Error-Bounded Online Trajectory Simplification with Multi-Agent Reinforcement Learning

Zheng Wang[1], Cheng Long[1,*], Gao Cong[1], Qianru Zhang[2]
[1]School of Computer Science and Engineering, Nanyang Technological University, Singapore
[2]Department of Computer Science, The University of Hong Kong, Hong Kong SAR
{wang_zheng,c.long,gaocong}@ntu.edu.sg,qrzhang@cs.hku.hk

ABSTRACT

Trajectory data has been widely used in various applications, including taxi services, traffic management, mobility analysis, etc. It is usually collected at a sensor's side in real time and corresponds to a sequence of sampled points. Constrained by the storage and/or network bandwidth of a sensor, it is common to simplify raw trajectory data when it is collected by dropping some sampled points. Many algorithms have been proposed for the error-bounded online trajectory simplification (EB-OTS) problem, which is to drop as many points as possible subject to that the error is bounded by an error tolerance. Nevertheless, these existing algorithms rely on pre-defined rules for decision making during the trajectory simplification process and there is no theoretical ground supporting their effectiveness. In this paper, we propose a multi-agent reinforcement learning method called MARL4TS for EB-OTS. MARL4TS involves two agents for different decision making problems during the trajectory simplification processes. Besides, MARL4TS has its objective equivalent to that of the EB-OTS problem, which provides some theoretical ground of its effectiveness. We conduct extensive experiments on real-world trajectory datasets, which verify that MARL4TS outperforms all existing algorithms in effectiveness and provides competitive efficiency.

CCS CONCEPTS

• **Information systems** → *Data mining*; • **Computing methodologies** → *Knowledge representation and reasoning*.

KEYWORDS

trajectory data; online trajectory simplification; multi-agent reinforcement learning

ACM Reference Format:
Zheng Wang[1], Cheng Long[1,*], Gao Cong[1], Qianru Zhang[2]. 2021. Error-Bounded Online Trajectory Simplification with Multi-Agent Reinforcement Learning. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '21), August 14–18, 2021, Virtual Event, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3447548.3467351

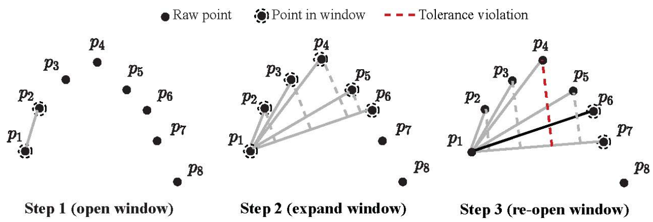


Figure 1: Illustration of the three-step process. Step 1: it opens a window with p_1 and p_2. Step 2: it repeatedly expands the window to p_6 and marks the points p_2, p_3, p_4, p_5, p_6 as safe points because the errors of segments $\overline{p_1p_2}, \overline{p_1p_3}, \overline{p_1p_4}, \overline{p_1p_5}, \overline{p_1p_6}$ are all within a given error tolerance. Step 3: It performs this step since the error of $\overline{p_1p_7}$ violates the tolerance (at point p_4). Specifically, it (1) chooses point p_6 among all safe points based on some pre-defined rule, (2) drops points between p_1 and p_6 (both exclusively), and (3) re-opens a new window involving the chosen safe point p_6 and its following point p_7.

1 INTRODUCTION

Trajectory data, which captures traces of moving objects such as pedestrians, vehicles and robots, has been widely used in different applications, including taxi services, vehicle and fleet management, traffic management, mobility analysis and sports games, etc. Trajectory data is usually collected at a sensor's side, say, by sampling the location of a moving object periodically. Therefore, it corresponds to a sequence of time-stamped locations (called points).

Since the trajectory data is collected in real time, it is often of big volume in a raw form [35]. On the other hand, a sensor is usually constrained by its storage and/or network bandwidth. To deal with these issues of big volume of raw data and constrained resources, a common practice is to *simplify* the trajectory data while it is being collected in an online manner, which is called *online trajectory simplification* (OTS). Specifically, OTS is to drop some raw points of a trajectory before they are stored and/or transmitted and keep the remaining points as a *simplified trajectory*. In this paper, we study an OTS problem, which is to drop as many points as possible (or equivalently to keep as few points as possible) such that the information loss, which is captured as the "error" of the simplified trajectory, is bounded by an error tolerance. We call this problem *error-bounded online trajectory simplification* (EB-OTS).

Many existing algorithms have been developed for EB-OTS problem, including OPW [23], CISED-S [17], BQS [19, 20], FBQS [19, 20], OPERB [18], Intersect [21], Angular [8], and Interval [7]. These algorithms all adopt the following three-step process. Step 1: it

[*] Cheng Long is the corresponding author.

KDD '21, August 14–18, 2021, Virtual Event, Singapore
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8332-5/21/08…$15.00
https://doi.org/10.1145/3447548.3467351

opens a window involving the first and second points. Step 2: it checks whether the segment linking the first point and the last point in the window can approximate the trajectory consisting of all points in the window within the error tolerance. If so, it marks the last point of the window as a *safe point*, expands the window by including one following point outside the window, and repeats Step 2; otherwise, it proceeds to Step 3. Step 3: it (1) chooses a point among all safe points in the window based on some *pre-defined rules*, (2) drops all the points between the first point and the chosen safe point in the window, both exclusively (note that the error caused by dropping these points should be bounded by the error tolerance), and (3) re-opens a new window involving the chosen safe point and the one following the chosen point, and goes back to Step 2. An example illustrating the three-step process is shown in Figure 1.

These existing algorithms differ from each other in (1) adopting different strategies for checking the error in Step 2 and (2) using different pre-defined rules in Step 3. For example, the OPW algorithm computes the error *exactly* in Step 2 and explores two pre-defined rules in Step 3, namely one to choose the last safe point of the window and one to choose the point such that dropping it would cause the greatest error. The CISED-S algorithm computes only an *upper bound* of the error in Step 2 for better efficiency and expands the window if the upper bound is bounded by the error tolerance, and chooses the last safe point in the window in Step 3.

These existing algorithms suffer from two issues. First, in Step 2, they expand the window by one each time, for which it needs to check the error caused by dropping a sequence of points, which incurs expensive computation costs. Note that for algorithms such as OPW, they compute the error exactly for each point, which takes $O(n)$ time, and as a result, they have the time complexity of $O(n^2)$, where n is the number of points of the inputted trajectory. Second, in Step 3, they use pre-defined rules for choosing a safe point for re-opening a window, e.g., choosing the last safe point in the window. These rules are human-crafted and cannot adapt to different problem instances and inputted data. There is no theoretical ground supporting that these rules would achieve the objective of the EB-OTS problem, and as a result, the effectiveness of the algorithms with these rules is not guaranteed.

We observe that the aforementioned three-step process corresponds to a *sequential decision process*, i.e., it sequentially make decisions on how to expand a window (if possible) and on how to choose a safe point for re-opening a window (if expanding a window is not possible). All existing algorithms utilize pre-defined rules for this sequential decision process, e.g., expanding a window by one point in Step 2 and choosing the last safe point in a window in Step 3. Inspired by the fact that reinforcement learning (RL) has been shown to work effectively for sequential decision making problems such as those in robotics [10] and gaming [24], we propose to leverage RL for the EB-OTS problem. Specifically, we introduce two agents, namely Agent-E and Agent-R, for expanding a window and re-opening a window in the three-step process, respectively. Agent-E decides how many points to be included for expanding a window. Agent-R decides which safe point to choose for re-opening a window. For both Agent-E and Agent-R, we carefully define their underlying Markov decision process (MDP) [29] model, including states, actions, transitions and rewards such that the states can be computed efficiently and capture essential information for decision making and the model can be learned effectively. In particular, we let the two agents share their rewards so that they can work cooperatively for achieving effective online trajectory simplification. We call this multi-agent RL-based algorithm for trajectory simplification MARL4TS. MARL4TS is applicable to all error measurements that are commonly used, while most existing algorithms are designed for specific error measurement (e.g., CISED-S is applicable only when the *synchronous Euclidean distance (SED) [23, 26–28] is used for measuring the error of a simplified trajectory*). In addition, we develop two techniques to further boost the efficiency and effectiveness of MARL4TS, namely one of imposing a maximum size over windows to be considered (for improving efficiency) and one of delaying to re-open a window for a certain number of points so as to potentially include more safe points for consideration when re-opening a window (for improving the effectiveness).

In summary, the major contributions of this paper are as follows. First, we develop the first multi-agent RL-based method MARL4TS for the EB-OTS problem. MARL4TS achieves the best effectiveness compared with *all* existing algorithms under *different* parameter settings and error measurements. This marks a significant improvement in trajectory simplification methodology, especially given that trajectory simplification is a fundamental problem, for which a rich set of techniques have been developed. Second, we formally show that the objective of EB-OTS is perfectly captured by that of MARL4TS. This provides some theoretical ground/explanation on the superior effectiveness over existing algorithms, which are based on pre-defined rules and do not optimize EB-OTS's objective directly. In addition, we show that MARL4TS runs in linear time, which provides theoretical justifications of its competitive efficiency. Third, as far as we know, this is the first work that targets the EB-OTS problem under a general error measurement that covers multiple existing error measurements which are widely used. Correspondingly, we probably have conducted the most extensive experiments on EB-OTS in a single research paper (across error measurements, datasets, and settings).

The rest of this paper is organized as follows. We study the literature in Section 2. We review some basic concepts and provide the problem definition in Section 3. We present our MARL4TS algorithm in Section 4 and present the experimental results in Section 5. Finally, we conclude our paper in Section 6.

2 RELATED WORK

(1) Error-Bounded Online Trajectory Simplification. Many existing algorithms have been developed for the EB-OTS problem, including OPW [23], CISED-S [17], BQS [19, 20], FBQS [19, 20], OPERB [18], Intersect [21], Angular [8], Interval [7], and DOTS [3]. All except for DOTS adopt the three-step process as described in Section 1, but use different strategies for expanding and re-opening a window in Step 2 and Step 3, respectively. Specifically, BOPW, NOPW and MOPW are three variants of OPW. They check the error exactly and expand a window by including one point in Step 2. When re-opening a window in Step 3, BOPW chooses the last safe point, NOPW chooses the first safe point that causes the tolerance violation, and MOPW chooses the safe point such that dropping it incurs the greatest error. These three algorithms have the time complexity of $O(n^2)$ and are applicable to multiple error measurements. CISED-S proposes an upper bound estimation via a spatiotemporal cone intersection technique for error checking and expands a window by including one point in Step 2 and adopts the same strategy for Step 3 as BOPW. It has the time complexity of $O(n)$ and works for the synchronous Euclidean distance (SED) error measurement [23, 26–28] only. BQS adopts the same strategies for expanding and re-opening windows as BOPW and improves the efficiency of the error checking procedure in Step 2. The time complexity remains $O(n^2)$ in the worst case. Further, FBQS reduces the time complexity of BQS to $O(n)$ by estimating an upper bound of the error in Step 2. OPERB adopts the same strategies as BOPW and achieves a fast upper bound computation in Step 2 based on a local distance checking method. OPERB has the time complexity of $O(n)$. BQS, FBQS and OPERB are applicable for the perpendicular Euclidean distance (PED) error measurement [1, 19, 20, 23] only. Intersect, Augular and Interval adopt the same strategies as BOPW. They differ from each other in estimating an upper bound of the error in Step 2 and each of them has a time complexity of $O(n)$. They are applicable to the direction-based distance (DAD) error measurement [7, 8, 21, 22] only. DOTS assumes the Integral Square Synchronized Euclidean Distance (ISSED) error measurement, which is not targeted in this paper. Besides, it does not follow the three-step process. Our solution MARL4TS is based on a learned policy from multi-agent reinforcement learning instead of some pre-defined rules for decision making. In addition, MARL4TS is applicable to multiple error measurements including the aforementioned SED, PED, and DAD and speed-based distance (SAD) error measurement [27].

(2) Other Trajectory Simplification Studies. Some other studies target error-bounded trajectory simplification in a batch mode, which assumes full access to the whole trajectory throughout the trajectory simplification process [6, 9, 21, 27]. It is clear that the algorithms proposed in these studies cannot be adopted for the ET-OTS problem. In addition, there are studies on the dual problem of the error-bounded trajectory simplification problem, which is called *size-bounded trajectory simplification* (also called the *Min-Error* problem). Interested readers are referred to a survey paper [35] and a concurrent work [32] and the references therein for details. It is worth mentioning that [32] targets the size-bounded trajectory simplification problem in both the online mode and the batch mode and proposes a reinforcement learning method for the problem. In contrast, this paper targets the EB-OTS problem and develops a multi-agent reinforcement learning method. There are some other studies that develop dead reckoning techniques for online trajectory simplification, which decide whether to discard an incoming point according to some criterion [11, 28, 31].

(3) Road Network based Trajectory Compression. Trajectory compression is process of compressing trajectories that are constrained on road networks with and/or without information loss [4, 5, 13]. The major idea is to first map the raw trajectories onto road networks, and as a result, a trajectory would become a sequence of roads with some temporal information. It then compresses the sequence data and temporal information with various techniques, e.g., a sequence of roads can be compressed to be shorter one involving the first road and the last road if it corresponds to a shortest path from the first road to the last one. In this paper, we focus on the trajectories that are generated in a free space, such as those of animals [21], pedestrians [36], sports players [33], etc.

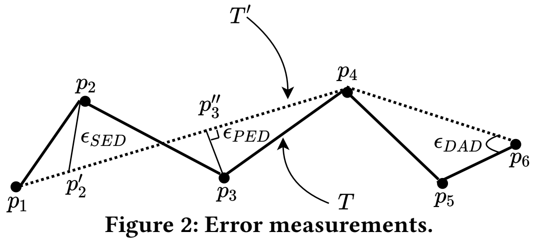


Figure 2: Error measurements.

(4) Multi-agent Reinforcement Learning. Reinforcement learning is a machine learning technique, which is usually used to make sequential decisions when an agent interacts with its surrounding environment [30]. Multi-agent reinforcement learning is for cases where multiple agents are involved [12, 16]. In recent years, multi-agent reinforcement learning has attracted much research attention. For example, it has been used for many problems including express system [14], courier dispatching [15], traffic management [16], similar sub-trajectory search [34], etc. In this paper, we use multi-agent reinforcement learning for error-bounded online trajectory simplification.

3 PROBLEM STATEMENT

(1) Trajectory. A trajectory, denoted by T, consists of a sequence of time-stamped locations to capture the trace of a moving object. Let $T =< p_1, p_2, ..., p_n >$ be a trajectory, where p_i has the form of a triplet (x_i, y_i, t_i), which means that a moving object is at location (x_i, y_i) at time t_i. We use $|T|$ to denote the size of the trajectory T (i.e., $|T| = n$). We use $T[i : j]$ $(i \leq j)$ to denote the subtrajectory of T, which starts form point p_i and ends at point p_j. Let $\overline{p_i p_{i+1}}$ $(1 \leq i \leq n-1)$ denote the line segment which connects two locations (x_i, y_i) and (x_{i+1}, y_{i+1}), meaning that an object moves from (x_i, y_i) to (x_{i+1}, y_{i+1}) along the line segment at a constant speed that equals to $\frac{d(p_i, p_{i+1})}{t_{i+1}-t_i}$, where $d(\cdot, \cdot)$ denotes the Euclidean distance.

(2) Trajectory Simplification. A simplified trajectory of T, denoted by T', has the form of $< p_{s_1}, p_{s_2}, ..., p_{s_m} >$, where $m \leq n$ and $1 = s_1 < s_2 ... < s_m = n$. Consider Figure 2 for example. $T =< p_1, p_2, p_3, p_4, p_5, p_6 >$ is an original trajectory and $T' =< p_1, p_4, p_6 >$ is a simplified trajectory of T.

(3) Error Measurements. Within the time period $[t_{s_j}, t_{s_{j+1}}]$ $(1 \leq j \leq m-1)$, based on the simplified trajectory T', it represents that the object moves along the segment $\overline{p_{s_j} p_{s_{j+1}}}$, while based on the original trajectory T, it represents that the object moves along a sequence of segments which are formed by a sequence of points $p_{s_j}, p_{s_j+1}, ..., p_{s_{j+1}}$. That is, the segment $\overline{p_{s_j} p_{s_{j+1}}}$ in simplified trajectory T' approximates a sequence of segments in the original trajectory starting from the points $p_{s_j}, p_{s_j+1}, ..., p_{s_{j+1}-1}$. Thus, the segment $\overline{p_{s_j} p_{s_{j+1}}}$ is called the *anchor segment* of each of points $p_{s_j}, p_{s_j+1}, ..., p_{s_{j+1}-1}$. All points in the original trajectory T, except for the point p_n, have an anchor segment. For example, in Figure 1, $\overline{p_1 p_4}$ approximates a sequence of three segments $\overline{p_1 p_2}$, $\overline{p_2 p_3}$ and

$\overline{p_3p_4}$ in T and corresponds to the anchor segment of points p_1, p_2 and p_3. Note that there exists some discrepancy between the movement along the simplified trajectory T' and that along the original trajectory T, which is measured as the error of T' wrt T and denoted by $\epsilon(T')$.

Quite a few error measurements have been proposed, among which, four that are widely-used include Synchronized Euclidean Distance (SED) [23, 26–28], Perpendicular Euclidean Distance (PED) [1, 19, 20, 23], Direction-aware Distance (DAD) [7, 8, 21, 22], and Speed-aware Distance (SAD) [27]. The major ideas of these error measurements are as follows. (1) It first defines the error of a segment $\overline{p_{s_j}p_{s_{j+1}}}$ wrt a point p_i that takes the segment as its anchor segment, denoted by $\epsilon(\overline{p_{s_j}p_{s_{j+1}}}|p_i)$ $(s_j \leq i < s_{j+1})$. Different error measurements adopt different definitions for $\epsilon(\overline{p_{s_j}p_{s_{j+1}}}|p_i)$. For illustrations, consider Figure 2, where $\epsilon_{SED}(\overline{p_1p_4}|p_2)$, $\epsilon_{PED}(\overline{p_1p_4}|p_3)$, and $\epsilon_{DAD}(\overline{p_4p_6}|p_5)$ are shown. Detailed definitions can be found in [32]. (2) It then defines the error of a segment $\overline{p_{s_j}p_{s_{j+1}}}$, denoted by $\epsilon(\overline{p_{s_j}p_{s_{j+1}}})$, as the maximum error wrt a point that take the segment as its anchor segment, i.e.,

$$\epsilon(\overline{p_{s_j}p_{s_{j+1}}}) = \max_{s_j \leq i < s_{j+1}} \epsilon(\overline{p_{s_j}p_{s_{j+1}}}|p_i) \tag{1}$$

(3) It finally defines the error of T' as the maximum error of its segments, i.e.,

$$\epsilon(T') = \max_{1 \leq j \leq m-1} \epsilon(\overline{p_{s_j}p_{s_{j+1}}}) \tag{2}$$

Note that the maximum function but not the summation function is adopted by existing studies in Equation (2) since with the latter, it would be sensitive to the size of trajectory, i.e., the error would be accumulated when a trajectory gets longer.

(4) Problem Definition (EB-OTS). We study the EB-OTS problem, which is defined as follows.

Problem 1 (EB-OTS). *Given a raw trajectory $T =< p_1, p_2, ..., p_n >$ that is inputted in a streaming fashion and an error tolerance ε_t, the EB-OTS problem is to find a simplified trajectory T' of T such that $\epsilon(T') \leq \varepsilon_t$ and $|T'|$ is minimized. Here, $\epsilon(T')$ could be instantiated with SED, PED, DAD, or SAD.*

4 METHODOLOGY

We observe that the three-step process as described in Section 1 corresponds to a *sequential decision process*, i.e., it sequentially makes decisions on how to expand a window if possible and how to choose a safe point for re-opening a window if expanding a window is not possible. Therefore, we propose to leverage reinforcement learning (RL) to help the decision making of the three-step process. In particular, RL is widely used to guide agents on how to take actions to maximize a cumulative reward in an environment, where the environment is modelled as a *Markov decision process* [29] (MDP). In this paper, we introduce two agents, namely Agent-E and Agent-R, to help with expanding a window and re-opening a window, respectively. We capture the decision problems of the two agents with MDP models (Section 4.1 and Section 4.2), learn the optimal policies for the MDPs via Deep-Q-Network (DQN)[24] (Section 4.3), and utilize the learned policies of Agent-E and Agent-R for expanding and re-opening a window in the three-step process, respectively (Section 4.4). We call the resulting algorithm MARL4TS.

4.1 MDP for Expanding a Window (Agent-E)

Consider the task of expanding a window in Step 2 of the three-step process. Existing algorithms all adopt a pre-defined rule of expanding a window by including one following point outside the window. While this strategy gives each point a chance to be chosen for re-opening a window in Step 3 if it is a safe point, it is somehow a bit too conservative. For instance, in a scenario where the underlying object moves along a straight line and at a constant speed, we have much confidence to drop the corresponding sampled points. This motivates us to potentially include multiple points for expanding a window so as to reduce the number of times of checking the error of dropping points from a window. Specifically, we introduce an agent called Agent-E for deciding how many points to be included for expanding a window. We capture this decision making problem as a MDP, involving states, actions, transitions and rewards, which are defined as follows.

(1) States. We denote a state of Agent-E's MDP by s^e. Suppose the window involves points $p_l, p_{l+1}, ..., p_r$, which we denote by $W[l, r]$. Intuitively, the state should capture essential information of the window for deciding how many points to be included when expanding the window. We propose to define the state as a pair of two numbers, namely, (1) the ratio between the distance from p_l to p_r when traveled along the raw trajectory and that when traveled along the segment $\overline{p_lp_r}$ and (2) the number of points in the window. Formally, the state s^e is defined as follows.

$$s^e := (\frac{\sum_{l \leq h < r} d(p_h, p_{h+1})}{d(p_l, p_r)}, r - l + 1) \tag{3}$$

The intuition of the above definition is as follows. The first number captures how smooth the underlying trace is to some extent, e.g., if it is close to 1, it means that the trace is close to a straight line, and if it is significantly larger than 1, it means that trace may involve big turns. The second number captures the size of the window. In addition, with this definition, the state can be updated in $O(1)$ time, which is important since the main purpose of Agent-E is to boost the efficiency of the three-step process.

(2) Actions. We denote an action of Agent-E's MDP by a^e. Suppose the current window is $W[l, r]$. We define J possible actions for expanding $W[l, r]$, namely, (1) expanding to $W[l, r+1]$, (2) expanding $W[l, r + 2]$, ..., (J) expanding to $W[l, r + J]$. Here, J is a hyperparameter, which is to be decided via empirical studies. Formally, we define a^e as follows.

$$a^e := j \ (1 \leq j \leq J) \tag{4}$$

where $a^e = j$ means to expand $W[l, r]$ to $W[l, r + j]$. Note that when J is set to 1, it reduces to the conventional rule of always expanding a window by including one following point outside the window.

(3) Transitions. Let s^e be a state and $W[l, r]$ be the window at s^e. Suppose we take an action $a^e = j$, which expands the window to $W[l, r + j]$. There are two cases. Case 1: the error of dropping the points in the window, i.e., $\epsilon(\overline{p_lp_r})$, is bounded by the error tolerance ε_t. In this case, it would continue to expand the window $W[l, r + j]$ (in Step 2). A new state can be computed based on the window $W[l, r + j]$. Case 2: it corresponds to the other case. In this case, it would re-open a window with the Agent-R (to be introduced in

Section 4.2) (in Step 3). A new state can be computed based on the re-opened window. Note that a re-opened window can always be expanded further since it involves exactly two adjacent points.

(4) Rewards. When expanding a window, it does not drop any points and thus the rewards cannot be immediately observed. Since Agent-E and Agent-R cooperate for the same objective, i.e., to drop as many points as possible without violating the error tolerance, we let Agent-E and Agent-R share their rewards. Specifically, we set the reward of performing an action of expanding a window to be equal to that of the following action of re-opening a window (which will be introduced in Section 4.2).

4.2 MDP for Re-Opening a Window (Agent-R)

Consider the task of re-opening a window in Step 3 of the three-step process. The key is to choose a safe point in the window since once it has been chosen, a window can be re-opened by including the chosen safe point and the point following the chosen safe point. Existing algorithms all use pre-defined rules for choosing a safe point, e.g., choosing the *last* safe point in the windows. These pre-defined rules are simple heuristics and cannot adapt to different inputs with different settings. In addition, there is no theoretical ground supporting that they optimize the objective of the EB-OTS problem. We introduce an agent called Agent-R for deciding which safe point in the window to choose for re-opening a window. We capture this decision making problem as a MDP, which are defined as follows.

(1) States. We denote a state of Agent-R's MDP by s^r. Suppose that the current window is $W[l, r]$ and M safe points have been marked in Step 2. We denote these safe points by $p_{s_1}, p_{s_2}, ..., p_{s_M}$. We need to choose a point among these M safe points. An immediate idea is to incorporate all M safe points for defining the state. Nevertheless, it would have two issues. First, the state defined in this way would be M-dependent and cannot be used for other cases where the number of safe points is not M. Second, M is typically a large integer, and with this definition, the state space would be huge, which introduces difficulty to train the model effectively.

We propose to focus on the last K safe points, i.e., $p_{s_{M-K+1}}$, $p_{s_{M-K+2}}, ..., p_{s_M}$, for defining a state, where K is a hyperparameter that can be tuned. The rationale is that to re-open a window at a safe point that appears later would mean to drop more points. In case that there are fewer than K safe points, we simply duplicate the last safe point a few times to obtain K safe points. For each safe point p_{s_i} considered, we define a pair of two numbers, namely (1) the error of the segment linking p_l and p_{s_i} and (2) the number of points that would be dropped if point p_{s_i} is chosen for re-opening a window. We denote the pair by $c(p_{s_i})$, i.e., $c(p_{s_i}) = (\epsilon(\overline{p_l p_{s_i}}), s_i - l + 1)$. Note that $\epsilon(\overline{p_l p_{s_i}})$ has already been computed when checking whether the window $W[l, s_i]$ can be expanded in Step 2. These numbers capture the the consequence of choosing p_{s_i} for re-opening a window to some extent. Formally, we define s^r as follows.

$$s^r := \{c(p_{s_{M-K+1}}), c(p_{s_{M-K+2}}), ..., c(p_{s_M})\}. \tag{5}$$

The above definition is M-independent. In addition, with this definition, the state space is controllable via the hyperparameter K. Note that we would normalize the values of the state in each dimension to avoid potential data scale issues.

(2) Actions. We denote an action of Agent-R's MDP by a^r. We define K actions, each to choose one of the last K safe points for re-opening a window. The rationale is the same as that of defining the state s^r. Formally, we defne a^r as follows.

$$a^r := k \ (0 \leq k \leq K - 1) \tag{6}$$

where the action $a^r = k$ means to choose the safe point $p_{s_{M-k}}$, drop all the points between p_l and $p_{s_{M-k}}$ (both exclusively), and re-open a window involving $p_{s_{M-k}}$ and $p_{s_{M-k}+1}$, i.e., $W[s_{M-k}, s_{M-k} + 1]$.

(3) Transitions. After we take an action a^r at a state s^r, which re-opens a window. It would continue to expand the window iteratively until it is not possible to do so. By then, some safe points would have been marked, based on which, we can compute a new state $s^{r'}$, which corresponds to the next state of s^r.

(4) Rewards. Consider that Agent-R performs an action a^r at a state s^r and then it arrives at a new state $s^{r'}$. We define the reward associated with this transition from state s^r to state $s^{r'}$, which we denote by R, as follows. Let $W[l, r]$ and $W[l', r']$ be the windows before and after the action a^r is taken, respectively. Recall that the action a^r involves the process of dropping the points between p_l and $p_{l'}$. We therefore define the reward to be equal to the number of points dropped. Formally, we define R as follows.

$$R := l' - l - 1 \tag{7}$$

The intuition is that if more points are dropped, it is more towards the objective of the EB-OTS problem and thus the reward should be larger. It is clear that with this reward definition, the objective of the Agent-R's MDP, i.e., maximizing the accumulative rewards, is equivalent to that of the EB-OTS problem, i.e., maximizing the number of points dropped. A formal verification of this equivalence is as follows. Suppose that the Agent-R goes through a sequence of N states, $s_1^r, s_2^r, ..., s_N^r$ and correspondingly it receives a sequence of $N-1$ rewards, $R_1, R_2, ..., R_{N-1}$. Let $W[l_1, r_1], W[l_2, r_2], ..., W[l_N, r_N]$ be the windows at the N states. Note that $l_1 = 1$ and N is the number of points kept among the first l_N points. Then, the accumulative rewards can be computed as follows.

$$\sum_{t=1}^{N-1} R_t = \sum_{t=1}^{N-1} (l_{t+1} - l_t - 1) = l_N - l_1 - N + 1 = l_N - N \tag{8}$$

where $l_N - N$ corresponds to the number of points dropped before point p_{l_N}.

4.3 Learning Policies of MDPs

We adopt the Deep-Q-Network (DQN) [25] method for learning the policies from the MDPs of Agent-E and Agent-R, which has been widely used for MPDs with continuous state spaces. The major idea of DQN is as follows. It first defines an optimal action-value function $Q^*(s, a)$ (called Q function), which represents the maximum expected accumulative rewards it would receive by following any policy after taking the action a at the state s. Formally, $Q^*(s, a)$ is defined as follows.

$$Q^*(s, a) = E_{s'}[R + \gamma \cdot \max_{a'} Q^*(s', a')] \tag{9}$$

where s' is a possible next state and R is the reward observed after action a is taken at a given state s, and γ is a discount factor. It then estimates $Q^*(s, a)$ by a parameterized neural network denoted

Algorithm 1: The MARL4TS algorithm

```
Input: A trajectory $T =< p_1, p_2, ..., p_n >$ which is inputted in an online
    fashion; A given error tolerance $\varepsilon_t$;
Output: A simplified trajectory $T'$ with $\varepsilon(T') \leq \varepsilon_t$;
1  //Step 1
2  Open a window $W[1, 2]$;
3  while true do
4      //Step 2
5      Let $W[l, r]$ denote the current window;
6      if $\epsilon(\overline{p_l p_r}) \leq \varepsilon_t$ then
7          Mark $p_r$ as a safe point;
8          if $p_r$ is the last point, i.e., $p_n$ then
9              Drop all points between $p_l$ and $p_r$;
10             Break;
11         end
12         Construct a state $s^e$ (as in Equation (3));
13         Choose an action $a^e = \arg\max_a Q^*(s^e, a; \theta^e)$;
14         Expand the window to $W[l, r + j]$ where $a^e = j$ (Round $r + j$ to
               be $n$ if $p_{r+j}$ will not arrive, i.e., $r + j > n$);
15     else
16         //Step 3
17         Construct a state $s^r$ (as in Equation (5));
18         Choose an action $a^r = \arg\max_a Q^*(s^r, a; \theta^r)$;
19         Drop all points between $p_l$ and the chosen safe point indicated by
               $a^r$;
20         Re-open a window involving the chosen safe point and the point
               following the chosen safe point;
21     end
22 end
23 Return trajectory $T'$ consisting of all points that have not been dropped;
```

by $Q^*(s, a; \theta)$ (for details, please refer to [25]). It finally returns the policy, which always chooses for a given state s the action a that maximizes $Q^*(s, a; \theta)$. We denote the estimated Q functions of Agent-E and Agent-R by $Q^*(s, a; \theta^e)$ and $Q^*(s, a; \theta^r)$, respectively.

4.4 The MARL4TS Algorithm

The MARL4TS algorithm follows the three-step process and uses Agent-E and Agent-R for expanding and re-opening a window, respectively. We present MARL4TS in Algorithm 1. Specifically, it first opens a window $W[1, 2]$ (line 1-2). It then enters a while loop (line 3-22). Let $W[l, r]$ be the current window (line 5). It checks whether $\epsilon(\overline{p_l p_r})$ is bounded by the error tolerance. If so, it performs the following steps (line 7-14). It first marks p_r as a safe point (line 7). It then terminates the loop if p_r is the last point by dropping all the points in the window except for p_l and p_r (line 8-11) and expands the window with Agent-E otherwise (line 12-14). If not, it chooses a safe point with Agent-R (line 17-18), drops all points between the first point and the chosen safe point in the window (line 19) and re-opens a new window involving the chosen safe point and the point following the chosen safe point (line 20). Finally, it returns a simplified trajectory involving all points that have not been dropped (line 23).

We illustrate the MARL4TS algorithm with an example shown in Figure 3, where the inputted trajectory is shown at the left side. It first opens a window $W[1, 2]$. Since $\epsilon(\overline{p_1 p_2})$ is bounded by ε_t, it marks p_2 as a safe point and uses Agent-E to expand the window to $W[1, 3]$. Similarly, it expands the window to $W[1, 8]$ with Agent-E. Since $\epsilon(\overline{p_1 p_8})$ is not bounded by ε_t, it uses Agent-R to choose a safe point p_7, drops the points p_2, p_3, p_4, p_5, p_6 and re-opens a window $W[7, 8]$. Since $\epsilon(\overline{p_7 p_8})$ is bounded by ε_t and p_8 is the last point, it breaks from the loop and returns the simplified trajectory $T' =< p_1, p_7, p_8 >$.

Table 1: Dataset statistics.

Statistics	Geolife	T-Drive	Indoor
# of trajectories	17,621	10,359	529,397
Total # of points	24,876,978	17,740,902	330,117,253
Ave. # of points per trajectory	1,412	1,713	624
Sampling rate	1s ~ 5s	177s	0.03s ~ 0.06s
Average distance	9.96m	623m	0.037m

We further develop two techniques for boosting the effectiveness and efficiency of MARL4TS as follows. First, in Step 3 of the three-step process, before we choose a safe point, we check for each of the D points that follow p_r, i.e., the last point in the window, whether it is a safe point. Here, D is a hyperparameter. We then focus on the last K safe points among all safe points in the window and those from the D points for defining a state and choosing among them a safe point for re-opening a window. The rationale is to consider potentially more safe points for more effective trajectory simplification, which is verified via experiments. Second, in Step 2 of the three-step process, we impose a constraint that the window to be expanded has its size bounded by C. Here, C is a hyperparameter. In the case the size of an expanded window exceeds C, its right end would be shifted towards the left until the size reaches C. The rationale is that the cost of checking the error of dropping all points in a window (except for the first and the last points) depends on the size of the window, i.e., it is $O(n)$ in the worst-case if the size of a window is not bounded. With a bounded window size, the cost would be $O(C)$. The benefit of this technique for improving the efficiency is verified in experiments.

Time complexity. The time complexity of the MARL4TS algorithm is $O(n)$, where n denotes the number of points in the inputted trajectory T, which we analyze as follows. The cost is dominated by those of Step 2 and Step 3. In Step 2, the cost of one transition consists of (1) that of checking the error of a segment, which is bounded by $O(1)$ given that C is a constant and (2) that of constructing a state and sampling an action for Agent-E, which is $O(1)$. In Step 3, the cost of one transition includes (1) that of constructing the state based on the last K safe points for Agent-R, which is $O(1)$ given that K is a constant and the errors of segments have already been computed during Step 2 and (2) that of sampling an action, choosing a safe point, dropping points, which is $O(1)$. In addition, Step 2 involves at most nC transitions and Step 3 involves no more transitions than Step 2. Therefore, the time complexity of MARL4TS is $O(nC) \times O(1) = O(n)$.

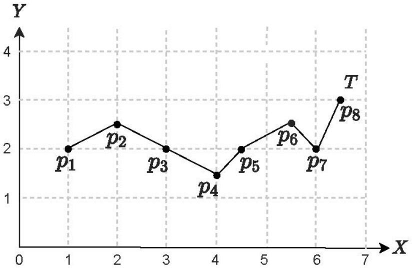
5 EXPERIMENTS

5.1 Experimental Setup

Dataset. We conduct our experiments on three real-world trajectory datasets, namely Geolife, T-Drive and Indoor, which are widely used for evaluating the performance of trajectory simplification in previous work [18, 21, 22] including the recent dedicated evaluation work [35]. We show the statistics of datasets in Table 1. Specifically, Geolife[1] records the outdoor trajectories of 182 users in different transportation modes like walking and driving for a period of five years. T-Drive [2] tracks the trajectories of 10,357 taxis in Beijing, and

[1] http://research.microsoft.com/en-us/downloads/b16d359d-d164-469e-9fd4-daa38f2b2e13/

[2] http://research.microsoft.com/apps/pubs/?id=152883



Initial	$l = 1, r = 2$ and $\varepsilon_t = 1.0$				
$W[l,r]$	$\epsilon(\overline{p_l p_r})$	Safe points	Agent	State	Action
$W[1,2]$	$\epsilon(\overline{p_1 p_2}) = 0.0 < \varepsilon_t$	$< p_2 >$	E	$s_1^e = \{1.0, 2\}$	Expand to p_3
$W[1,3]$	$\epsilon(\overline{p_1 p_3}) = 0.5 < \varepsilon_t$	$< p_2, p_3 >$	E	$s_2^e = \{1.118, 3\}$	Expand to p_5
$W[1,5]$	$\epsilon(\overline{p_1 p_5}) = 0.5 < \varepsilon_t$	$< p_2, p_3, p_5 >$	E	$s_3^e = \{1.160, 5\}$	Expand to p_7
$W[1,7]$	$\epsilon(\overline{p_1 p_7}) = 0.5 < \varepsilon_t$	$< p_2, p_3, p_5, p_7 >$	E	$s_4^e = \{1.177, 7\}$	Expand to p_8
$W[1,8]$	$\epsilon(\overline{p_1 p_8}) = 1.029 > \varepsilon_t$	$< p_2, p_3, p_5, p_7 >$	R	$s_1^r = \{(0.5, 5), (0.5, 7)\}$	Re-open at p_7
$W[7,8]$	$\epsilon(\overline{p_7 p_8}) = 0.0 < \varepsilon_t$	$< p_8 >$	-	-	-
Output	Return $T' =< p_1, p_7, p_8 >$				

Figure 3: Running example of MARL4TS with PED.

Indoor [3] contains the trajectories of visitors in the "ATC" shopping center in Osaka, Japan from October 2012 to November 2013.

Baselines. To evaluate the effectiveness and efficiency of the proposed method MARL4TS, we have carefully examined the evaluation paper [35] and the recent literature to cover all baselines that are proposed or can be adapted for the EB-OTS problem, including OPW [23], CISED-S [17], BQS [19, 20], FBQS [19, 20], OPERB [18], Intersect [21], Angular [8], and Interval [7]. The descriptions of these algorithm can be found in Section 2.

Parameter Settings. For Agent-E, we use a feedforward neural network with 2 layers. In the first layer, we use the tanh function with 23 neurons, and in the second layer, we use the linear function with J neurons as the output corresponding to different actions. In order to avoid the data scale issues if any, batch normalization is employed before the activation. For Agent-R, we use a feedforward neural network with one hidden layer followed by one output layer. The hidden layer has 25 neurons with the tanh function as the activation and the output layer has K neurons corresponding to different actions. Before training the neural networks, we shuffle the states by randomly positioning the K tuples in each state to improve the model quality [2]. The default hyperparameters J, K, D, C are set as 2, 5, 5 and 50, respectively. The effects of these parameters are studied in experiments. We randomly sample 40%, 40%, 2% trajectories from Geolife, T-Drive, Indoor dataset for training and choose the best models during the process, and the remaining trajectories are used for testing. For both agents, the size of the replay memory M is set at 2000. In addition, we train the model using Adam stochastic gradient descent with an initial learning rate of 0.01. The minimal ϵ is set at 0.1 with decay 0.99 for the ϵ-greedy strategy, and the reward discount rate γ is set at 0.99.

5.2 Experimental Results

(1) Effectiveness evaluation (comparison with existing approximate algorithms). We randomly sample 1000 trajectories from a dataset and vary the error tolerance ε_t from 20m to 100m for SED and PED by following [17, 18], 0.2 radian to 1.0 radian for DAD by following [21], and 5km/h to 25km/h for SAD by following [27]. Figure 4 shows the average compression ratios of MARL4TS and the existing algorithms. MARL4TS consistently outperforms all the existing algorithms under all error measurements due to its data-driven nature. For example, BOPW is the best baseline for SED, PED and SAD, and MARL4TS outperforms it by 4% (resp. 5% and 4%) for SED (resp. PED and SAD); Interval is the best for DAD, and MARL4TS outperforms it by 3%. Note that the improvement contributes a lot in practice because the storage overhead of trajectories is generally huge.

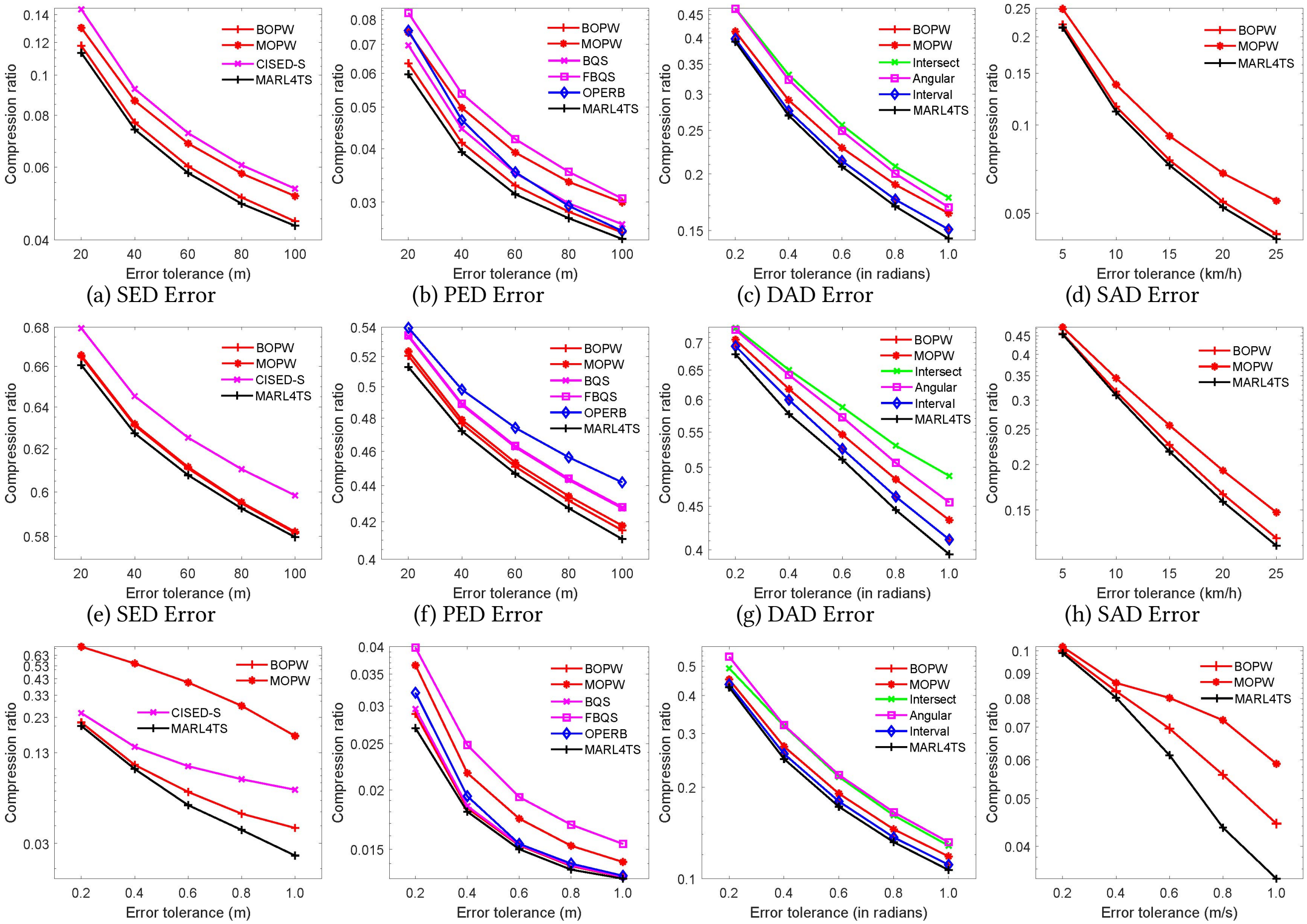
Table 2: Ablation study of the learned policy (Geolife).

Measurement	SED		PED	
Algorithm	Ratio	Time (s)	Ratio	Time (s)
MARL4TS	11.864%	0.88	6.304%	0.46
w/o Agent-E	11.857%	0.97	6.299%	0.56
w/o Agent-R	13.386%	0.61	6.744%	0.33
w/o Agent-E and Agent-R	13.019%	0.70	6.632%	0.36

(2) Effectiveness evaluation (ablation study of the learned policy). To show the effect of the learned policy in MARL4TS model, we conduct an ablation study by replacing the policies of Agent-E, Agent-R, or both Agent-E and Agent-R with the following predefined rules: it expands a window by including one point to replace Agent-E (Step 2) and it chooses the last safe point to replace Agent-R (Step 3). Table 2 reports the average compression ratio and running time on 1,000 randomly sampled trajectories from Geolife (ε_t is fixed at 20m) under the SED and PED error. We can see all these components contribute to the effectiveness or efficiency. For example, Agent-E improves the efficiency by 9.3% and Agent-R improves the effectiveness by 11.4%. As expected, these components illustrate a trade-off between the efficiency and effectiveness. The results based on other error measurements and datasets show a similar trend and thus omitted.

(3) Efficiency evaluation (varying the trajectory length $|T|$). To evaluate the effect of trajectory length $|T|$ on efficiency, we follow [18, 21] by varying $|T|$ from 20,000 to 100,000. For each setting, we randomly select 100 trajectories and fix ε_t at 20m for SED and PED, 0.4 radian for DAD, and 5km/h for SAD. We report the results of average processing time per point in Figure 5, since a trajectory is fed point by point in the online scenario. In addition, we show the sampling rate (1s) of the Geolife dataset with a dotted line for ease of reference. We observe that MARL4TS is slower than some baselines that are designed for a specific error (i.e., OPERB for PED, Intersect, Angular and Interval for DAD), though they have the same time complexity. This is because these existing algorithms are tailored for specific error measurements, e.g., OPERB developed a local distance checking method for PED, while MARL4TS is designed for some general error measurement. In addition, we notice MARL4TS is much faster than the opening window algorithms (i.e., BOPW, NOPW and MOPW) since it employs Agent-E to save some computation of error checking. Overall, MARL4TS runs faster than most of existing baselines and would meet the practical needs: for a trajectory with about 10,000 points, it takes around 0.1ms per point, which is 10,000 times faster than the sampling rate (1s). The results

[3] http://www.irc.atr.jp/crest2010_HRI/ATC_dataset/

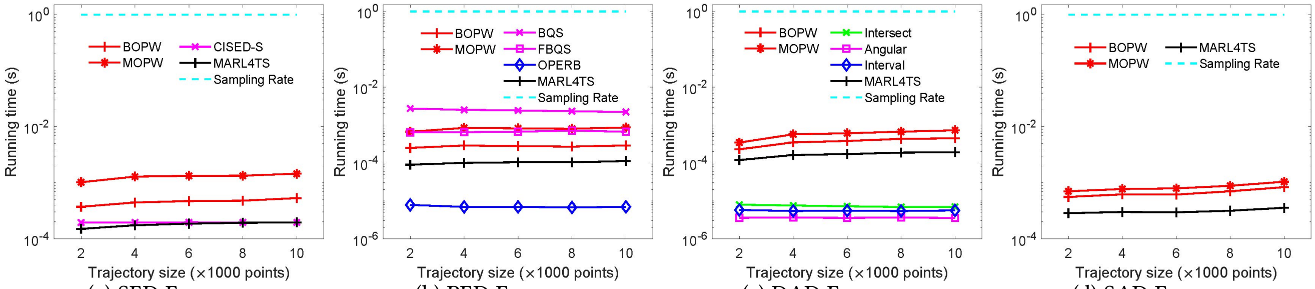


(a) SED Error (b) PED Error (c) DAD Error (d) SAD Error

(e) SED Error (f) PED Error (g) DAD Error (h) SAD Error

(i) SED Error (j) PED Error (k) DAD Error (l) SAD Error

Figure 4: Effectiveness with varying the error tolerance ε_t (Geolife (a)-(d), T-Drive (e)-(h), and Indoor (i) - (l)).



(a) SED Error (b) PED Error (c) DAD Error (d) SAD Error

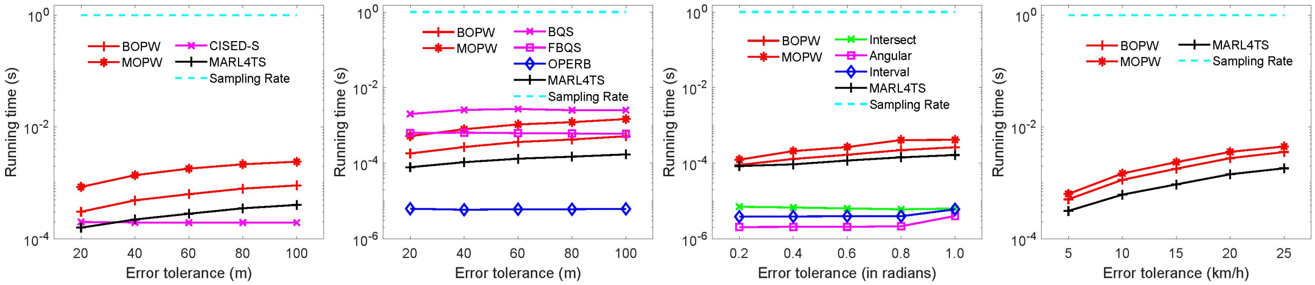
Figure 5: Efficiency with varying the trajectory length $|T|$ (Geolife).

on other datasets are qualitatively similar as those reported and are omitted.

(4) Efficiency evaluation (varying the error tolerance ε_t). To study the efficiency with varying error tolerances ε_t, we set ε_t from 20m to 100m for PED and SED by following [18], 0.2 radian to 1.0 radian for DAD by following [21], and 5km/h to 25km/h for SAD on Geolife. For each setting, we randomly sample 100 trajectories with a fixed $|T|$ at 50,000. Figure 6 shows the results of average running time. Similarly, MARL4TS is slower than some algorithms that are specifically designed for one error measurement, but runs reasonably fast. For example, it takes around 1ms per point for a trajectory with 50,000 points. The running time for all the methods slightly increases with ε_t. This is because the window would be incrementally extended with a larger setting of ε_t, and the corresponding time cost of checking the window increases.

6 CONCLUSION

In this paper, we study the error-bounded online trajectory simplification problem called EB-OTS, which is to drop as many points as possible from a trajectory which is inputted in a streaming manner



(a) SED Error (b) PED Error (c) DAD Error (d) SAD Error

Figure 6: Efficiency with varying the error tolerance ε_t (Geolife).

while the information loss, captured as the "error", is bounded by an error tolerance. We propose a multi-agent RL-based method called MARL4TS. Compared with existing algorithms, MARL4TS is data-driven and applicable to multiple error measurements. We conduct extensive experiments on real-world trajectory datasets, which show that MARL4TS simplifies trajectories with consistently lower compression ratios and runs comparably fast. In the future, we plan to explore the error-bounded trajectory simplification problem in the batch scenario, i.e., the whole trajectory remains accessible during the simplification process.

Acknowledgments. This research was supported by the Nanyang Technological University Start-Up Grant from the College of Engineering under Grant M4082302, by the Ministry of Education, Singapore, under its Academic Research Fund Tier 1 (RG20/19 (S) and RG114/19 (S)), and by a MOE Tier-2 grant MOE2019-T2-2-181.

REFERENCES

[1] Richard Bellman. 1961. On the approximation of curves by line segments using dynamic programming. *Commun. ACM* 4, 6 (1961), 284.

[2] Yoshua Bengio. 2012. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*. Springer, 437–478.

[3] Weiquan Cao and Yunzhao Li. 2017. DOTS: An online and near-optimal trajectory simplification algorithm. *Journal of Systems and Software* 126 (2017), 34–44.

[4] Chao Chen, Yan Ding, Xuefeng Xie, Shu Zhang, Zhu Wang, and Liang Feng. 2019. TrajCompressor: An online map-matching-based trajectory compression framework leveraging vehicle heading direction and change. *IEEE TITS* 21, 5 (2019), 2012–2028.

[5] Yunheng Han, Weiwei Sun, and Baihua Zheng. 2017. COMPRESS: A comprehensive framework of trajectory compression in road networks. *TODS* 42, 2 (2017), 1–49.

[6] John Edward Hershberger and Jack Snoeyink. 1992. *Speeding up the Douglas-Peucker line-simplification algorithm.* University of British Columbia, Department of Computer Science.

[7] Bingqing Ke, Jie Shao, and Dongxiang Zhang. 2017. An efficient online approach for direction-preserving trajectory simplification with interval bounds. In *MDM*. IEEE, 50–55.

[8] 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*. Springer, 79–91.

[9] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. 2001. An online algorithm for segmenting time series. In *ICDM*. IEEE, 289–296.

[10] Jens Kober, J Andrew Bagnell, and Jan Peters. 2013. Reinforcement learning in robotics: A survey. *IJRR* 32, 11 (2013), 1238–1274.

[11] Ralph Lange, Frank Dürr, and Kurt Rothermel. 2011. Efficient real-time trajectory tracking. *VLDBJ* 20, 5 (2011), 671–694.

[12] Hyun-Rok Lee and Taesik Lee. 2019. Improved cooperative multi-agent reinforcement learning algorithm augmented by mixing demonstrations from centralized policy. In *AAMAS*. 1089–1098.

[13] 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.

[14] Yexin Li, Yu Zheng, and Qiang Yang. 2019. Efficient and Effective Express via Contextual Cooperative Reinforcement Learning. In *SIGKDD*. 510–519.

[15] Yexin Li, Yu Zheng, and Qiang Yang. 2020. Cooperative Multi-Agent Reinforcement Learning in Express System. In *CIKM*. 805–814.

[16] 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.

[17] Xuelian 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.

[18] Xuelian Lin, Shuai Ma, Han Zhang, Tianyu Wo, and Jinpeng Huai. 2017. One-pass error bounded trajectory simplification. *PVLDB* 10, 7 (2017), 841–852.

[19] 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*. IEEE, 987–998.

[20] 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. *IEEE TKDE* 28, 11 (2016), 2827–2841.

[21] Cheng Long, Raymond Chi-Wing Wong, and HV Jagadish. 2013. Direction-preserving trajectory simplification. *PVLDB* 6, 10 (2013), 949–960.

[22] Cheng Long, Raymond Chi-Wing Wong, and HV Jagadish. 2014. Trajectory simplification: on minimizing the direction-based error. *PVLDB* 8, 1 (2014), 49–60.

[23] Nirvana Meratnia and A Rolf. 2004. Spatiotemporal compression techniques for moving point objects. In *EDBT*. Springer, 765–782.

[24] 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).

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.

[26] 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 *Proceedings of the 2nd International Conference on Computing for Geospatial Research & Applications*. ACM, 13.

[27] 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.

[28] Michalis Potamias, Kostas Patroumpas, and Timos Sellis. 2006. Sampling trajectory streams with spatiotemporal criteria. In *SSDBM*. IEEE, 275–284.

[29] Martin L Puterman. 2014. *Markov Decision Processes.: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.

[30] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction.* MIT press.

[31] Goce Trajcevski, Hu Cao, Peter Scheuermanny, Ouri Wolfsonz, and Dennis Vaccaro. 2006. On-line data reduction and the quality of history in moving objects databases. In *Proceedings of the 5th ACM international workshop on Data engineering for wireless and mobile access*. ACM, 19–26.

[32] Zheng Wang, Cheng Long, and Gao Cong. 2021. Trajectory Simplification with Reinforcement Learning. In *ICDE*.

[33] Zheng Wang, Cheng Long, Gao Cong, and Ce Ju. 2019. Effective and efficient sports play retrieval with deep representation learning. In *SIGKDD*. 499–509.

[34] 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.

[35] 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.

[36] Yu Zheng, Xing Xie, Wei-Ying Ma, et al. 2010. Geolife: A collaborative social networking service among user, location and trajectory. *IEEE Data Eng. Bull.* 33, 2 (2010), 32–39.

ADDITIONAL EXPERIMENTS

Evaluation Platform. All the methods are implemented in Python 3.6. The implementation of MARL4TS is based on Keras 2.2.0. The experiments are conducted on a machine with Intel(R) Xeon(R) CPU E5-1620 v2 @3.70GHz 16.0GB RAM and one Nvidia GeForce GTX 1070 GPU. The codes and datasets can be downloaded via the link [4] to reproduce the results.

(1) Training time. With the default setup in Section 5.1, we report the training time of the MARL4TS model under different error measurements across different datasets in Table 3. It normally takes from hours to days to train a satisfactory RL model.

(2) Comparison with the exact algorithm SP. We study the effectiveness of MARL4TS by comparing with the algorithm SP [21], which is an exact algorithm for the offline version of the ET-OTS problem (i.e., all points of a trajectory remain accessible throughout the trajectory simplification process) and has the time complexity of $O(n^3)$. The experiment is conducted on small datasets only by randomly selecting a set of 100 trajectories from Geolife because SP is very slow. As shown in figure 8, the compression ratio of MARL4TS is very close to that of SP under SED, PED, DAD and SAD; however, MARL4TS is faster than SP by around three orders of magnitude. We omit the results on other datasets since they are qualitatively similar.

(3) Case study. We present a case study in Figure 9, where the blue solid lines indicate a raw trajectory and the red points indicate the points in the simplified trajectories by different algorithms under SED. We label the compression ratio for each algorithm. The results clearly show that MARL4TS returns better results than baselines. For example, the ratio of MARL4TS (i.e., 6.1%) is clearly smaller than CISED-S (i.e., 9.7%).

(4) Transferability Test. We conduct a transferability test by applying the model learned from T-Drive dataset to Geolife dataset directly without any adaptation and vice versa. The results are reported in Figure 7. We observe that MARL4TS has a good transferability. The average compression ratios are very close though the models are learned from different datasets, and consistently outperform the best baseline (i.e., BOPW).

(5) Effectiveness evaluation (varying parameter *J*). In table 4, we report the effects of hyper-parameter J (i.e., the maximum number of points that could be included for expanding a window) for MARL4TS on randomly sampled 1000 trajectories from Geolife under SED. As expected, we observe the compression ratio degrades but the efficiency improves as J increases. This is because with a larger J, Agent-E tends to expand more points, which lead to the number of safe points that are provided to Agent-R for re-opening the window become fewer and correspondingly the space of possible simplified trajectories becomes smaller. In addition, the effort of constructing states and actions becomes less. We also report the results of a statistic called pruning, which is the relative reduction of the number of error checks when $J = i$ compared with when $J = 1$, i.e., $\frac{\#AC_{J=1} - \#AC_{J=i}}{\#AC_{J=1}}$, where $\#AC_{J=i}$ denotes the number of error checks when $J = i$. We choose $J = 2$ as the default setting since it gives satisfactory effectiveness and efficiency.

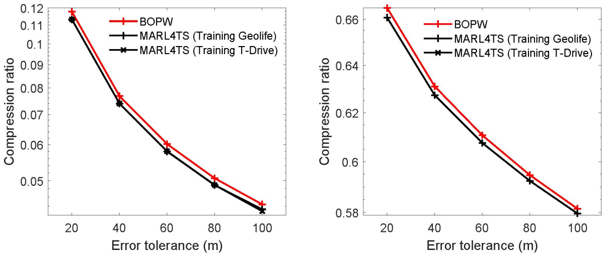
(6) Effectiveness evaluation (varying parameter *K*). Table 5 shows impacts of parameter K for MARL4TS under the SED error. We randomly sample 1000 trajectories from Geolife and use parameter K to control the state space. When K is fixed, a state is of a fixed size. As shown in Table 5, the compression ratio becomes smaller with the increase of parameter K since the state space contains more safe points, for which the action could have more choices. As expected, time cost becomes larger as K increases. We choose $K = 5$ as the default setting because the setting provides a good trade-off between efficiency and effectiveness.

(7) Effectiveness evaluation (varying parameter *D*). Parameter D controls the number of points in a window with a delaying mechanism in order to contain more safe points for Agent-E to pick. Table 6 reports the average compression ratio on 1,000 randomly sampled trajectories from Geolife under SED. We observe that with the increase of D, the compression ratio becomes smaller and the running time becomes larger. This is because introducing the delayed mechanism is useful for optimizing MARL4TS model. Meanwhile, as D increases, the time cost increases correspondingly, which is as expected.

(8) Effectiveness evaluation (varying parameter *C*). Table 7 shows the effects of the parameter C, which is a constraint mechanism to limit the maximum size over windows for MARL4TS. We randomly sample 1000 trajectories from Geolife and report the average compression ratio under SED. We observe the ratio improves but the time cost becomes larger as C increases. This is consistent with the objective of the constraint mechanism to lower the time complexity of checking errors while a small C would make it miss some potential candidate points for Agent-R to re-open a new window.

Table 3: Training time (hours).

Measurement	SED	PED	DAD	SAD
Geolife	51.6	36.9	123.6	94.8
T-Drive	33.1	30.1	47.9	35.3
Indoor	29.4	9.7	62.7	54.5



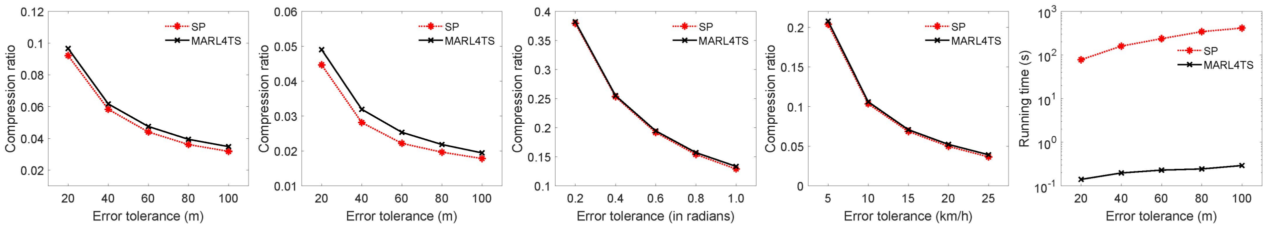
(a) Effectiveness on Geolife (b) Effectiveness on T-Drive

Figure 7: Transferability Test

Table 4: Impacts of parameter *J* for MARL4TS (Geolife).

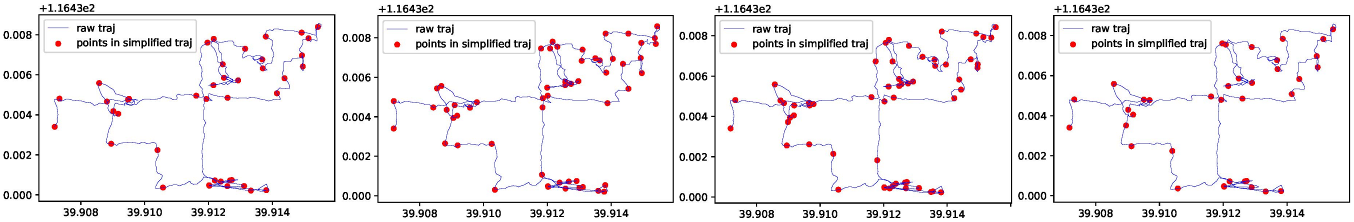
Parameter	$J = 1$	$J = 2$	$J = 3$	$J = 4$	$J = 5$
Ratio (SED)	11.857%	11.864%	13.047%	13.379%	13.390%
Time (s)	0.89	0.81	0.78	0.68	0.64
Pruning	0.0%	12.941%	25.313%	29.851%	33.488%

[4] https://github.com/zhengwang125/EB-OTS



(a) SED Error (b) PED Error (c) DAD Error (d) SAD Error (e) Time (SED)

Figure 8: Comparison with the exact algorithm SP (Geolife).



(a) BOPW (ratio = 6.9%) (b) MOPW (ratio = 9.4%) (c) CISED-S (ratio = 9.7%) (d) MARL4TS (ratio = 6.1%)

Figure 9: Case Study (Geolife).

Table 5: Impacts of parameter K for MARL4TS (Geolife).

Parameter	$K = 1$	$K = 2$	$K = 3$	$K = 4$	$K = 5$	$K = 6$	$K = 7$	$K = 8$	$K = 9$	$K = 10$
Ratio (SED)	13.386%	13.385%	13.260%	11.869%	11.864%	11.863%	11.857%	11.857%	11.855%	11.854%
Time (s)	0.59	0.65	0.76	0.83	0.90	0.96	1.06	1.11	1.17	1.31

Table 6: Impacts of parameter D for MARL4TS (Geolife).

Parameter	$D = 0$	$D = 1$	$D = 2$	$D = 3$	$D = 4$	$D = 5$	$D = 6$	$D = 7$	$D = 8$	$D = 9$
Ratio (SED)	13.446%	12.055%	11.987%	11.958%	11.888%	11.864%	11.855%	11.834%	11.805%	11.804%
Time (s)	0.51	0.65	0.67	0.84	0.87	0.88	0.96	1.13	1.17	1.31

Table 7: Impacts of parameter C for MARL4TS (Geolife).

Parameter	$C = 10$	$C = 20$	$C = 30$	$C = 40$	$C = 50$	$C = 60$	$C = 70$	$C = 80$	$C = 90$	$C = 100$
Ratio (SED)	12.023%	11.901%	11.880%	11.867%	11.864%	11.862%	11.861%	11.860%	11.860%	11.860%
Time (s)	0.42	0.47	0.48	0.53	0.59	0.61	0.63	0.69	0.71	0.74