaluation measures in these studies [3, 22, 43] to understand how well the existing simplification algorithms support various types of queries, but not considered in simplification algorithm design.

Other Types of Trajectory Simplification. Other studies of trajectory simplification include: (1) studies simplifying a trajectory such that the error of the simplified trajectory is bounded

and as many points as possible are dropped, considering batch mode [10, 14, 26, 31] and online mode [23, 24, 26, 28, 41], and (2) studies simplifying a trajectory by immediately deciding whether to keep or drop an incoming point (also called *dead reckoning*) [30–32]. These studies do not return trajectories with sizes bounded by user-specified parameters and thus cannot be used for our QDTS problem.

Road Network-based Trajectory Compression. Road network-based trajectory compression [9, 16, 17, 35] aims to compress trajectories that are generated by objects in road networks. Specifically, raw trajectories are initially map-matched to an underlying road network to obtain map-matched trajectories that consist of sequences of road segments. The map-matched trajectories are treated as strings, where each road segment is considered as a character. Then, string compression algorithms such as Huffman coding [11] can be utilized to compress the trajectories with or without information loss. In contrast, we aim to reduce trajectory data in its original form, i.e., as a sequence of time-stamped locations, without the input of a road network.

Reinforcement Learning. Reinforcement Learning (RL) aims to guide agents on how to take actions to maximize a cumulative reward in an environment, where the environment is usually modeled as a Markov decision process (MDP), involving states, actions, and rewards [36]. Recently, RL has been applied successfully to solve many algorithmic problems, such as similarity search [40], index learning [42], fleet management [20], and trajectory simplification [38, 41]. Our study differs from the existing studies of RL-based trajectory simplification in three aspects. 1) Our method simplifies trajectories collectively in a database, rather than simplifying each trajectory with an uniform compression ratio [38]. 2) Our method optimizes the query accuracy on a simplified database, rather than minimizing an error measure on a single trajectory [38] or minimizing the size of a simplified trajectory [41]. Both existing methods are query un-aware. 3) Our method builds an octree on the trajectory database and iteratively chooses a point by choosing a cube with a traversal on an octree and then choosing a point within the cube. It leverages two agents for the two decision making processes of choosing a cube and a point. These decision making processes are different from those in the existing methods [38, 41] and the corresponding designs (e.g., those of MDPs) are different.

3 PRELIMINARIES AND PROBLEM STATEMENT

3.1 Preliminaries

Trajectories and Segments. A trajectory T is a sequence of time-stamped points: $T = \langle p_1, p_2, \dots, p_n \rangle$, where n is the length of T (i.e., $n = |T|$). Each point p_i ($1 \leq i \leq n$) is a triple $p_i = (x_i, y_i, t_i)$, indicating that the moving object is at location (x_i, y_i) at time t_i . We define the line $\overline{p_i p_{i+1}}$ linking two neighboring points as a segment in the trajectory. Thus, the trajectory T corresponds to a sequence of segments $\overline{p_1 p_2}, \overline{p_2 p_3}, \dots, \overline{p_{n-1} p_n}$. A trajectory database D consists of a set of trajectories. We define N to be the total number of points in D (i.e., $N = |D|$).

Trajectory Simplification and Errors. Trajectory simplification aims to eliminate points from a trajectory T to obtain a simplified

4 METHODOLOGY

We propose a new algorithm called RL4QDTS for query accuracy driven trajectory simplification. It starts with the most simplified database, in which each simplified trajectory T' of an original trajectory T consists of only the first and last points of T . It then introduces original points into the simplified database iteratively until its budget is exhausted. For better efficiency, it builds an octree on the database of trajectories. Whenever it needs to choose a point, it first chooses a cube in the octree and then chooses a point in that cube. The octree recursively partitions a 2D spatial and 1D temporal space into 8 sub-spaces, which we call (spatial-temporal) cubes. Specifically, RL4QDTS employs an agent (called Agent-Cube) to traverse the octree to find a cube. Then, it employs another agent (called Agent-Point) to choose a point in the chosen cube to be inserted into the simplified database. The decision making processes by the two agents are modeled as Markov decision processes (MDP) [36] and the MDPs are designed so that the agents cooperatively optimize the query accuracy on the simplified database.

We present the details of the MDPs of Agent-Cube and Agent-Point in Sections 4.1 and 4.2, respectively. We then describe how the policies for the two MDPs are learned, in Section 4.3. We finally present the RL4QDTS algorithm that leverages the two agents for simplifying a trajectory database, in Section 4.4.

4.1 Agent-Cube: MDP for Choosing a Cube

Consider the task of choosing a cube. Agent-Cube chooses a cube by traversing the octree top-down, starting from the root node. Each time it visits a node, it decides whether to stop, meaning that the node's cube is chosen. If it does not stop, it decides which node among the 8 child nodes to visit. We define the Markov decision process (MDP) of Agent-Cube as follows.

(1) States. Let s^c denote a state of Agent-Cube's MDP, which we define as follows. We denote by B_i^j ($i > 1, 1 \leq j \leq 8$) a cube of the octree, which is at the i^{th} level and corresponds to the j^{th} child node of its parent node. We designate B_1^1 to denote the root node of the tree. Consider that Agent-Cube is currently visiting cube B_i^j . For cube B_i^j , we use the number of trajectories (denoted by $M_{B_i^j}$) and the number of queries (denoted by $Q_{B_i^j}$) that fall into it, to capture the distributions of the data and queries. As queries are not available beforehand, and thus we synthetically generate a workload of range queries, each query location is sampled randomly by following a certain distribution (e.g., data distribution). Formally, the state s^c at a cube B_i^j is defined by its 8 child nodes ($B_{i+1}^1, B_{i+1}^2, \dots, B_{i+1}^8$) with two distribution features (data and query) as follows.

$$s^c = \left\{ \frac{M_{B_{i+1}^1}}{M_{B_i^j}}, \frac{Q_{B_{i+1}^1}}{Q_{B_i^j}}, \dots, \frac{M_{B_{i+1}^8}}{M_{B_i^j}}, \frac{Q_{B_{i+1}^8}}{Q_{B_i^j}} \right\}, i \geq 1. \quad (4)$$

Here, the values of a state are normalized by dividing by the total numbers of trajectories and queries in cube B_i^j (i.e., $M_{B_i^j}$ and $Q_{B_i^j}$) to avoid data scale issues. We explain the intuition of the state design as follows. The data values in the states of cubes capture how trajectories are distributed over the cubes. For example, if a cube has only few trajectories and is sparse, an agent tends to select

this cube to ensure that data in that cube is not lost. Similarly, query values in the cubes capture how queries are distributed over the cubes. Intuitively, an agent tends to select a cube with a larger value since the data would serve more queries. We note that in cases we have some knowledge of the query workload for testing (e.g., its distribution), we can generate query workloads by following the distribution for training; in cases we have no knowledge of the query workload for testing, we can generate a query workload by following the data distribution for training. In our experiments, we conduct experiments which verify to some extent the transferability of our method for cases where the query workload for testing does not follow that of the one used for training.

(2) Actions. Let a^c denote an action of Agent-Cube's MDP. With the currently visited cube being B_i^j , we define two types of possible action: (1) Proceed to visit one of the 8 child nodes, and (2) Stop the traversal and return the current cube to Agent-Point, to choose a point within the chosen cube. Formally, a^c is defined as follows.

$$a^c = k \quad (1 \leq k \leq 9). \quad (5)$$

Here, $a^c = 1, 2, \dots, 8$ means to traverse one of the 8 child nodes of the current one and $a^c = 9$ means to stop at the current node. Furthermore, we constrain the action space by only considering the cubes that involve trajectories. Suppose we take an action $a^c = k$, which corresponds to one of the two transition cases. Case 1: it explores the next cube B_{i+1}^k if $1 \leq k \leq 8$, and a new state at cube B_{i+1}^k can be computed using Equation 4. Case 2: it stops and returns to Agent-Point if $k = 9$. More details of Agent-Point are presented in Section 4.2.

(3) Rewards. When the action is to explore one of the 8 child nodes, the reward cannot be immediately observed, since no point has been inserted into the simplified database. When the action is to choose the current cube for Agent-Point to choose a point within the cube, the simplified database would be updated and some reward signal can be acquired (e.g., by measuring the difference between the query accuracy on the original database and that on the updated simplified database). In summary, Agent-Cube would finally choose a cube for Agent-Point and then acquire a certain reward signal. Therefore, we make Agent-Cube and Agent-Point share the same rewards, since they cooperate towards the same objective, i.e., learning a query-aware policy such that a simplified database preserves the query accuracy as much as possible compared to the original database. In particular, we set the reward of an action by Agent-Cube to be equal to that of the following action of choosing a point within the selected cube. More details of the reward definition are presented in Section 4.2.

4.2 Agent-Point: MDP for Choosing a Point

We denote the chosen cube by Agent-Cube as B for simplicity. Next, we define the MDP of Agent-Point for choosing a point within B to introduce to the database.

(1) States. Let s^p denote a state of Agent-Point's MDP, which we define as follows. Let N_B (resp. M_B) denote the number of points (resp. trajectories) in the cube B . To define the state, one idea is to incorporate all N_B points. However, this idea has two issues. (1) The definition in this way is N_B -dependent, which is not suitable for other cases when the number of points is not N_B . (2) N_B is

generally very large. With this definition, the state space would be huge and the model is hard to train.

We design the states such that these two issues are avoided as follows. First, let $p_{s_a}^{T_i}$ and $p_{s_b}^{T_i}$ denote the first point and last point of a trajectory T_i within the cube, respectively. For each point $p_{s_j}^{T_i}$ in the cube with $s_a \leq s_j \leq s_b$, we define a pair of two values, denoted by $v(p_{s_j}^{T_i})$. The first value, denoted by $v_s(p_{s_j}^{T_i})$, is equal to the “spatial” distance between $p_{s_j}^{T_i}$ and the synchronous point on the segment linking the points immediately before and after $p_{s_j}^{T_i}$ in the trajectory T_i . The second value, denoted by $v_t(p_{s_j}^{T_i})$, is equal to the “temporal” difference between the time of $p_{s_j}^{T_i}$ and the time of $p_{s_j}^{T_i}$ ’s closest point on the segment linking the points immediately before and after $p_{s_j}^{T_i}$ in the trajectory T_i . In summary, we have:

$$v(p_{s_j}^{T_i}) = (v_s(p_{s_j}^{T_i}), v_t(p_{s_j}^{T_i})). \quad (6)$$

The intuition of the two values is to capture the features of the point $p_{s_j}^{T_i}$ from both the spatial and temporal aspects given the context of trajectory simplification.

Among all points in each trajectory T_i , we then find a point (denoted as $p_{s_*}^{T_i}$) which has the maximum v_s , where s_* denotes its index. That is,

$$s_* = \arg \max_{s_a \leq s_j \leq s_b} v_s(p_{s_j}^{T_i}). \quad (7)$$

Finally, the state s^P of Agent-Point is defined as the set of K largest v_s values of $v(p_{s_*}^{T_i})$ among the B_M trajectories, that is

$$s^P = \{v(p_{s_*}^{T_{\pi(1)}}), v(p_{s_*}^{T_{\pi(2)}}), \dots, v(p_{s_*}^{T_{\pi(K)}})\}, \quad (8)$$

where π denotes the permutation of T_1, T_2, \dots, T_{M_B} such that $v_s(p_{s_*}^{T_{\pi(1)}}), v_s(p_{s_*}^{T_{\pi(2)}}), \dots, v_s(p_{s_*}^{T_{\pi(M_B)}})$ is sorted descendingly. K ($K \leq M_B$) is a hyper-parameter that can be tuned empirically to control the size of the state space. Note that if a point has been introduced in the database, the point will not be used for the state definition.

Here, we refer to an example for illustrating the state definition. Consider a cube B_3^4 (the bottom right node at the third tree level) in Figure 2. It contains two points p_5 and p_8 for the definition. For p_5 (resp. p_8), we calculate the values as (1.6, 0.5) (resp. (1.3, 0.7)) for capturing the spatial and temporal distances with respect to its simplified segment $\overline{p_4 p_6}$ on T_2 (resp. $\overline{p_7 p_9}$ on T_3). Then, the state is constructed as $s^P = \{(1.6, 0.5), (1.3, 0.7)\}$ with the setting of $K = 2$.

Our state design avoids the two aforementioned issues, where K is generally much smaller than N_B or M_B . With this design, a state has a fixed size that is independent from the number of trajectories in the cube.

(2) Actions. Let a^P denote an action of Agent-Point. The design of actions is consistent with the design of state s^P . Specifically, the actions are defined as follows:

$$a^P = k \quad (1 \leq k \leq K), \quad (9)$$

where action $a^P = k$ means to introduce point $p_{s_*}^{T_{\pi(k)}}$ into D' .

(3) Rewards. A reward, which is associated with a transition, reflects the quality of an action taken at a given state. A larger reward indicates a better quality of the performed action. Since our objective is to obtain a simplified database that serves queries more

effectively (i.e., minimizing the difference between the query results on the original database and those on the simplified database), the reward design is expected to reflect the improvement of query performance as more points are included in the simplified database.

To this end, we use the query workloads that have been used for defining the states (e.g., we use a set of range queries, where each query location is randomly sampled by following the data distribution). One option is to perform the queries after each point is inserted to the simplified database, which is associated with the transition from the current state s^P to the next state $s^{P'}$ when an action a^P is taken. However, it would be prohibitively costly to perform queries for each inserted point. In addition, since the simplified database D' has not been fully constructed, the query improvement with inserting just one point is often negligible and it is hard to demonstrate the quality of the action.

In our design, we choose to perform the queries after Δ (e.g., $\Delta = 50$) points are inserted for achieving accumulative effects. Specifically, we denote the reward by R . At state s_i^P , we consider the simplified database (denoted by D'). At state $s_{i+\Delta}^P$, we consider the simplified database (denoted by D''). We then define the reward R as follows.

$$R = \text{diff}(Q(D), Q(D')) - \text{diff}(Q(D), Q(D'')), \quad (10)$$

where $\text{diff}(Q(D), Q(D'))$ measures the difference between the results of queries on the original database D and the simplified database D' . The intuition is that if the difference for the simplified database D'' is smaller, then the reward is larger. Furthermore, we make the reward R be shared by all transitions that are involved when traversing from s_i^P to $s_{i+\Delta}^P$ as well as those of Agent-Cube that are involved in this process.

With the above reward definition, the objective of the MDP, i.e., maximizing the accumulative rewards, would be equivalent to that of the QDTS problem, i.e., minimizing the difference between queries on the original database and those on the simplified database. To see this, suppose we traverse a sequence of N' states $s_1^P, s_2^P, \dots, s_{N'}^P$ (for simplicity, we assume $\delta = 1$ for this analysis). Correspondingly, we receive a sequence of rewards $R_1, R_2, \dots, R_{N'-1}$. We assume that the future rewards are accumulated without discounted rates, and thus the accumulative reward is calculated as follows.

$$\begin{aligned} \sum_{t=1}^{N'-1} R_t &= \sum_{t=1}^{N'-1} \text{diff}(Q(D), Q(D'_t)) - \text{diff}(Q(D), Q(D''_t)) \\ &= \text{diff}(Q(D), Q(D'_1)) - \text{diff}(Q(D), Q(D''_{N'-1})) \\ &= C - \text{diff}(Q(D), Q(D''_{N'-1})), \end{aligned} \quad (11)$$

where D'_t (resp. D''_t) denotes the simplified database at the state s_t^P before (resp. after) the action a_t^P is performed. We regard the initial term $\text{diff}(Q(D), Q(D'_1))$ as a constant C and no points have been inserted at that state. Therefore, the objective of the MDP is to maximize $C - \text{diff}(Q(D), Q(D''_{N'-1}))$ or equivalently to minimize $\text{diff}(Q(D), Q(D''_{N'-1}))$, which is exactly the objective of QDTS.

Algorithm 1: The framework of RL4QDTS algorithm

```
1 Function RL4QDTS( $D = \langle T_1, T_2, \dots, T_M \rangle, W$ ):
2   Build an octree  $OT$  for  $D$ ;
3   for  $i=1, 2, \dots, M$  do
4     Insert point  $p_1^{T_i}$  and  $p_{|T_i|}^{T_i}$  into  $D'$ ;
5   end
6   for  $i=2M+1, 2M+2, \dots, W$  do
7      $B \leftarrow \text{Agent-Cube}(OT)$ ;
8      $D' \leftarrow \text{Agent-Point}(B, D')$ ;
9   end
10 return  $D'$ 
```

Algorithm 2: The Agent-Cube

```
1 Function Agent-Cube( $OT$ ):
2    $h \leftarrow 1$  and  $l \leftarrow 1$ ;
3   while true do
4     Construct a state
        $s_h^c \leftarrow \{ \frac{M_{B_{h+1}^1}}{M_{B_h^1}}, \frac{Q_{B_{h+1}^1}}{Q_{B_h^1}}, \dots, \frac{M_{B_{h+1}^8}}{M_{B_h^8}}, \frac{Q_{B_{h+1}^8}}{Q_{B_h^8}} \}$ ;
5     Sample an action  $a_h^c \sim \pi_{\theta^c}(a|s_h^c)$ ;
6     if  $a_h^c = 9$  then
7        $B \leftarrow B_h^l$ ;
8       Break;
9     else
10       $h \leftarrow h + 1$  and  $l \leftarrow a_h^c$ ;
11      Continue;
12    end
13  end
14 return  $B$ 
```

4.3 Policy Learning via DQN

The core problem of a MDP is to find an optimal policy, which guides an agent to choose an action at a specific state, such that the accumulative reward is maximized. Considering that the states in our MDPs are continuous, we adopt the Deep-Q-Networks (DQN) [29] for learning a policy from the MDPs of Agent-Cube and Agent-Point. Specifically, we adopt the deep Q learning with replay memory [29] for learning the policy, denoted by $\pi_{\theta^c}(a|s^c)$ for Agent-Cube (resp. $\pi_{\theta^p}(a|s^p)$ for Agent-Point), which samples an action a at a given state s^c (resp. s^p) via DQN, whose parameters are denoted by θ^c (resp. θ^p). We note that other RL algorithms such as policy gradient can also be used for continuous state MDPs.

4.4 The RL4QDTS Algorithm

Algorithm 1 details the framework of RL4QDTS with the learned policies of Agent-Cube and Agent-Point for the QDTS problem. Specifically, RL4QDTS starts by building an octree for the original trajectory database D (line 2), and then inserts the first and the last points of each trajectory into a simplified trajectory database D' (lines 3 – 5). The remaining budget $W - 2M$ is utilized in lines 6 – 9. First, it calls Agent-Cube (to be presented in Algorithm 2) to choose a cube B , and then the cube is fed into Agent-Point (to be

Algorithm 3: The Agent-Point

```
1 Function Agent-Point( $B, D'$ ):
2   Compute  $v(p_{s_*^j}^{T_j})$  ( $1 \leq j \leq M_B$ ) for  $T_j \in B$ ;
3   Maintains a descending permutation  $\pi$  of the values
   with a max-priority queue;
4   Construct a state
        $s^p \leftarrow \{v(p_{s_*^{\pi(1)}}^{T_{\pi(1)}}), v(p_{s_*^{\pi(2)}}^{T_{\pi(2)}}), \dots, v(p_{s_*^{\pi(K)}}^{T_{\pi(K)}})\}$ ;
5   Sample an action  $a^p \sim \pi_{\theta^p}(a|s^p)$ ;
6   Insert the point  $p_{s_*^{\pi(k)}}^{T_{\pi(k)}}$  into  $D'$  where  $a^p = k$ 
       ( $1 \leq k \leq K$ );
7 return  $D'$ 
```

presented in Algorithm 3) for updating D' . The process continues until the budget is exhausted. The RL4QDTS algorithm returns D' , which contains W points (line 10).

Agent-Cube in Algorithm 2 first initializes the indexes h and l to indicate a cube B_h^l (line 2). To sample a cube, in lines 3 – 13, it constructs a state s_h^c using Equation 4 (line 4), and samples an action a_h^c with the learned policy $\pi_{\theta^c}(a|s_h^c)$, which takes s_h^c as input (line 5). If the action is $a_h^c = 9$, it breaks and returns the current cube denoted by B (lines 6 – 8); otherwise, it updates the indexes by $h \leftarrow h + 1$ and $l \leftarrow a_h^c$, and explores the next cube B_h^l (lines 9 – 12).

Agent-Point in Algorithm 3 takes a cube B as input and computes the value of $v(p_{s_*^j}^{T_j})$ for each trajectory T_j using Equations 6 and 7, where $1 \leq j \leq M_B$ (line 2). Then, it maintains a descending permutation π of the values with a max-priority queue (line 3). Next, it constructs a state s^p using Equation 8 (line 4), and samples an action with the learned policy $\pi_{\theta^p}(a|s^p)$, which takes s^p as input (line 5). Let $a^p = k$ ($1 \leq k \leq K$) denote the sampled action. It then takes the action by inserting the point $p_{s_*^{\pi(k)}}^{T_{\pi(k)}}$ into D' (line 6).

We illustrate the RL4QDTS Algorithm with the running example in Figure 2. Here, we use a quadtree (instead of an octree) by ignoring the temporal dimension of trajectory points for ease of demonstration. The input is a trajectory database $D = \langle T_1, T_2, T_3 \rangle$ with storage budget $W = 7$. Suppose the range query workload involves Q_1 and Q_2 , where we have $Q_1(D) = \langle T_2 \rangle$ and $Q_2(D) = \langle T_2, T_3 \rangle$ based on the original database D . We build a quadtree and record the number of trajectories (M_B) and queries (Q_B) in the tree nodes. (1) The algorithm first inserts the first and the last points of each trajectory into D' , meaning that the remaining budget is one point (i.e., $7 - 2 \times 3 = 1$). (2) Then, Agent-Cube starts at the root node B_1^1 and constructs its state by observing the four child nodes. It takes the action to explore node B_2^2 (Top Left in the figure). (3) Similarly, at B_2^2 , it takes the action to explore node B_3^4 (Bottom Right). (4) At B_3^4 , Agent-Cube receives the action of providing B_3^4 to Agent-Point. (5) Agent-Point constructs the state at cube B_3^4 and takes the action to insert point p_5 into D' . (6) Finally, the algorithm breaks from the loop and returns the simplified database D' since the budget is exhausted. We observe that D' outputs the same results (i.e., $Q_1(D') = \langle T_2 \rangle$ and $Q_2(D') = \langle T_2, T_3 \rangle$) as when querying D , which means that the query accuracy is preserved.

In addition, we develop two techniques to enhance the effectiveness and efficiency of RL4QDTS. First, we constrain the octree

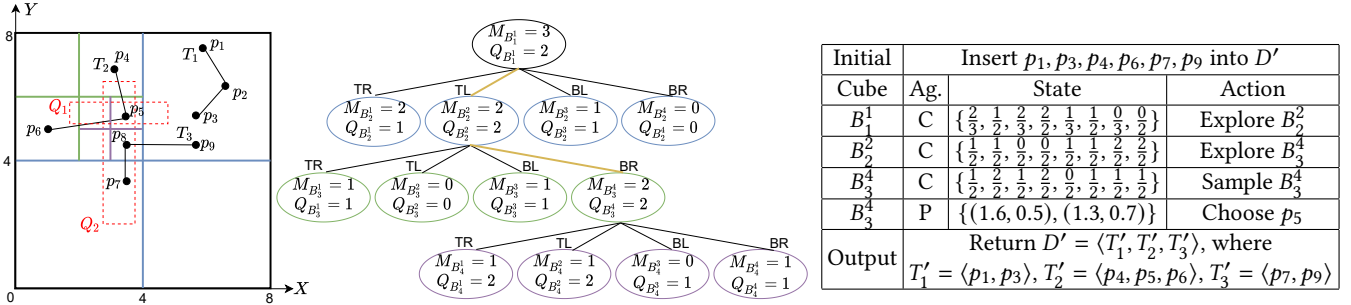


Figure 2: A running example of RL4QDTS, where we eliminate the temporal dimension and use a quadtree for ease of demonstration. Left: there are three trajectories T_1, T_2 and T_3 in the original database D , and two range queries Q_1 and Q_2 from query workload. Middle: TR, TL, BL and BR denote Top Right, Top Left, Bottom Left and Bottom Right nodes, respectively. We use different colors (black, blue and green) to mark different quadtree levels, and record the number of trajectories (M_B) and queries (Q_B) in the tree nodes. The Agent-Cube traverses from the root node, then to its TL node, then to its BR node marked as yellow lines. Right: the MDPs of Agent-Cube (C) and Agent-Point (P) for inserting the point p_5 into the simplified database D' .

Table 1: Dataset statistics.

Statistics	Geolife	T-Drive	Chengdu	OSM
# of trajectories	17,621	10,359	179,756	513,380
Total # of points	24,876,978	17,740,902	32,151,865	2,913,478,785
Ave. # of pts per traj	1,412	1,713	178	5,675
Sampling rate	1s ~ 5s	177s	2s ~ 4s	53.5s
Average length	9.96m	623m	25m	180m

traversal of Agent-Cube by a maximum tree depth E . If Agent-Cube reaches this level, it returns the currently visited cube to Agent-Point. The rationale is to prevent a very long traversal path for Agent-Cube, since in this case, it is difficult to train a policy to converge - recall that the reward is computed with delays. The benefits are verified in experiments. Second, we set a start level S so that the Agent-Cube starts traversing the octree by randomly sampling a cube following the query distribution (the one that has been used for defining states) from the start level S . The number of points in a cube decreases as the tree level increases. If Agent-Cube stops at the root level, Agent-Point will operate on all points in the database. Hyperparameter S can be used to avoid returning cells with excessive numbers of points.

Time complexity. The time complexity of the RL4QDTS algorithm is $O(N + W(n + \log M_B))$, where N , W , n , and M_B denote the total number of points in the original database, the storage budget, the maximum number of points in the input trajectories, and the maximum number of trajectories in the data cubes. Specifically, it takes $O(N)$ time to build an octree on the original database with maximum tree depth E , which is a small constant [34]. The part of processing of the remaining $W - 2M$ points dominates the complexity, including (1) choosing a cube by Agent-Cube with cost $O(1)$, which explores the octree for a bounded number of levels; (2) computing the values by Agent-Point with cost $O(n)$; (3) maintaining the min-priority queue with cost $O(\log M_B)$; (4) constructing a state, sampling an action, and inserting a point by Agent-Point with cost $O(1)$ assuming K is a small constant. We note that the RL4QDTS algorithm has the same complexity as the error-driven

algorithms [10, 14, 38] for simplifying a set of trajectories. In addition, we note that simplification is normally performed once offline, after which the simplified database is used for online querying.

5 EXPERIMENTS

5.1 Experimental Setup

Dataset. We conduct the experiments on four real-world trajectory datasets, Geolife¹, T-Drive², Chengdu³ and OSM⁴. Geolife contains trajectories from 182 users during a period of five years (2007 – 2012), and the trajectory data is distributed across 30 cities in China, with most trajectories being from Beijing. T-Drive contains trajectories from 10,357 taxis when driving in Beijing over a period of one week. Chengdu contains taxi trajectories from 2016-11-01 to 2016-11-07, released by DiDi Chuxing. OSM is used to test the scalability, which contains three billion points, released by the community on OpenStreetMap. The datasets are widely used in previous trajectory simplification studies [27, 38, 43], and detailed statistics are shown in Table 1.

Baselines. In the literature, no algorithms have been proposed for the QDTS problem. Given that the EDTS problem takes a storage budget for a trajectory as input, we consider existing algorithms that have been proposed for EDTS as baselines in our experiments. Specifically, we consider four algorithms, namely Top-Down [10], Bottom-Up [14], RLTS+ [38], and Span-Search [27]. Among them, Top-Down, Bottom-Up, and RLTS+ are general frameworks that can be applied with different error measures, while Span-Search works with DAD only. We adapt Top-Down, Bottom-Up, and RLTS+ in two ways. The first is to simplify each trajectory in the database one by one by calling one of the algorithms (this adaptation is denoted as “E”). The second is to consider the database as a whole and simplify the database by inserting or dropping points among all points in the database as it simplifies a trajectory (this adaptation is

¹<https://www.microsoft.com/en-us/research/publication/geolife-gps-trajectory-dataset-user-guide/>

²<https://www.microsoft.com/en-us/research/publication/t-drive-trajectory-data-sample/>

³https://drive.google.com/file/d/1f1FcYH_Vw8PnUUoH4V5L3BFJ9SrWM0bm/view?usp=sharing

⁴<https://star.cs.ucr.edu/?OSM/GPS#center=43.6,-56.1&zoom=2>

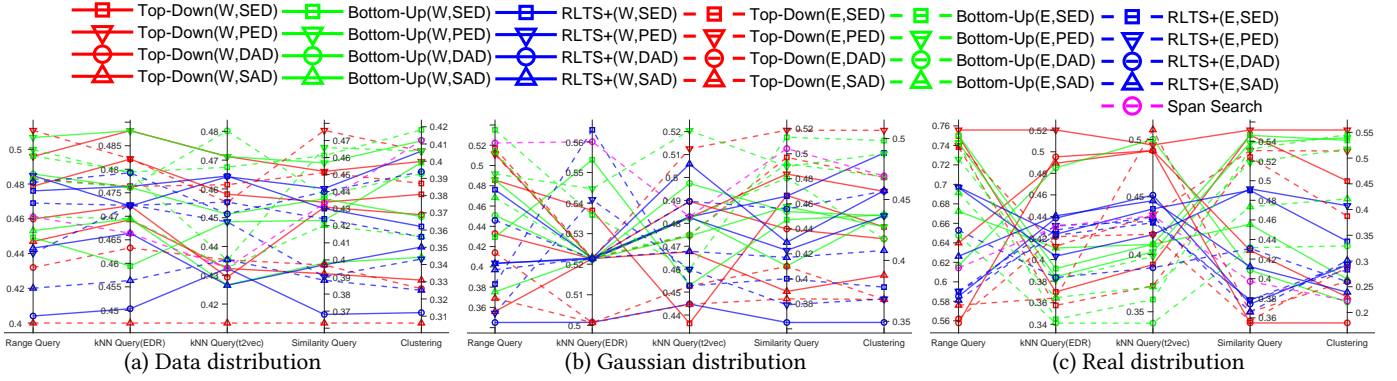


Figure 3: Skyline selection with existing algorithms.

denoted as “W”). In summary, for each of the algorithms Top-Down, Bottom-Up, and RLTS+, we obtain 8 ($= 4 \cdot 2$) adaptations as baselines, each corresponding to a combination of an error measure SED, PED, DAD, or SAD, and an adaptation method (“E” and “W”). In total, we have 25 baselines including 24 ($= 3 \cdot 8$) adaptations of Top-Down, Bottom-Up, and RLTS+ and 1 adaption of Span-Search (we note that for Span-Search, the “W” adaptation is not possible).

Evaluation Platform. We implement RL4QDTS and the baselines in Python 3.6 and Keras 2.2.0. The experiments are conducted on a 10-cores server with an Intel(R) Core(TM) i9-9820X CPU @3.30GHz 64.0GB RAM and an Nvidia GeForce RTX 2080 GPU. The datasets and code are available via the link⁵.

Model Training and Parameter Settings. We implement Agent-Cube with a two-layered feedforward neural network. The first layer has 25 neurons and uses the tanh activation function. The second layer has 9 neurons corresponding to the action space and uses a linear activation function. We set the hyperparameters S and E to be 9 and 12, respectively, based on empirical findings. We also implement Agent-Point with a two-layered feedforward neural network, where the first layer involves 25 neurons using the tanh activation function. The second layer involves K neurons corresponding to the action space and uses a linear activation function, where K is set to 2. We employ batch normalization in the neural networks to avoid data scale issues.

For training, we randomly sample 6,000 trajectories from Geolife (resp. 6,000 trajectories from T-Drive, 48,000 trajectories from Chengdu and 6,000 trajectories from OSM), and the remaining trajectories are used for testing. From the 6,000 (resp. 6,000, 48,000 and 6,000) trajectories, we randomly prepare 12 databases each with 500 (resp. 500, 4,000 and 500) trajectories. Further, we generate 5 episodes for each database for training the policy, and the best model is chosen during the training process. In addition, we set $\delta = 50$, i.e., for every 50 points that have been inserted, we perform 100 range queries, each with a spatial region of 2km by 2km and a temporal duration of 7 days, for constructing states and acquiring rewards. Here, we only use the range query for training since it is a basic query type involving both the spatial and temporal dimensions. We vary the distributions of range queries

across three distributions, namely (1) the data distribution, (2) the Gaussian distribution, and (3) the real distribution, for training the model. (1) and (2) are for the Geolife, T-Drive and OSM datasets and are adopted by following [33], where the parameters are set to be $\mu = 0.5$ and $\sigma = 0.25$ in the Gaussian distribution. (3) is for the Chengdu dataset, for which queries are generated near the pickup and dropoff locations that are provided in the dataset. The discount rate is set to 0.99. The RL4QDTS model is trained via Adam stochastic gradient descent with an initial learning rate of 0.01. The minimal ϵ is set to 0.1 with decay 0.99 for ϵ -greedy in DQN, and the size of the replay memory is set to 2000.

For testing, we notice the RL4QDTS involves some randomness of sampling a cube at the start level. Then, for each result of RL4QDTS, we run the algorithm 50 times and collect the averages and standard deviations of the query metrics. For range queries, we set the range query to be a cube with a spatial region of 2km by 2km and a temporal duration of 7 days. We use 7 days as the window length for both k NN and similarity queries. For k NN queries, we set $k = 3$ and use EDR and t2vec as the similarity measures. For EDR, we use a threshold of 2km and for t2vec, we adopt the settings as described in the original paper [18]. For similarity queries, the distance threshold is set to 5 km. For clustering, we adopt the TRACCLUS algorithm by following the original paper [15].

We emphasize that (1) we only produce *one* simplified database, which is used to support *multiple* types of queries, and (2) the models are trained based on range queries only and are used across all types queries.

5.2 Experimental Results

(1) Effectiveness evaluation (skyline selection of existing algorithms). Since we have 25 baselines, we select the skylines of the baselines for each query task to achieve more targeted comparisons. We construct a trajectory database D containing around 1.5 million points, and the storage budget for the simplification is set to $W = 0.25\% \cdot |D|$ for Geolife (resp. $W = 2\% \cdot |D|$ for Chengdu). In Figure 3, we show the effectiveness of the algorithms for five query tasks: range query, k NN Query(EDR), k NN Query(t2vec), Similarity Query, and clustering, with the three query distributions. For each task, we query 100 times and report the average results of the F_1 -score as described in Section 3.2. We conclude the selected baselines

⁵<https://www.dropbox.com/sh/ui2yegn2wmok8hs/AAAXRfeKH1R7AHATmMmsW3Cga?dl=0>

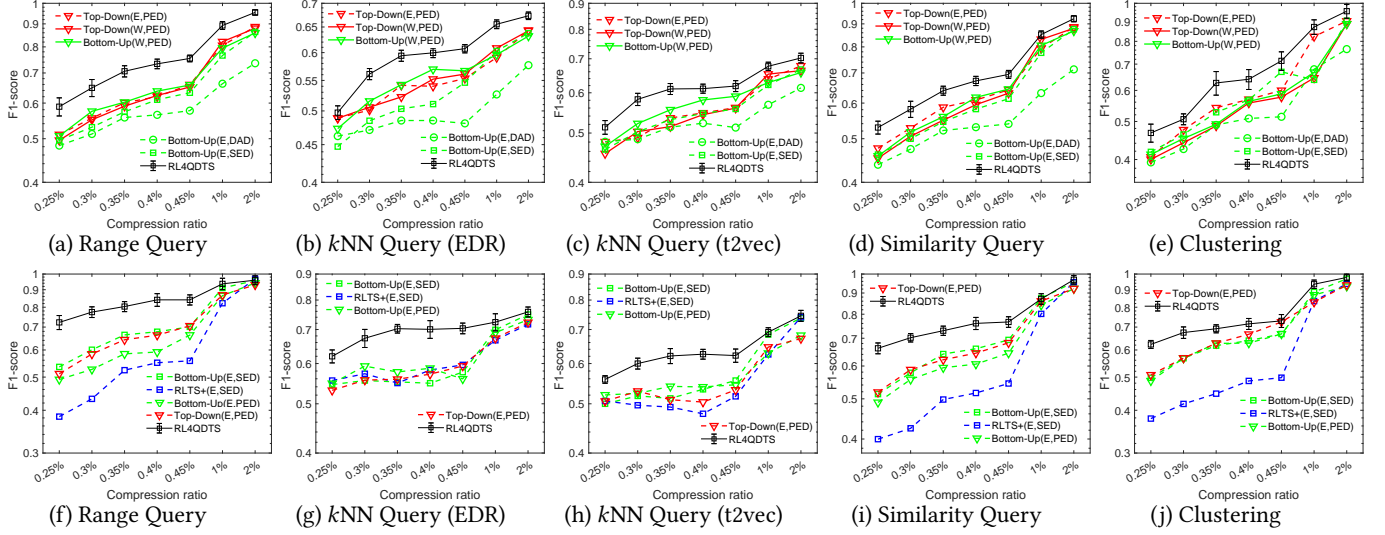


Figure 4: Comparison with skylines on Geolife (data distribution (a)-(e) and Gaussian distribution (f)-(j)).

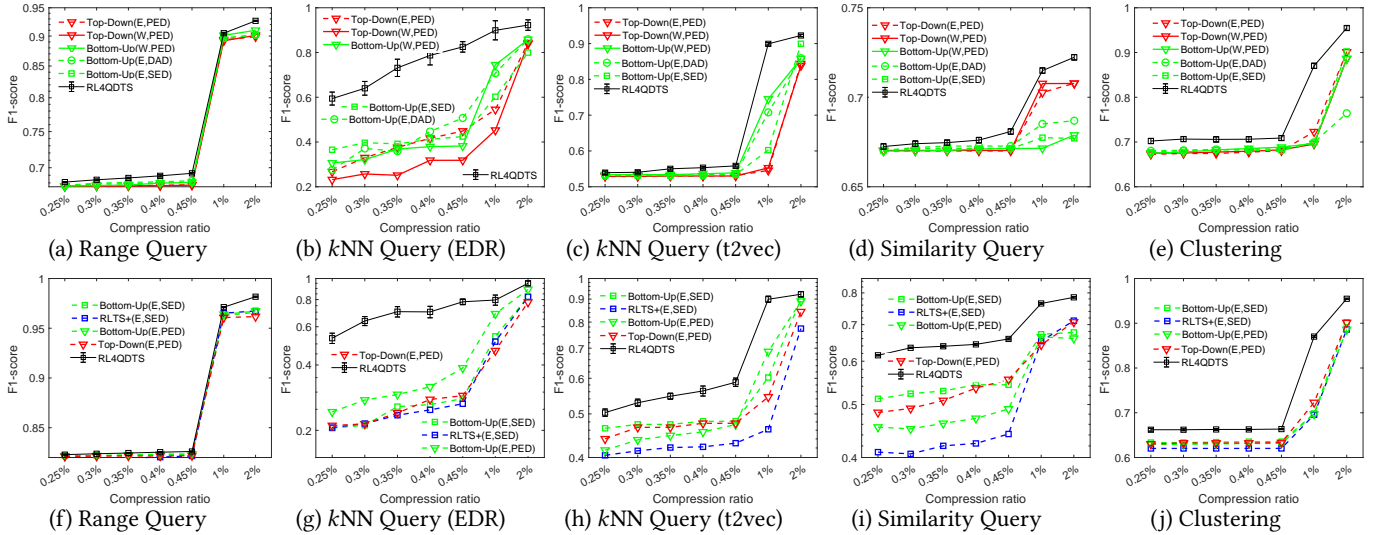


Figure 5: Comparison with skylines on T-Drive (data distribution (a)-(e) and Gaussian distribution (f)-(j)).

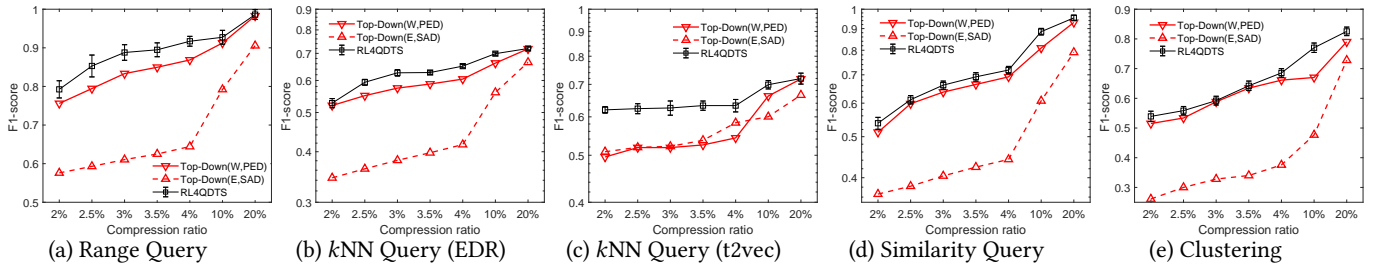


Figure 6: Comparison with skylines on Chengdu (real distribution (a)-(e)).

for comparisons as follows. For the data distribution, we observe that Top-Down(E,PED), Top-Down(W,PED), Bottom-Up(W,PED), Bottom-Up(E,DAD), and Bottom-Up(E,SED) are on the skyline.

For the Gaussian distribution, we observe that Bottom-Up(E,SED), RLTS+(E,SED), Bottom-Up(E,PED), and Top-Down(E,PED) are on

Table 2: Ablation study for RL4QDTS (Geolife).

Effectiveness	Range Query	Time (s)
RL4QDTS	0.733 ± 0.018	61.11
w/o Agent-Cube	0.673 ± 0.023	50.32
w/o Agent-Point	0.716 ± 0.021	59.31
w/o Agent-Cube and Agent-Point	0.641 ± 0.023	48.18

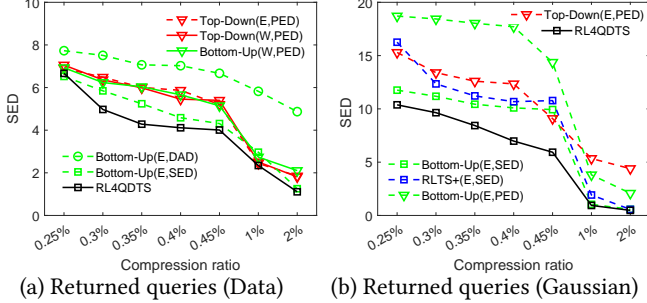


Figure 7: Deformation study

Table 3: Impacts of parameter S for RL4QDTS (Geolife), where RQ denotes the Range Query.

Start	7	8	9	10	11
RQ	0.557 ± 0.018	0.608 ± 0.019	0.733 ± 0.018	0.712 ± 0.021	0.686 ± 0.017
Time(s)	74.37	63.28	61.11	57.76	51.83

Table 4: Impacts of parameter E for RL4QDTS (Geolife).

End	10	11	12	13	14
RQ	0.673 ± 0.023	0.681 ± 0.018	0.733 ± 0.018	0.693 ± 0.019	0.671 ± 0.021
Time(s)	50.32	57.76	61.11	62.31	67.42

the skyline. For the real distribution, we observe that Top-Down(W,PED) and Top-Down(E,SAD) are on the skyline.

(2) Effectiveness evaluation (comparison with skyline). We compare RL4QDTS with the selected skyline methods for each query task. We vary the storage budget W from $0.25\% \cdot |D|$ to $2\% \cdot |D|$ for Geolife and T-Drive, and $2\% \cdot |D|$ to $20\% \cdot |D|$ for Chengdu. Figure 4 shows the results on the two query distributions (i.e., data and Gaussian) on Geolife. For RL4QDTS, we show its error bars obtained by running the algorithm 50 times as described in Section 5.1. The results based on T-Drive and Chengdu, shown in Figure 5 and Figure 6, respectively, demonstrate trends similar to those seen on Geolife. Overall, we observe that RL4QDTS consistently outperforms the existing error-driven methods across the different storage budgets, different query tasks with different generation distributions, and real datasets. Consider the results on Geolife for example. RL4QDTS outperforms the best skyline(s) by 34.6% (resp. 10.9%, 15.7%, 34.8%, and 39.9%) for range query (resp. k NN Query(EDR), k NN Query(t2vec), Similarity Query, and clustering) for the data distribution, and by 34.9% (resp. 11.6%, 14.5%, 27.6% and 22.8%) for range query (resp. k NN Query(EDR), k NN Query(t2vec), Similarity Query and clustering) for the Gaussian distribution. This is because RL4QDTS takes the query quality as the objective for trajectory simplification, and learns a query accuracy aware policy for the simplification to preserve the query quality directly; while existing methods aim to minimize a given error measure, and query quality is not considered directly.

Table 5: Impacts of parameter K for RL4QDTS (Geolife).

K	1	2	3	4	5
RQ	0.716 ± 0.021	0.733 ± 0.018	0.724 ± 0.021	0.723 ± 0.023	0.706 ± 0.023
Time(s)	59.31	61.11	62.43	63.85	64.35

Table 6: Impacts of k in k NN query for RL4QDTS (Geolife).

k NN	1	2	3	4	5
EDR	0.281 ± 0.013	0.472 ± 0.011	0.594 ± 0.013	0.641 ± 0.015	0.669 ± 0.015
t2vec	0.372 ± 0.016	0.525 ± 0.016	0.611 ± 0.015	0.661 ± 0.013	0.686 ± 0.015

(3) Effectiveness evaluation (ablation study). We conduct an ablation study to investigate the effects of Agent-Cube and Agent-Point in RL4QDTS. (1) We drop Agent-Cube by setting the start level $S = 9$ and the end level $E = 10$, so that Agent-Cube reduces to randomly sampling a cube according to the data distribution and then returning the cube to Agent-Point. (2) We drop Agent-Point and instead insert the point with the maximum value into a simplified database. (3) We drop both Agent-Cube and Agent-Point with the strategies described above. Table 2 reports the average results of 100 range queries with a distribution that follows the data distribution on a randomly sampled trajectory database with around 1.5 million points from Geolife. Overall, all components contribute to the result. Specifically, we observe that Agent-Cube improves the effectiveness by 6.1% and Agent-Point improves the effectiveness by 2.4%. As expected, the two agents cooperate to optimize query quality. In addition, we notice that without Agent-Cube, Agent-Point, or both, the efficiency improves since the agents employ learning models (i.e., deep neural networks) to make decisions. Without them, simple operations are used for the same tasks.

(4) Effectiveness evaluation (deformation study). We investigate the deformation of the trajectories in the returned queries. In Figure 7, we run 100 range queries with the data and Gaussian distributions, and report the average SED of the returned trajectories, which measures the deformation in terms of the synchronized distances between the original trajectories and their simplified ones. As expected, the deformation of RL4QDTS is consistently lower than skyline methods. This is because RL4QDTS is a query-aware solution, which preserves more points for those trajectories to answer the queries. For the skyline methods, they fail to preserve the trajectories in the returned queries, though the distances (e.g., SED) can be optimized explicitly for all trajectories including both the returned ones and others.

(5) Parameter study (varying parameter S in Agent-Cube). We evaluate the effect of the start level S in Agent-Cube. We report the average F_1 -score of 100 range queries that follows the data distribution on a database with around 1.5 million points, and we report the running time. Here, we fix the end level $E = 12$ and vary the start level S from 7 to 11. In Table 3, we observe that the efficiency improves as S increases. This is because the efficiency of RL4QDTS depends on the number of points in a sampled cube. A larger start level S corresponds to a smaller cube containing fewer candidate points, making the model run more efficiently to select a point. In contrast, a larger cube (corresponding to a smaller start level S) contains more candidates, e.g., in the extreme case of setting $S = 1$, the Agent-Cube may select a point from the whole database, resulting in scalability issue. In addition, we observe that a moderate

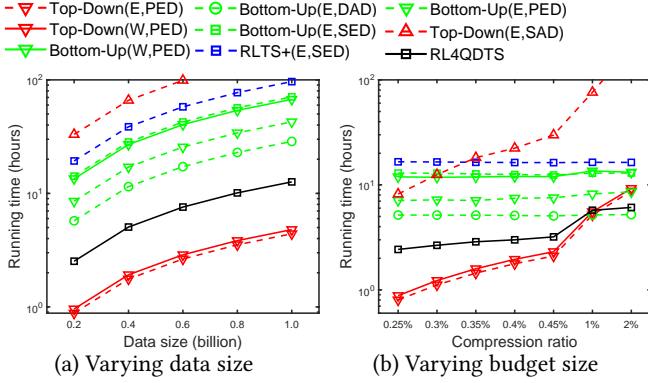


Figure 8: Efficiency evaluation (OSM)

setting (i.e., $S = 9$) brings the best effectiveness. The reason is that a smaller cube would make the model miss potential points to be introduced. Further, while a larger cube would contain many candidates of points, the sampled cube may not capture the query distribution accurately, e.g., in the extreme case that a cube covers the whole database, it does not reflect how queries are distributed at a finer granularity.

(6) Parameter study (varying parameter E in Agent-Cube).

We study the effect of end level E and fix the start level $S = 9$. We consider the effectiveness by running 100 range queries with the data distribution on a database with around 1.5 million points. We report the average F_1 -score and the running time in Table 4. We observe that the effectiveness improves as the end level E increases and then degrades as E increases further. This is because parameter E controls the maximum depth of the octree traversal. When E is small, Agent-Cube has a very limited search space, and the selected cubes may not be of good quality. When E is large, it is more difficult to train the RL policy to converge. In addition, we notice that the model runs fast with a smaller E , since it makes the model stops earlier. We choose $E = 12$ as it brings the best usability and runs comparably fast.

(7) Parameter study (varying parameter K in Agent-Point).

We study the effect of parameter K , which controls the state space of Agent-Point for decision-making. We construct a database with around 1.5 million points randomly sampled from Geolife, and report the average results of 100 range queries with the data distribution and the corresponding running time. In Table 5, we observe that the setting of $K = 2$ gives the best effectiveness, and runs reasonably fast. When $K = 1$, it reduces to greedily choose the point with the maximum value within a cube. When K becomes larger, the model performance degrades, since it becomes more difficult to train the model to choose a point among a larger number of candidate points. In addition, the time cost increases as K increases because more time is spent on constructing states. We set K to 2 by default, since this provides a reasonable trade-off between effectiveness and efficiency.

(8) Parameter study (varying parameter k in k NN query).

We study the effect of different k in k NN queries with EDR and t2vec on Geolife. In Table 6, we vary the k from 1 to 5, and report the effectiveness by running 100 k NN queries with the data distribution. We observe that the effectiveness improves as k increases, because it overlaps more similar trajectories with a larger k in the returned

Table 7: Training cost (Geolife).

Size	3,000	4,000	5,000	6,000	7,000
RQ	0.703 ± 0.017	0.714 ± 0.015	0.724 ± 0.021	0.733 ± 0.018	0.733 ± 0.018
Time(h)	0.67	0.88	1.15	1.36	1.55

Table 8: Impacts of Δ for training (Geolife).

Δ	20	30	40	50	60
RQ	0.681 ± 0.017	0.686 ± 0.016	0.701 ± 0.015	0.733 ± 0.018	0.685 ± 0.018
Time(h)	2.35	1.91	1.44	1.36	1.34

queries. In addition, the results on EDR and t2vec show similar trends.

(9) Scalability test (varying the data size N). We study scalability when varying the database size on OSM. We compare all skyline methods as shown in Figure 3, and vary the trajectory database size N from 0.2 billion to 1 billion points, with a fixed storage budget $W = 0.25\% \cdot N$. The running times (the maximum is set to 100 hours) are shown in Figure 8(a). Overall, we observe that RL4QDTS runs faster than many of the existing methods. It is only slower than the methods adapted from Top-Down because these methods are based on the idea of iteratively introducing points from the original single trajectory or whole database according to some simple criterion that can be computed efficiently. In contrast, RL4QDTS employs a learned policy for that task to improve effectiveness, and this incurs time costs for constructing states and sampling actions of the two agents. In addition, we notice that the algorithms adapted with the “W” option are generally slower than the algorithms adapted with the “E” option. For example, Bottom-Up(E,PED) is faster than Bottom-Up(W,PED) by around 40%, since the Bottom-Up(W,PED) operates on all points in the database, which generally is more expensive.

(10) Efficiency evaluation (varying the budget size W). We further study the effect of budget size W from $0.25\% \cdot N$ to $2\% \cdot N$, with a fixed N of 0.1 billion points. Figure 8(b) illustrates the running time on Geolife. RL4QDTS is slower than the Top-Down adaptations, but is faster than the Bottom-Up adaptations by at least a factor of two times. As W increases, RL4QDTS becomes faster than Top-Down adaptations, because it computes the values based on a partial trajectory within a cube by Agent-Point; however, Top-Down adaptations computes that values based on a whole trajectory.

(11) Training time. In Table 7, we show the training time of RL4QDTS on Geolife with the default settings described in Section 5.1. We randomly construct 5 training sets with 3,000 to 7,000 trajectories. For each training set, we report the time cost (hours) and effectiveness based on the selected best model during the training. We observe that the training generally takes several minutes and that the cost increases almost linearly with the number of trajectories. The effectiveness improves slightly when the number of trajectories exceeds 6,000. We use the setting of 6,000 trajectories to train the RL4QDTS model, which is enough to obtain a good model with a reasonable training cost. We further study the impacts of Δ in Table 8. We observe that the training time decreases as Δ increases, because it performs fewer queries for obtaining rewards. Besides, a moderate Δ is with the best effectiveness. This is because with a smaller Δ , it is hard to demonstrate the quality of actions, and with a larger Δ , it causes a long delayed reward during the training.

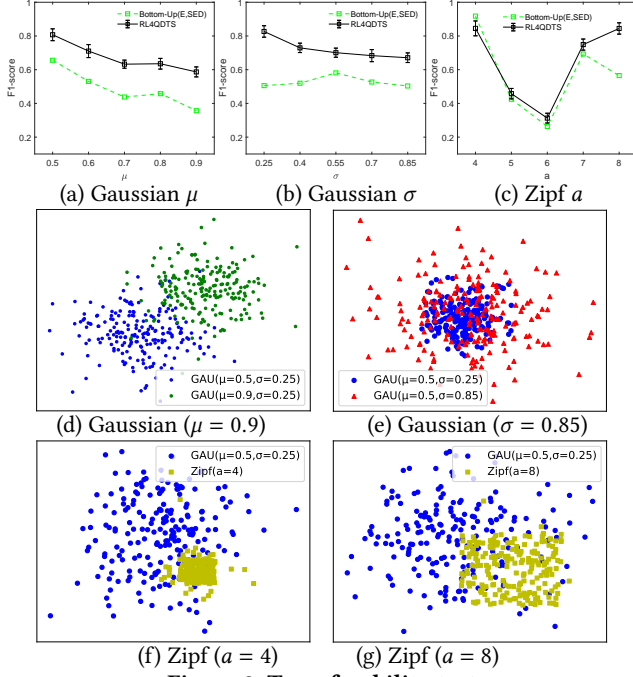


Figure 9: Transferability test

(12) Transferability test (with distribution changes). We consider two scenarios to test RL4QDTS transferability. First, we train RL4QDTS based on a query workload following the Gaussian distribution with $\mu = 0.5$ and $\sigma = 0.25$ on Geolife, and test its effectiveness for range queries which follow the Gaussian distributions with different μ (from 0.5 to 0.9), and σ (from 0.25 to 0.85). The results are shown in Figure 9(a) and (b). This is for testing the transferability for moderate changes in query distribution. Second, we use the same model that has been trained based on the Gaussian distribution and test its effectiveness for range queries which follow a Zipf distribution with different exponent parameters a (from 4 to 8). The a controls the density of the queries. We empirically set a starting from 4, since it will produce many duplicated queries that are closely together when a is further small. The results are shown in Figure 9(c). This is for testing the transferability for significant changes in query distribution. We also report the results of the best baseline Bottom-Up(E,SED). In Figure 9(a) and (b), we observe that as the change becomes more significant, RL4QDTS has its performance degrades slightly yet it still consistently outperforms the baseline across all settings of μ and σ . In Figure 9(c), we observe that RL4QDTS performs comparably well with the baseline in most cases and better sometimes though the query workload distribution has changed drastically, which shows the robustness of our method. This may be explained by that RL4QDTS does not rely on any error measures and preserves patterns and knowledge embedded in the data via neural networks, which remain useful to optimize queries even if the distributions are changed. We further visualize some distributions in Figure 9(d)-(g), which show changes in the query workload distribution to different extents.

6 CONCLUSION

We propose the query accuracy driven trajectory simplification problem, which aims to find a simplified trajectory database, such that the difference between the query results on the original database and those on the simplified database is minimized. We develop the first solution to the problem, called RL4QDTS, which is based on multi-agent reinforcement learning. Compared with existing error-driven trajectory simplification algorithms, RL4QDTS is able to collectively simplify trajectories across a database while optimizing the objective of the QDTS problem directly. Extensive experiments on three real-world trajectory datasets show that RL4QDTS is able to consistently outperform the existing EDTs algorithms for five query processing operations. One interesting research direction is to compress road networks-based trajectories with data-driven solutions to achieve better effectiveness and efficiency.

REFERENCES

- [1] Richard Bellman. 1961. On the approximation of curves by line segments using dynamic programming. *Commun. ACM* 4, 6 (1961), 284.
- [2] Hu Cao, Ouri Wolfson, and Goce Trajcevski. 2003. Spatio-temporal data reduction with deterministic error bounds. In *Proceedings of the 2003 joint workshop on Foundations of mobile computing*. 33–42.
- [3] Hu Cao, Ouri Wolfson, and Goce Trajcevski. 2006. Spatio-temporal data reduction with deterministic error bounds. *The VLDB Journal* 15, 3 (2006), 211–228.
- [4] Lei Chen, M Tamer Özsu, and Vincent Oria. 2005. Robust and fast similarity search for moving object trajectories. In *SIGMOD*. 491–502.
- [5] Yun Chen and Jignesh M Patel. 2009. Design and evaluation of trajectory join algorithms. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 266–275.
- [6] Zaiben Chen, Heng Tao Shen, and Xiaofang Zhou. 2011. Discovering popular routes from trajectories. In *ICDE*. 900–911.
- [7] David H Douglas and Thomas K Peucker. 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica* 10, 2 (1973), 112–122.
- [8] Elias Frentzos, Kostas Gratsias, Nikos Pelekis, and Yannis Theodoridis. 2005. Nearest neighbor search on moving object trajectories. In *International Symposium on Spatial and Temporal Databases*. Springer, 328–345.
- [9] Yunheng Han, Weiwei Sun, and Baihua Zheng. 2017. COMPRESS: A comprehensive framework of trajectory compression in road networks. *TODS* 42, 2 (2017), 1–49.
- [10] John Edward Hersherberger and Jack Snoeyink. 1992. *Speeding up the Douglas-Peucker line-simplification algorithm*. University of British Columbia, Department of Computer Science.
- [11] David A Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.
- [12] Bingqing Ke, Jie Shao, and Dongxiang Zhang. 2017. An efficient online approach for direction-preserving trajectory simplification with interval bounds. In *MDM*. 50–55.
- [13] Bingqing Ke, Jie Shao, Yi Zhang, Dongxiang Zhang, and Yang Yang. 2016. An online approach for direction-based trajectory compression with error bound guarantee. In *APWeb*. 79–91.
- [14] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. 2001. An online algorithm for segmenting time series. In *ICDM*. 289–296.
- [15] Jae-Gil Lee, Jiawei Han, and Kyu-Young Whang. 2007. Trajectory clustering: a partition-and-group framework. In *SIGMOD*. 593–604.
- [16] Tianyi Li, Lu Chen, Christian S Jensen, and Torben Bach Pedersen. 2021. TRACE: real-time compression of streaming trajectories in road networks. *Proceedings of the VLDB Endowment* 14, 7 (2021), 1175–1187.
- [17] Tianyi Li, Ruikai Huang, Lu Chen, Christian S Jensen, and Torben Bach Pedersen. 2020. Compression of uncertain trajectories in road networks. *PVLDB* 13, 7 (2020), 1050–1063.
- [18] Xiucheng Li, Kaiqi Zhao, Gao Cong, Christian S Jensen, and Wei Wei. 2018. Deep representation learning for trajectory similarity computation. In *ICDE*. 617–628.
- [19] Zhenhui Li, Jiawei Han, Ming Ji, Lu-An Tang, Yintao Yu, Bolin Ding, Jae-Gil Lee, and Roland Kays. 2011. Movemine: Mining moving object data for discovery of animal movement patterns. *TIST* 2, 4 (2011), 1–32.
- [20] Kaixiang Lin, Renyu Zhao, Zhe Xu, and Jiayu Zhou. 2018. Efficient large-scale fleet management via multi-agent deep reinforcement learning. In *SIGKDD*. 1774–1783.
- [21] Xuellian Lin, Jiahao Jiang, Shuai Ma, Yimeng Zuo, and Chunming Hu. 2019. One-pass trajectory simplification using the synchronous Euclidean distance. *VLDBJ*

- 28, 6 (2019), 897–921.
- [22] Xuelian Lin, Shuai Ma, Jiahao Jiang, Yanchen Hou, and Tianyu Wo. 2021. Error Bounded Line Simplification Algorithms for Trajectory Compression: An Experimental Evaluation. *TODS* 46, 3 (2021), 1–44.
 - [23] Xuelian Lin, Shuai Ma, Han Zhang, Tianyu Wo, and Jinpeng Huai. 2017. One-pass error bounded trajectory simplification. *PVLDB* 10, 7 (2017), 841–852.
 - [24] Jiajun Liu, Kun Zhao, Philipp Sommer, Shuo Shang, Brano Kusy, and Raja Jurdak. 2015. Bounded quadrant system: Error-bounded trajectory compression on the go. In *ICDE*. 987–998.
 - [25] Jiajun Liu, Kun Zhao, Philipp Sommer, Shuo Shang, Brano Kusy, Jae-Gil Lee, and Raja Jurdak. 2016. A novel framework for online amnesic trajectory compression in resource-constrained environments. *TKDE* 28, 11 (2016), 2827–2841.
 - [26] Cheng Long, Raymond Chi-Wing Wong, and HV Jagadish. 2013. Direction-preserving trajectory simplification. *PVLDB* 6, 10 (2013), 949–960.
 - [27] Cheng Long, Raymond Chi-Wing Wong, and HV Jagadish. 2014. Trajectory simplification: on minimizing the direction-based error. *PVLDB* 8, 1 (2014), 49–60.
 - [28] Nirvana Meratnia and A Rolf. 2004. Spatiotemporal compression techniques for moving point objects. In *EDBT*. 765–782.
 - [29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).
 - [30] Jonathan Muckell, Jeong-Hyon Hwang, Vikram Patil, Catherine T Lawson, Fan Ping, and SS Ravi. 2011. SQUISH: an online approach for GPS trajectory compression. In *Computing for Geospatial Research & Applications*. 1–8.
 - [31] Jonathan Muckell, Paul W Olsen, Jeong-Hyon Hwang, Catherine T Lawson, and SS Ravi. 2014. Compression of trajectory data: a comprehensive evaluation and new approach. *GeoInformatica* 18, 3 (2014), 435–460.
 - [32] Michalis Potamias, Kostas Patroumpas, and Timos Sellis. 2006. Sampling trajectory streams with spatiotemporal criteria. In *SSDBM*. 275–284.
 - [33] Jianzhong Qi, Guanli Liu, Christian S Jensen, and Lars Kulik. 2020. Effectively learning spatial indices. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2341–2354.
 - [34] Hanan Samet. 1984. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)* 16, 2 (1984), 187–260.
 - [35] Renchu Song, Weiwei Sun, Baihua Zheng, and Yu Zheng. 2014. PRESS: A novel framework of trajectory compression in road networks. *arXiv preprint arXiv:1402.1546* (2014).
 - [36] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.
 - [37] Sheng Wang, Zhifeng Bao, J Shane Culpepper, and Gao Cong. 2021. A survey on trajectory data management, analytics, and learning. *CSUR* 54, 2 (2021), 1–36.
 - [38] Zheng Wang, Cheng Long, and Gao Cong. 2021. Trajectory Simplification with Reinforcement Learning. In *ICDE*. 684–695.
 - [39] Zheng Wang, Cheng Long, Gao Cong, and Ce Ju. 2019. Effective and efficient sports play retrieval with deep representation learning. In *SIGKDD*. 499–509.
 - [40] Zheng Wang, Cheng Long, Gao Cong, and Yiding Liu. 2020. Efficient and effective similar subtrajectory search with deep reinforcement learning. *PVLDB* 13, 12 (2020), 2312–2325.
 - [41] Zheng Wang, Cheng Long, Gao Cong, and Qianru Zhang. 2021. Error-Bounded Online Trajectory Simplification with Multi-Agent Reinforcement Learning. In *KDD*. 1758–1768.
 - [42] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Ake Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning data layouts for big data analytics. In *SIGMOD*. 193–208.
 - [43] Dongxiang Zhang, Mengting Ding, Dingyu Yang, Yi Liu, Ju Fan, and Heng Tao Shen. 2018. Trajectory simplification: an experimental study and quality analysis. *PVLDB* 11, 9 (2018), 934–946.