Trajectory Simplification with Reinforcement Learning

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

***Abstract*—Trajectory data is used in various applications including traffic analysis, logistics, and mobility services. It is usually collected continuously by sensors and accumulated at a server resulting in big volume. A common practice is to conduct trajectory simplification which is to drop some points of a trajectory when they are being collected (online mode) and/or after they are accumulated (batch mode). Existing algorithms usually involve some decision making tasks (e.g., deciding which point to drop), for which, some human-crafted rules are used. In this paper, we propose to learn a policy for the decision making tasks via reinforcement learning (RL) and develop trajectory simplification methods based on the learned policy. Compared with existing algorithms, our RL-based methods are data-driven and can adapt to different dynamics underlying the problem. We conduct extensive experiments to verify that our RL-based methods compute simplified trajectories with smaller errors while running comparably fast (and faster in the batch mode) compared with existing methods.**

***Index Terms*—trajectory data, trajectory simplification, reinforcement learning**

I. INTRODUCTION

Trajectory data is a data type that captures traces of moving objects such as vehicles, pedestrians, robots, etc. It is central to many applications such as urban mobility analysis, logistics, transportation, sports games, etc. Trajectory data is typically generated continuously and collected by remote sensors such as GPS devices. One typical scenario is that a sensor periodically checks the coordinates and time, which corresponds to a time-stamped location (called spatio-temporal point or simply point), and stores the point in a buffer. Once the buffer is full, the sensor flushes the buffer by transmitting all those points in the buffer to a server via network and then continues the process of recording points. Typically, a sensor has a small storage budget, low computation capability, and limited network bandwidth. A consequent issue is that the buffer would become occupied frequently and the workload of transmitting the points is high. In addition, in some applications, there could be hundreds of thousands of sensors, which collect trajectory data simultaneously. Once the trajectory data collected by all these sensors is accumulated at a server, the volume would be huge, which has been illustrated by several existing studies (see the survey paper [1]). A consequent issue is that the huge volume of trajectory data would increase the storage cost and more importantly make the query processing on the data expensive.

A common practice that has been used to deal with the aforementioned two issues is to conduct *trajectory simplification*, which essentially is to drop some points of a given trajectory and keep the remaining ones as a simplified trajectory. When it is conducted at a sensor's side, we say it is in an *online* mode, and when it is conducted at a server's side, we say it is in a *batch* mode. Specifically, in the online mode, the trajectory data is inputted point by point and once a point is dropped, it is no longer accessible. In the batch mode, the trajectory data is inputted completely once and remains accessible during the whole course of trajectory simplification. The rationale behind trajectory simplification is two-fold. First, not all points of a trajectory carry equal amount of information and some carry little or even no information. For example, when an object moves along a straight line at a constant speed, all points except for the first and last ones carry no information and could be dropped. Second, by dropping those points carrying little or no information, the burden on the transmission, storage, and query processing would be lowered down significantly. In this paper, we consider a problem of trajectory simplification, which is to drop at least a certain number of points (or equivalently to keep at most a certain number of points) such that the information loss, captured as the "error" of the simplified trajectory, is minimized. We call this problem *Min-Error*.

Quite a few algorithms exist for the Min-Error problem in the online mode, including STTrace [2], SQUISH [3], and SUISH-E [4]. All these algorithms share the idea that it maintains a buffer of a certain size and keeps storing points in the buffer, and whenever the buffer becomes full, it picks one point from the buffer and drops it. In addition, they all make the decision on which point to drop by defining some "importance value" of each point and always dropping the point with the least importance value. Roughly, the importance value of a point is defined as some form of error that would be introduced when the point is dropped, i.e., the smaller the error is, the smaller the importance value is, meaning the point is less important and should be dropped. When a point is dropped, the importance values of the remaining points should be updated, in which the existing algorithms differ from one another. STTrace simply re-computes the values, while SQUISH distributes the value of a point that has been dropped to its neighboring points (so that the importance values would be carried on) and SQUISH-E is similar to SQUISH with only some slight refinements on the update procedure. As could be noticed, these algorithms are mainly based on some human-crafted rules for deciding which point to drop.

In this paper, we propose a reinforcement learning (RL) method for the trajectory simplification problem in the online mode. We treat the trajectory simplification problem (in the online mode) as one of a sequential decision process, i.e., it scans a trajectory sequentially, and whenever the buffer is full, it makes a decision on which point to drop. We then model the process as a *Markov decision process* (MDP), learn the policy for the MDP via a widely-used *policy gradient* method [5]–[7], and develop a trajectory simplification method based on the learned policy. We call this method *RLTS*. Compared with existing algorithms, our RLTS method computes simplified trajectories with smaller errors, which is mainly due to its data-driven nature and also its capability of adapting to different dynamics of the inputted points. Same as the existing algorithms STTrace, SQUISH, and SQUISH-E, the RLTS method has the time complexity of $O((n-W)\log W)$, where n is the number of points in the inputted trajectory and W is the buffer size.

We also propose a variant of RLTS, called RLTS-Skip, by augmenting the MDP with additional actions of *skipping* points from being scanned. The rationale is that some points may carry little information so that they can be dropped immediately without being inserted in the buffer when they are being scanned, as RLTS does. The benefit is that the efforts of deciding and taking actions for these points that are skipped are saved and the efficiency is boosted. In addition, RLTS-Skip accepts a parameter J, which controls the maximum number of points that are allowed to be skipped. Therefore, RLTS-Skip provides a tunable trade-off between efficiency and effectiveness with different settings of J. Note that when $J = 0$, RLTS-Skip reduces to RLTS.

The RLTS algorithm (and also the RLTS-Skip algorithm) could be immediately applied to the Min-Error problem in the batch mode since in the batch mode, all points are accessible throughout the simplification process and we can simply scan the points from one end to the other. While this would work without any adaption, it does not fully unleash the power of a RL-based method. Recall in the batch mode, all points are accessible during the process, but with the RLTS and RLTS-Skip methods directly applied to the batch mode, all points that have been dropped before are ignored (note that it has to be this case in the online mode since the points that have been dropped are no longer accessible). Motivated by this, we adapt the RLTS and RLTS-Skip methods to the batch mode by redefining the states of the MDP involved such that all points but not only those points remained in the buffer are utilized so as to enrich the information that a state captures for the environment and finally make better actions based on the redefined states. We call the adapted methods RLTS and RLTS-Skip as well. Consequently, the method would have higher time complexity than for the online mode, but still comparable to those of existing algorithms for the batch mode. The RLTS method for the batch mode has the time complexity of $O((n-W)(n'+\log W))$, where n is the number of points in the inputted trajectory, W is the buffer size, and n' is a variable much smaller than n in practice.

In summary, our main contribution is as follows. We propose a reinforcement learning (RL)-based method called RLTS for trajectory simplification, which is the first of its kind. RLTS is data-driven and has the ability to adapt to different dynamics of underlying points. In addition, RLTS is generic and works for multiple error measurements while many existing algorithms work for some specific ones only. Furthermore, we propose a variant of RLTS, i.e., RLTS-Skip, which runs faster than RLTS and provides a controllable trade-off between the effectiveness and efficiency. Extensive experiments on real-life datasets demonstrate that our methods have better effectiveness and comparable (or better in the batch mode) efficiency compared with existing algorithms.

The rest of paper is organized as follows. We review the literature in Section II and give the problem definition in Section III. We introduce our RLTS and RLTS-Skip algorithms for the online and batch modes in Section IV and in Section V, respectively. We present our experimental results in Section VI and finally conclude our paper in Section VII.

II. Related Work

A. *Trajectory Simplification in Online Mode*

In the online mode, streaming trajectory data is continuously collected by sensors and stored in a local buffer. The trajectory simplification problem is to decide which points to be dropped and correspondingly which to be kept in the buffer and sent to the server's side later on. Among the existing studies of trajectory simplification in the online mode, [2]–[4] target the Min-Error problem, which aims to minimize the error given some storage budget. In [2], [8], the STTrace algorithm is proposed, which processes incoming points one by one and for each one, it first decides whether to drop it. If so, it moves to the next one; and if not, it drops one existing point in the buffer, and then inserts the point that is being processed to the buffer. The decision making is based on some heuristic values defined for the points. In [3], the authors propose the SQUISH algorithm and in [4] they propose an enhanced version of SQUISH called SQUISH-E. SQUISH and SQUISH-E follow the framework of STTrace, but use different definitions for the heuristic values of points. Each of these algorithms has the time complexity of $O((n-W)\log W)$. Our solution differs from these methods in that it is based on a policy learned via reinforcement learning instead of human-crafted heuristic values for decision making.

Other proposals on the online mode trajectory simplification do not target the Min-Error problem and are reviewed as follows. In [9]–[16], the authors address the problem of finding a simplified trajectory such that the size is minimized while satisfying a given error bound, i.e., a dual problem of the Min-Error problem. In [17], the "dead reckoning" technique is used for trajectory simplification, which predicts the future location of a moving object based on the assumption of a constant velocity and direction and discards a collected point if it deviates from the predicated location significantly. In [18], the authors propose to perform trajectory simplification based on some topologically persistent features, which indicate the importance of points.

B. Trajectory Simplification in Batch Mode

In the batch mode, all the points in the trajectory that is to be simplified are inputted together and remain accessible throughout the simplification process. Among those existing studies of trajectory simplification in the batch mode, [9], [19]–[21] cover the Min-Error problem, which is studied in this paper. Specifically, in [19], the authors propose a dynamic programming algorithm called *Bellman*. Bellman runs in at least cubic time, which is prohibitively expensive for large datasets. In [8], [9], [20], the authors explore different approximate algorithms for the problem, including *Top-Down* and *Bottom-Up*. Top-Down is inspired by the traditional *Douglas-Peucker* [22] algorithm and the idea is to start with two points (the first one and the last one) and then repetitively include a point that has the largest error based on some error measurement until the number of points reaches the storage budget. Top-Down has the time complexity of $O(Wn)$. Bottom-Up, as its name implies, works in a reverse way that it starts with all points of the inputted trajectory and repetitively drops a point that would introduce the smallest error based on some error measurement until the number of points left is within the storage budget. Bottom-Up has the time complexity of $O((n - W)(n' + \log n))$, where n' is bounded by n. In [21], the authors propose an approximate algorithm called *Span-Search*, which is specifically designed for the error measurement direction-aware distance (DAD). Span-Search has the time complexity of $O(cn \log^2 n)$, where c is a moderate constant. Again, these methods are mainly based on human-crafted rules, while our method is based on a learned policy.

Other trajectory simplification methods for the batch mode do not target the Min-Error problem and are reviewed as follows. In [23], [24] (and the references therein), the authors study the dual problem of Min-Error. In [25], the authors propose a solution for selecting a subset of a certain number of most representative points from a trajectory. In [26], the authors construct a reference trajectory set to support trajectory compression.

C. Road Network based Trajectory Compression

Trajectory compression [27]–[32] is a related but different problem. It aims to match the GPS points to road segments and leverage the knowledge from road networks to achieve a higher compression ratio. For example, Li et al [27] study uncertain trajectory compression and develop a framework to support probabilistic query processing in road networks. Chen et al [28] propose an online algorithm for trajectory mapping and compression utilizing vehicle heading directions upon the underlying road network. This line of research focuses on trajectory data that is generated on road networks while our work focuses movements in a free space, such as those of pedestrians, sports players, animals, etc., which are common and have been extensively studied.

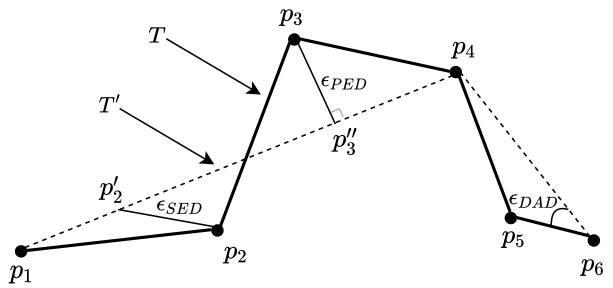


Fig. 1. A running example.

D. Reinforcement Learning

Reinforcement learning (RL) was proposed to guide agents on what actions to take in a specific environment to maximize a cumulative reward [33], where the environment is generally modeled as a Markov decision process (MDP) [34]. In recent years, RL models have been used successfully to solve algorithm problems. For example, Kong et al. [35] explore the RL methods for three classic combinatorial optimization problems. Wang et al. [36] propose an effective RL-based algorithm for the dynamic bipartite matching. Marcus et al. [37] apply RL for join order enumeration. In this paper, we model the trajectory simplification problem as an MDP and use a popular policy gradient method [5]–[7] for solving the problem. To the best of our knowledge, this is the first deep reinforcement learning based solution for trajectory simplification.

III. PRELIMINARIES AND PROBLEM STATEMENT

The trace of a moving object such as a vehicle or a user is usually captured by a trajectory, which corresponds to a sequence of time-stamped locations called *spatio-temporal points* (or simply *points*). Let $T =< p_1, p_2, ..., p_n >$ be a trajectory, where p_i is in the form of (x_i, y_i, t_i), meaning that a moving object is at location (x_i, y_i) at time t_i. We denote by $T[i : j]$ $(i \leq j)$ the subtrajectory of T, which starts form point p_i and ends at point p_j, i.e., $T[i : j] =< p_i, p_{i+1}, ..., p_j >$. We denote by $d(p_i, p_j)$ $(1 \leq i, j \leq n)$ the Euclidean distance between p_i's location and p_j's location. We define the size of the trajectory T, denoted by $|T|$, as the number of points involved in T, i.e., $|T| = n$. We denote the line segment linking location (x_i, y_i) and location (x_{i+1}, y_{i+1}) by $\overline{p_i p_{i+1}}$. It is interpreted that during the time period $[t_i, t_{i+1}]$ $(1 \leq i \leq n - 1)$, the object moves along the line segment $\overline{p_i p_{i+1}}$ at a constant speed from one end to the other and the speed is equal to $\frac{d(p_i, p_{i+1})}{t_{i+1} - t_i}$.

A. Trajectory Simplification and Error Measurements

Any trajectory resulted from T by dropping some points (that are neither the first nor the last point) corresponds to a *simplified trajectory* of T. A simplified trajectory of T, denoted by T', has the form of $T' =< p_{s_1}, p_{s_2}, ..., p_{s_m} >$ where $m \leq n$ and $1 = s_1 < s_2 < ... < s_m = n$. For example, in Figure 1, $T =< p_1, p_2, ..., p_6 >$ is a trajectory and $T' =< p_1, p_4, p_6 >$ is a simplified trajectory of T.

Consider the time period $[t_{s_j}, t_{s_{j+1}}]$ $(1 \leq j \leq m-1)$. Based on the simplified trajectory T', it is interpreted that the object moves along segment $\overline{p_{s_j} p_{s_{j+1}}}$, while based on trajectory T, it is interpreted that the object moves along a sequence of segments formed by a sequence of points $p_{s_j}, p_{s_j+1}, ..., p_{s_{j+1}}$. In other words, segment $\overline{p_{s_j} p_{s_{j+1}}}$ in T' *approximates* those segments starting at points $p_{s_j}, p_{s_j+1}, ..., p_{s_{j+1}-1}$, namely $\overline{p_{s_j} p_{s_j+1}}, \overline{p_{s_j+1} p_{s_j+2}}, ..., \overline{p_{s_{j+1}-1} p_{s_{j+1}}}$, in T. We say that segment $\overline{p_{s_j} p_{s_{j+1}}}$ is an *anchor segment* of each of the points $p_{s_j}, p_{s_j+1}, ..., p_{s_{j+1}-1}$. Note each point in trajectory T except for p_n has exactly one anchor segment in T'. For example, in Figure 1, $\overline{p_1 p_4}$ approximates a sequence of three segments $\overline{p_1 p_2}$, $\overline{p_2 p_3}$, $\overline{p_3 p_4}$ and corresponds to the anchor segment of p_1, p_2 and p_3. As could be noticed, there is a discrepancy between the interpreted movement based on the simplified trajectory T' and that based on the original trajectory T, and this discrepancy is measured as the *error* of T' wrt T, which we denote by $\epsilon(T')$.

Quite a few measurements have been proposed for defining $\epsilon(T')$, including (1) synchronized Euclidean distance (SED) [2]–[4], [9], (2) perpendicular Euclidean distance (PED) [9]–[11], [19], (3) direction-aware distance (DAD) [12], [13], [21], [24], and (4) speed-aware distance (SAD) [4]. These error measurements share the idea that (1) it defines the error of a segment $\overline{p_{s_j} p_{s_{j+1}}}$ wrt a point p_i ($s_j \leq i < s_{j+1}$) which takes $\overline{p_{s_j} p_{s_{j+1}}}$ as its anchor segment, which we denote by $\epsilon(\overline{p_{s_j} p_{s_{j+1}}} | p_i)$; (2) it defines the error of the segment $\overline{p_{s_j} p_{s_{j+1}}}$, which we denote by $\epsilon(\overline{p_{s_j} p_{s_{j+1}}})$, as the maximum among its errors wrt the points that take it as an anchor segment, i.e., $\epsilon(\overline{p_{s_j} p_{s_{j+1}}}) = \max_{s_j \leq i < s_{j+1}} \epsilon(\overline{p_{s_j} p_{s_{j+1}}} | p_i)$; (3) it then defines the error of simplified trajectory T' as the maximum error of a segment in T', i.e., $\epsilon(T') = \max_{1 \leq j \leq m-1} \epsilon(\overline{p_{s_j} p_{s_{j+1}}})$. That is, these error measurements, including SED, PED, DAD, and SAD, define $\epsilon(T')$ as follows.

$$\epsilon(T') = \max_{1 \leq j \leq m-1} \max_{s_j \leq i < s_{j+1}} \epsilon(\overline{p_{s_j} p_{s_{j+1}}} | p_i) \qquad (1)$$

Different measurements use different functions for defining $\epsilon(\overline{p_{s_j} p_{s_{j+1}}} | p_i)$ ($1 \leq j \leq m-1, s_j \leq i < s_{j+1}$), which takes the segment as its anchor segment.

- **SED** defines $\epsilon(\overline{p_{s_j} p_{s_{j+1}}} | p_i)$ as the Euclidean distance between p_i's location, i.e., (x_i, y_i), and the location at time t_i based on the movement interpreted by T', i.e., p_i's *synchronized* location based on T', which we denote by p'_i. In Figure 1, $\epsilon_{SED}(\overline{p_1 p_4} | p_2)$ corresponds to $d(p_2, p'_2)$.
- **PED** defines $\epsilon(\overline{p_{s_j} p_{s_{j+1}}} | p_i)$ as the Euclidean distance between p_i's location, i.e., (x_i, y_i), and the location on segment $\overline{p_{s_j} p_{s_{j+1}}}$, which is the *closest* from (x_i, y_i), denoted by p''_i. In Figure 1, $\epsilon_{PED}(\overline{p_1 p_4} | p_3)$ corresponds to $d(p_3, p''_3)$.
- **DAD** defines $\epsilon(\overline{p_{s_j} p_{s_{j+1}}} | p_i)$ as the *angular difference* between the direction along segment $\overline{p_i p_{i+1}}$ and that along segment $\overline{p_{s_j} p_{s_{j+1}}}$. In Figure 1, $\epsilon_{DAD}(\overline{p_4 p_6} | p_5)$ corresponds to the angle between $\overline{p_4 p_6}$ and $\overline{p_5 p_6}$.
- **SAD** defines $\epsilon(\overline{p_{s_j} p_{s_{j+1}}} | p_i)$ as the difference between the speed of segment $\overline{p_i p_{i+1}}$ and that of segment $\overline{p_{s_j} p_{s_{j+1}}}$. In Figure 1, $\epsilon_{SAD}(\overline{p_4 p_6} | p_5)$ corresponds to $|\frac{d(p_4, p_6)}{t_6 - t_4} - \frac{d(p_5, p_6)}{t_6 - t_5}|$.

B. Problem Definition

We study the trajectory simplification problem that is to find for a given trajectory, a simplified trajectory with its size bounded by a given storage budget and its error minimized. We call this problem *Min-Error*. Specifically, the Min-Error problem is defined as follows.

Problem 1 (Min-Error): Given a trajectory $T =< p_1, p_2, ..., p_n >$ and a storage budget W, which is an integer, the **Min-Error** problem is to find a simplified trajectory $T' =< p_{s_1}, p_{s_2}, ..., p_{s_m} >$ where $m \leq n$ and $1 = s_1 < s_2 < ... < s_m = n$ such that $|T'| \leq W$ and $\epsilon(T')$ is minimized, and $\epsilon(T')$ can be defined by any of those existing error measurements including SED, PED, DAD, and SAD.

The Min-Error problem has two modes, namely the online mode and the batch mode, for different application scenarios.

Online mode. In the online mode, the trajectory to be simplified is fed to the system point by point in an online fashion and those points that have been dropped during the trajectory simplification process will no longer be accessible. This mode is commonly used in applications such as remote sensing, where the sensors collect points from time to time and are constrained by storage budget, network bandwidth and energy.

Batch mode. In the batch mode, all the points in the trajectory to be simplified are fed together, and remain accessible during the simplification process. This mode is usually used at a server's side and the purpose is to reduce the storage cost (e.g., after the simplification, the original trajectory is discarded) and/or the query processing cost (e.g., it performs queries on the simplified trajectory data instead of the original data).

In this paper, we target both the online mode (Section IV) and the batch mode (Section V).

IV. Algorithms for Online Mode

In this section, by Min-Error, we mean the problem in the online mode unless specified otherwise. In the online mode, points are inputted one by one in an online fashion, while only a buffer with size W is available, i.e., at most W points can be retained throughout the trajectory simplification process. We adopt an existing strategy [2]–[4] that for the first W points, we store them in the buffer directly and for each of the remaining points, since the buffer is already full, we drop one point in the buffer to release some space and then store the new point in the buffer. Different from those existing strategies, which use some human-crafted heuristic values for deciding which point to drop when the buffer is full, we aim to achieve a more intelligent method for this decision-making task. Specifically, we treat the trajectory simplification problem as a *sequential decision making process* and model it as a *Markov decision process* (MDP) [34] (Section IV-A), use a policy gradient method [5]–[7] for learning an optimal policy for the MDP (Section IV-B), and then develop an algorithm called *RLTS*, which uses the learned policy for the Min-Error problem (Section IV-C). In Section IV-D, we present a variant of RLTS, called *RLTS-Skip*, which boosts the efficiency of RLTS via skipping some points from being scanned.

A. Min-Error Modeled as an MDP

We model the Min-Error problem as an MDP, which consists of four components, namely *states*, *actions*, *transitions*, and *rewards* as defined below.

1) States: Consider a situation where there are W points $p_{s_1}, p_{s_2}, ..., p_{s_W}$ in the buffer and a newly inputted point p_i ($i > W$) is to be inserted into the buffer next. The task is to drop one point from the buffer and then insert the point p_i into the buffer. Conceptually, it is equivalent to the process that we first append the point p_i to the buffer, i.e., the buffer becomes $p_{s_1}, p_{s_2}, ..., p_{s_W}, p_{s_{W+1}}$, involving $(W+1)$ points, and then we drop one point p_{s_j} ($2 \leq j \leq W$) from the buffer. Note that by the definition of trajectory simplification, we are not allowed to drop point p_{s_1}, i.e., p_1. An intuitive idea is to drop one of those points such that the error that is introduced as a consequence of the dropping operation is small. If we drop the point p_{s_j} ($2 \leq j \leq W$), two existing segments $\overline{p_{s_{j-1}} p_{s_j}}$ and $\overline{p_{s_j} p_{s_{j+1}}}$ would be destroyed, one new segment $\overline{p_{s_{j-1}} p_{s_{j+1}}}$ would be created, and other segments are unchanged. Since the error of a simplified trajectory is determined by those of its segments and the error of a segment is further determined by its errors wrt the points in the original trajectory, which take the segment as their anchor segments, the error of the newly created segment $\overline{p_{s_{j-1}} p_{s_{j+1}}}$ wrt the point p_{s_j}, i.e., $\epsilon(\overline{p_{s_{j-1}} p_{s_{j+1}}} | p_{s_j})$, captures the consequence of dropping p_{s_j} well.

Motivated by this, we define for each point p_{s_j} ($2 \leq j \leq W$), a *value*, denoted by $v(p_{s_j})$, as follows.

$$v(p_{s_j}) := \epsilon(\overline{p_{s_{j-1}} p_{s_{j+1}}} | p_{s_j}) \tag{2}$$

A lower value means that once the point is dropped, the introduced error tends to be smaller and thus it should be dropped with a higher chance. Then, we define the state of the situation, which we denote by s, based on the values of the points in the buffer. An immediate idea is to incorporate the values of all $(W-1)$ points p_{s_j} ($2 \leq j \leq W$) in the buffer for defining the state. However, this definition has two issues. First, since W is an input to the problem and for different problem instances, W is usually different. With this definition, the model that is defined for one input W would not be usable for other inputs different from W. Second, W is typically a moderate to large integer, e.g., it could be in thousands. With this definition, the state space would be huge and the model be hard to train.

We propose to define the state s as the set containing k lowest values, where k ($k \leq W - 1$) is hyper-parameter that could be tuned, instead of the set containing all $(W-1)$ values. We would set the hyper-parameter k via empirical studies. Specifically, we let π denote the permutation of $s_2, ..., s_W$ such that $v(p_{\pi(1)}), v(p_{\pi(2)}), ..., v(p_{\pi(W-1)})$ is a list of the values in an ascending order. Then, we define state s formally as follows.

$$s := \{v(p_{\pi(1)}), v(p_{\pi(2)}), ..., v(p_{\pi(k)})\} \tag{3}$$

With this definition, a state is of a fixed size that is independent from the problem input. In addition, the state space is controllable via the parameter k.

2) Actions: Suppose that there are $W + 1$ points $p_{s_1}, p_{s_2}, ..., p_{s_{W+1}}$ in the buffer (conceptually) and $s = \{v(p_{\pi(1)}), v(p_{\pi(2)}), ..., v(p_{\pi(k)})\}$ is the corresponding state. Essentially, the task is to pick a point among $p_{s_2}, ..., p_{s_W}$ and drop it. An immediate idea is to define $(W - 1)$ actions, each for a point p_{s_j} ($2 \leq j \leq W$), but then there would be two issues similar to those when we discuss a straightforward method for defining a state, namely (1) the definition would be W-dependent and thus it is not flexible and (2) the action space would be large and thus the model is hard to train. In fact, it is intuitive to restrict our attention to those points with small values since dropping one of these points incurs a small consequent error. Therefore, we focus on those points with their values maintained in the state, i.e., $p_{\pi(1)}, p_{\pi(2)}, ..., p_{\pi(k)}$.

With all these, we define an action space containing k actions, each meaning to drop a point $p_{\pi(j)}$ ($1 \leq j \leq k$). Formally, we define an action, which we denote by a, as follows.

$$a := j \quad (1 \leq j \leq k) \tag{4}$$

where the action $a = j$ means that it drops the point $p_{\pi(j)}$.

3) Transitions: Suppose that there are $W + 1$ points $p_{s_1}, p_{s_2}, ..., p_{s_{W+1}}$ in the buffer (conceptually), $s = \{v(p_{\pi(1)}), v(p_{\pi(2)}), ..., v(p_{\pi(k)})\}$ is the corresponding state, and a is the action to drop point p_{s_j} ($2 \leq j \leq W$). After the action a is taken, W points are left in the buffer. When a new point p_i, is inserted, we need to compute a new state, which we denote by s', i.e., state s' would be the next state when action a is taken at state s. We update the state s to state s' as follows. Recall that a state is mainly about the values of the points in the buffer, and in order to compute the state s', we examine how the points in the buffer and their corresponding values would have changed after p_{s_j} is dropped and a new point p_i is inputted.

First, we consider the consequence of dropping p_{s_j}. After the point p_{s_j} is dropped, only two neighboring points, namely $p_{s_{j-1}}$ and $p_{s_{j+1}}$, could have their anchor segment changed. Specifically, point $p_{s_{j-1}}$'s (if $j-1 \geq 2$) anchor segment would be changed from $\overline{p_{s_{j-2}} p_{s_j}}$ to $\overline{p_{s_{j-2}} p_{s_{j+1}}}$ and point $p_{s_{j+1}}$'s (if $j + 1 \leq W - 1$) anchor segment would be changed from $\overline{p_{s_j} p_{s_{j+2}}}$ to $\overline{p_{s_{j-1}} p_{s_{j+2}}}$. Therefore, the values of the two points need to be updated, which we do as follows.

$$v(p_{s_{j-1}}) = \max\{\epsilon(\overline{p_{s_{j-2}} p_{s_{j+1}}} | p_{s_{j-1}}), \epsilon(\overline{p_{s_{j-2}} p_{s_{j+1}}} | p_{s_j})\} \tag{5}$$

$$v(p_{s_{j+1}}) = \max\{\epsilon(\overline{p_{s_{j-1}} p_{s_{j+2}}} | p_{s_{j+1}}), \epsilon(\overline{p_{s_{j-1}} p_{s_{j+2}}} | p_{s_j})\} \tag{6}$$

Here, a small trick is to include $\epsilon(\overline{p_{s_{j-2}} p_{s_{j+1}}} | p_{s_j})$ and $\epsilon(\overline{p_{s_{j-1}} p_{s_{j+2}}} | p_{s_j})$ for the updates, since it would help to capture better the error of segment $\overline{p_{s_{j-2}} p_{s_{j+1}}}$ (resp. $\overline{p_{s_{j-1}} p_{s_{j+2}}}$) once point $p_{s_{j-1}}$ (resp. $p_{s_{j+1}}$) is dropped (since both $p_{s_{j-1}}$ (resp. $p_{s_{j+1}}$) and p_{s_j} take $\overline{p_{s_{j-2}} p_{s_{j+1}}}$ (resp. $\overline{p_{s_{j-1}} p_{s_{j+2}}}$) as their anchor segment after p_{s_j} is dropped).

Second, we consider the consequence of inserting point p_i. After the point p_i is inserted, the value of point $p_{s_{W+1}}$, which is previously not defined, can be computed as follows.

$$v(p_{s_{W+1}}) = \epsilon(\overline{p_W p_i} | p_{s_{W+1}}) \tag{7}$$

The values of all other points are unchanged. Based on the $(W+1)$ points including W points that are left in the buffer and one newly inputted point p_i, which we still denote by $p_{s_1}, p_{s_2}, ..., p_{s_{W+1}}$, and their values, we compute the state s' in the same way as we compute the state s (Section IV-A1).

4) Rewards: Consider that we perform an action a at a state s and then we arrive at a new state s'. We define the reward associated with this transition from state s to state s', which we denote by r, as follows. At state s, we have W points in the buffer and a new point p_i to be inserted. We consider those points in the buffer, which constitute a trajectory that corresponds to a simplified trajectory of the trajectory fed so far, i.e., $T[1:i-1]$. We denote this simplified trajectory by T'. Similarly, at state s', we also have a simplified trajectory of $T[1:i]$, which we denote by T''. We then define the reward r as follows.

$$r = \epsilon(T') - \epsilon(T'') \tag{8}$$

where $\epsilon(T')$ is wrt $T[1:i-1]$ and $\epsilon(T'')$ is wrt $T[1:i]$. The intuition is that if the error of the simplified trajectory resulted from the action, i.e., $\epsilon(T'')$, is smaller, then the reward is larger. With this definition, it would favor those actions that lead to simplified trajectories with smaller errors. In fact, it could be verified that with this reward definition, the goal of the MDP problem, which is to maximize the accumulative rewards, is well aligned with that of the trajectory simplification problem, which is to find a simplified trajectory with the error as small as possible. To see this, suppose that we go through a sequence of states $s_1, s_2, ..., s_N$ and correspondingly, we receive a sequence of rewards $r_1, r_2, ..., r_{N-1}$. In the case that the future rewards are not discounted, we have

$$\sum_{t=1}^{N-1} r_t = \sum_{t=1}^{N-1} (\epsilon(T'_t) - \epsilon(T''_t)) = \epsilon(T'_1) - \epsilon(T''_{N-1}) = -\epsilon(T''_{N-1}) \tag{9}$$

where T'_t (resp. T''_t) is the simplified trajectory at the state s_t before (resp. after) the action a_t is performed. Note that $\epsilon(T'_1) = 0$ since at the start state, no points have been dropped and thus the error is equal to 0, and $\epsilon(T''_{N-1})$ corresponds to the error of the simplified trajectory of T.

Remarks. We note that for the computation of the reward r, which involves the computations of the errors of two simplified trajectories, is only required for the learning process. Therefore, we can use a repository of trajectories for the learning process, and once a policy has been learned, we use it for trajectory simplification. In addition, we note that we can compute $\epsilon(T'_t)$'s and $\epsilon(T''_t)$'s *incrementally* except for $\epsilon(T'_1)$ since T''_t corresponds to T'_t with one point dropped $(1 \leq t \leq N)$ and T'_{t+1} corresponds to T''_t with one point inserted $(1 \leq t \leq N-1)$.

B. Policy Learning on the MDP

The core problem of an MDP is to find an optimal *policy* for the agent, which corresponds to a function that specifies the action that the agent should choose at a specific state so as to maximize the accumulative rewards. We learn the policy for the MDP constructed for the Min-Error problem via a *policy gradient* (PNet) method, which is widely used [5]–[7]. PNet models a stochastic policy as $\pi_\theta(a|s)$, which means the probability of selecting an action a for a given state s. PNet parameterizes $\pi_\theta(a|s)$ using a neural network as follows.

$$\pi_\theta(a|s) = \sigma(\boldsymbol{W}s + \boldsymbol{b}) \tag{10}$$

where σ denotes the softmax function and $\theta = \{\boldsymbol{W}, \boldsymbol{b}\}$ denotes the parameters of the neural network. Then, PNet computes the gradients of some performance measure wrt the parameters θ as follows.

$$\nabla_\theta J(\theta) = \sum_{t=1}^{N} \frac{R_t - \overline{R}}{\sigma_R} \nabla_\theta \ln \pi_\theta(a_t|s_t) \tag{11}$$

where $s_1, s_2, ..., s_N$ is a sequence of states, $a_1, a_2, ..., a_{N-1}$ is a sequence of actions by sampling $\pi_\theta(a|s_t)$, R_t denotes the accumulative reward since action a_t is taken, $\overline{R}$ is the mean of R_t's, and σ_R is the standard deviation of R_t's. PNet repeatedly updates the parameters θ via *gradient ascent* based on the gradients computed in Equation (11) until some stopping criterion specified by users is satisfied. Note that PNet corresponds to a variant of the REINFORCE algorithm with baseline [5]–[7] and the normalization mechanism based on the mean and standard deviation helps reduce the variance of the computed gradients in Equation (11).

C. The RLTS Algorithm

Our RLTS algorithm is based on the learned policy for the MDP that models the Min-Error problem, which is presented in Algorithm 1. Specifically, for each of the first W points, it stores it in the buffer directly (Line 1 - 3). It initializes an index t for a sequence of states and actions to be traversed (Line 4); Then, for each of the following points, says, p_i, it proceeds as follows (Line 5). It computes (incrementally if possible) the values of the points in the buffer except for the first one, i.e., $v(p_{s_j})$ $(2 \leq j \leq W)$, and maintains the values in a min-priority queue with the ascending permutation denoted by π (Line 6). It then constructs a state s_t containing the set of k lowest values of points (Line 7), and samples an action a_t based on the stochastic policy that has been learned (Line 8). Based on the sampled action $a_t = j$, it drops the point with the j^{th} lowest value, i.e., $p_{\pi(j)}$ (Line 9). It then inserts the point p_i that is being processed (Line 10). Finally, it returns a simplified trajectory T' which involves all the W points in the buffer (Line 13 - 14).

Time complexity. The time complexity of the RLTS algorithm is $O((n-W)\log W)$, where n is the number of points in the input trajectory T and W denotes the storage budget of the buffer. To see this, the complexity is dominated by the part processing the last $(n-W)$ points (Line 5 - 12 in Algorithm 1), and the time cost of processing one point consists of (1) that of computing the state whose cost is $O(1)$ incrementally (Line 6); (2) that of maintaining the min-priority queue is $O(\log W)$ (Line 6); and (3) that of constructing a state, sampling an action, dropping a point, inserting a point

Algorithm 1 The RLTS algorithm

```
Require: A trajectory $T =< p_1, p_2, ..., p_n >$ which is inputted in an online fashion; A buffer with a storage budget $W$ ($W < n$);
Ensure: A simplified trajectory $T'$ of $T$ with $|T'| \leq W$;
1:  for $i = 1, 2, ..., W$ do
2:      Store point $p_i$ into the buffer;
3:  end for
4:  $t \leftarrow 1$;
5:  for $i = W + 1, W + 2, ..., n$ do
6:      Compute (incrementally if possible) the values of the points in the buffer except for the first one, i.e., $v(p_{s_j})$ ($2 \leq j \leq W$) and maintain the values in a min-priority queue with the ascending permutation denoted by $\pi$;
7:      Construct a state $s_t \leftarrow \{v(p_{\pi(1)}), v(p_{\pi(2)}), ..., v(p_{\pi(k)})\}$;
8:      Sample an action $a_t \sim \pi_\theta(a|s)$;
9:      Drop the point $p_{\pi(j)}$ from the buffer where $a_t = j$;
10:     Insert the point $p_i$ into the buffer;
11:     $t \leftarrow t + 1$;
12: end for
13: Trajectory $T' \leftarrow$ the sequence of points in the buffer;
14: Return trajectory $T'$;
```

and updating the index is $O(1)$ assuming k is a constant (Line 7 - 10). We note that this time complexity is the same as those of existing algorithms [2]–[4] and all algorithms can meet practical requirements as shown in our experiments (e.g., the time of processing one point is much less than 1ms on a moderate machine). Compared with existing algorithms, RLTS is based on a learned policy but not some human-crafted rules, and thus it could return simplified trajectories with smaller errors.

D. The RLTS-Skip Algorithm

In the RLTS algorithm, each point is inserted to the buffer for sure after it is scanned. It may then be dropped when some following points are being scanned. Consequently, when each point is being scanned, some efforts are spent on deciding which existing point in the buffer to be dropped by going through a neural network in RLTS. While this strategy gives each point a chance to be included in the buffer and thus exploring a large space of possible simplified trajectories so as to achieve good performance for the Min-Error problem, it may be too conservative. Consider a scenario where the points that are scanned most recently constitute a trajectory that indicates a movement along a straight line with a constant speed. In this scenario, we have much confidence to drop a certain number of points in a row without including them one by one to the buffer and then dropping some of them at later stages. This would help save the efforts for deciding and taking actions when scanning these points and at the same time, the effectiveness should not be affected much.

Motivated by this, we propose to augment the MDP that is defined in Section IV-A by introducing J additional actions when scanning a point p_i, namely (1) dropping p_i and continuing to scan p_{i+1}, (2) dropping p_i and p_{i+1} and continuing to scan p_{i+2}, ..., and (J) dropping p_i, p_{i+1}, ..., and p_{i+J-1} and continuing to scan p_{i+J}. Here, J is a hyper-parameter, which could be tuned. These actions essentially mean (1) skipping 1 point, (2) skipping 2 points, ..., and (J) skipping J points during the process of scanning the points of a trajectory sequentially. As a result, the augmented MDP involves $(k+J)$ actions, namely the k actions of the original MDP as defined in Section IV-A2 and the J actions as newly introduced in this section. Note that these $(k + J)$ actions are exclusive and at each state, only one of them could be taken. All other components of the original MDP remain unchanged. We call the reinforcement learning algorithm based on this augmented MDP as *RLTS-Skip*. We note that when J is set to 0, RLTS-Skip reduces to RLTS.

As could be verified, RLTS-Skip has the same time complexity as RLTS. But in practice, RLTS-Skip should have better efficiency than RLTS for two reasons: (1) At each state, RLTS-Skip can take either a "dropping" action (among the k actions) or a "skipping" action (among the J actions), RLTS can only take a "dropping" action, and the cost incurred by a "skipping" action is smaller than that by a "dropping" action (since for a "dropping" action, the values of three points need to be updated (Equation (5), (6), and (7)) while for a "skipping" action, only the value of one point need to be updated (Equation (7)); and (2) For RLTS-Skip, for those points that are skipped, the efforts for deciding and taking an action are saved.

V. Algorithms for Batch mode

In the batch mode, the trajectory to be simplified is inputted completely at the very beginning and remains accessible throughout the simplification process. Consider that we scan the point one by one from one end to the other, and then the method that has been developed for the online mode can carry over for the batch mode. While this immediate adaption would still enjoy the advantage of a learned policy, it does not make full use of the data that is available in the batch mode for better effectiveness.

Recall that in the online mode, the state of RLTS is defined based on the values of the points in the buffer and the value of a point is defined as the error of its anchor segment wrt the point only. That is, the errors of the point's anchor segment wrt other points that take the segment as an anchor segment are ignored since those points have been dropped already and are no longer accessible. In the batch mode, this is not the case, we can make use of these errors for defining the value of a point so as to capture richer information. Specifically, we define the value of a point p_{s_j} ($2 \leq j \leq W$) in the buffer as the maximum error of the p_{s_j}'s anchor segment wrt a point that takes the segment as its anchor segment as follows.

$$v(p_{s_j}) := \max_{s_{j-1} < i < s_{j+1}} \epsilon(\overline{p_{s_{j-1}} p_{s_{j+1}}} | p_i) \tag{12}$$

With this new definition of the value of a point, we define an MDP for the Min-Error problem in the batch mode similarly

TABLE I
DATASET STATISTICS.

Statistics	Geolife	T-Drive	Truck
# of trajectories	17,621	10,359	10,110
Total # of points	24,876,978	17,740,902	10,059,685
Ave. # of points per trajectory	1,412	1,713	995
Sampling rate	1s $\sim$ 5s	177s	3s $\sim$ 60s
Average distance	9.96m	623m	82.74m

as we do for RLTS in the online mode and learn the policy based on the MDP, and then use the learned policy for the trajectory simplification task. We abuse the term "RLTS" for naming the resulting method for the batch mode. Essentially, RLTS for the batch mode is exactly the same as that for the online mode except that the definition of the value of point is enhanced. We refine the state definition of the MDP for RLTS-Skip in the online mode by (1) using the new definition of the value of a point (Equation (12)); and (2) appending J values to the original k values, each corresponding to the error incurred by dropping j points $p_i, ..., p_{i+j-1}$ for $1 \leq j \leq J$, when scanning point p_i. We then develop an algorithm based on the MDP with the refined state. We abuse the term "RLTS-Skip" for naming the resulting method for the batch mode.

Time complexity. With the new definition in Equation (12), it takes $O(|s_{j+1} - s_{j-1}|)$ time to compute the value of point in the batch mode, as compared to $O(1)$ in the online mode. As a result, the time complexity of RLTS and RLTS-Skip for the batch mode becomes $O((n-W)(n' + \log W))$, where n' is the cost of computing the value of a point and is bounded by n (in practice, $n' << n$).

VI. EXPERIMENTS

A. Experimental Setup

Dataset. Our experiments are conducted on three real-world trajectory datasets, namely Geolife, T-Drive and Trucks. Geolife[1] records the outdoor trajectories of 182 users for a period of five years. T-Drive [2] tracks the trajectories of 10,357 taxis in Beijing, and Truck [3] records the GPS trajectories of 10,368 trucks in China during a period from March to October, 2015. The three datasets are widely used in evaluating trajectory simplification [15], [21], [24] including the recent dedicated evaluation work [1] and the detailed statistics are summarized in Table I.

Algorithms for Comparison. We review the trajectory simplification literature thoroughly and identify seven methods for comparison, including (1) STTrace [2], (2) SQUISH [3] and (3) SQUISH-E [4] for the online mode and (4) Bellman [19], (5) Top-Down [38], (6) Bottom-Up [20], [24], and (7) Span-Search [21] for the batch mode. We cross check these algorithms against those covered in a recent survey [1] on trajectory simplification so that all existing methods that are proposed for the Min-Error problem are included for comparison.

- STTrace [2], SQUISH [3] and SQUISH-E [4]. As discussed in Section II-A, these three methods follow a similar framework but define the heuristic values of points differently for deciding which point from a buffer to drop. Each of them has the time complexity of $O(n \log W)$.
- Bellman [19]. It is a dynamic programming algorithm and solve the Min-Error problem exactly. It has a time complexity of $O(n^3)$.
- Top-Down [38] (adaption of Douglas-Peucker [22]). It recursively splits a trajectory at a point, where the greatest error would be incurred if the trajectory is approximated with one segment until the original trajectory is splitted into $(W-1)$ parts. Then, it approximates each part with one segment and these $(W-1)$ segments constitute a simplified trajectory of the original one. It has a time complexity of $O(nW)$. Note that the Douglas-Peucker algorithm accepts an error tolerance and is usually used for the dual problem of Min-Error.
- Bottom-Up [20], [24]. It corresponds to a reverse version of Top-Down and iteratively merges two adjacent segments until the simplified trajectory reaches the storage budget. It has a time complexity of $O((n-W)(n' + \log W))$, where n' is bounded by n.
- Span-Search [21]. It is an approximate algorithm designed specifically for the DAD error measurement and has the time complexity of $O(n \log^2 n)$.

Note that we do not compare with the adaptions of the algorithms that are designed for the dual problem of Min-Error (via binary search) since these adapted algorithms would have the time complexity at least $O(n^2 \log n)$, e.g., for DAD, the time complexity is $O(n^2 C \log n)$, where $C < n$, which are clearly higher than those of RLTS and RLTS-Skip and not scalable on large datasets.

Parameter Setting and Policy Learning. The neural network used in the RLTS and RLTS-Skip methods involves one input layer, one hidden layer and one output layer, where the hidden layer involves 20 neurons and uses the tanh function as the activation function. In order to avoid the data scale issues, batch normalization provided by tensorflow is employed before the activation. For RLTS, the output layer involves k neurons and k is set as 3 by default. For RLTS-Skip, the output layer involves $(k+J)$ neurons and k and J are set as 3 and 2, respectively. We study the effects of these two hyper-parameters in Section VI-B. We randomly sample 1,000 trajectories from a training dataset, and for each trajectory, we generate 10 episodes for policy learning. Since each trajectory involves around 1,000 points, there would be about 10 million transition steps in the learning process. In addition, we use the Adam stochastic gradient descent with the learning rate of 0.001 based on empirical findings. For the reward discount factor, we tried several settings, and since the results are similar, we set it as 0.99. We take the policy, which gives the maximum reward per episode and use it for trajectory simplification. For the online mode, we sample an action with the probability outputted by the softmax function at each state,

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

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

[3]http://mashuai.buaa.edu.cn/traj.html

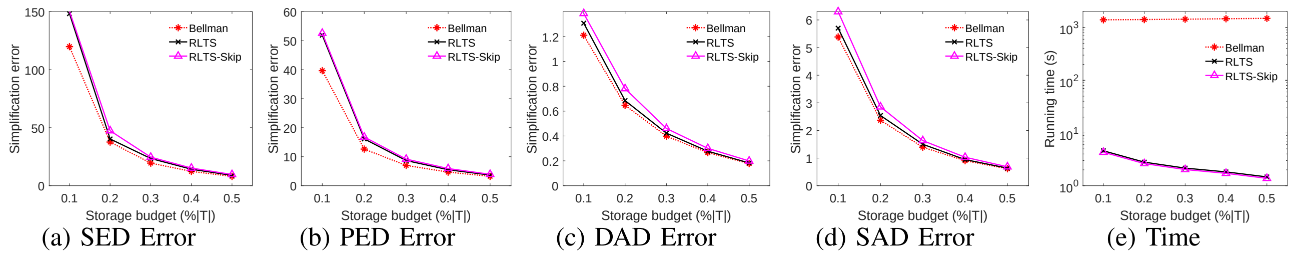


Fig. 2. Comparison with Bellman (Batch mode, Geolife).

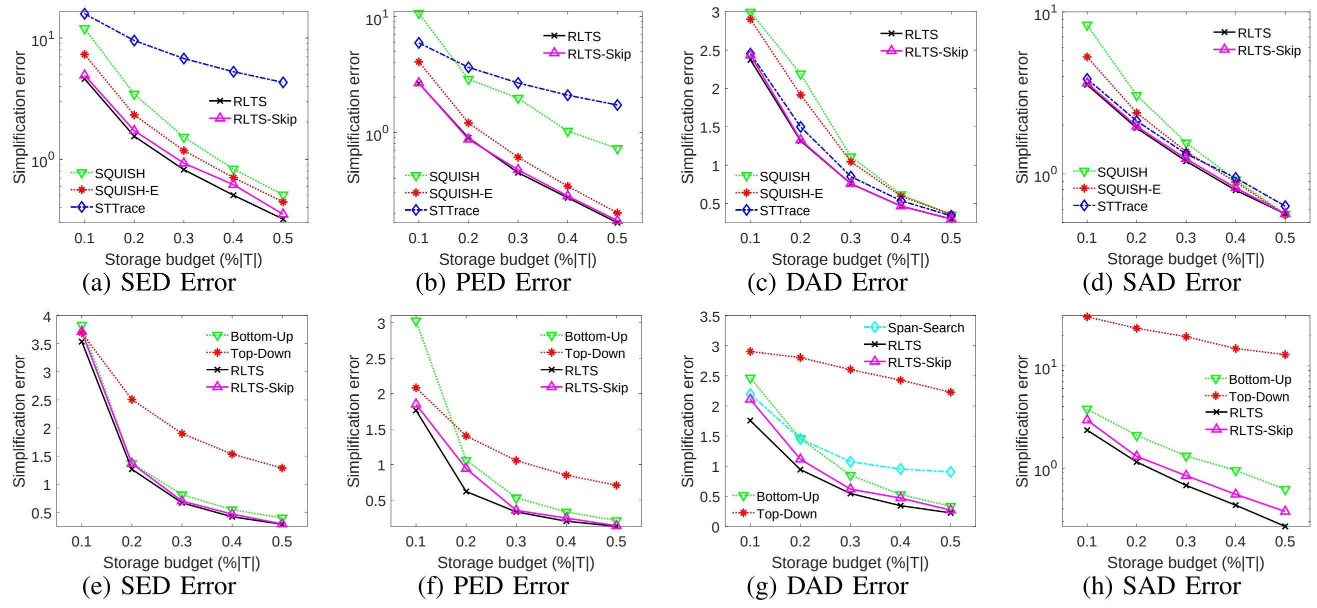


Fig. 3. Effectiveness evaluation with varying W ((a)-(d): Online mode, (e)-(h): Batch mode, Geolife).

and for the batch mode, we take the action with the maximum probability based on empirical findings.

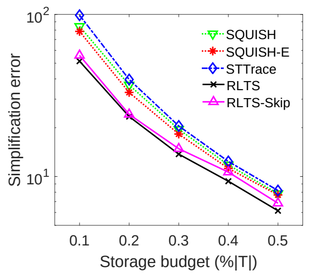
Evaluation Platform. All the methods are implemented in Python 3.6. The implementation of RLTS and RLTS-Skip is based on tensorflow 1.8.0. The experiments are conducted on a machine with Intel(R) Xeon(R) CPU E5-1620 v2 @3.70GHz 16.0GB RAM and one Nvidia GeForce GTX 1070 GPU. The codes and datasets can be downloaded via the link https://www.dropbox.com/sh/xaldzowli8pc4v6/AAA7ZMxNJAFtbrQJmN3DSInTa?dl=0.

B. Experimental Results

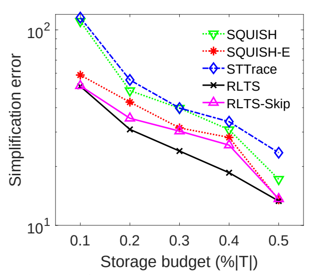
(1) Comparison with the exact algorithm Bellman (Batch mode). Since Bellman has a cubic time complexity and is slow, we compare it with RLTS and RLTS-Skip on some small datasets only, for which we randomly select a set of 100 trajectories, each with around 300 points from Geolife. We report the average simplification error and running time in Figure 2. We can observe that the simplification error of RLTS and RLTS-Skip is very close to the exact algorithm Bellman in terms of all the four measures; however, RLTS and RLTS-Skip run faster than Bellman by around three orders of magnitude. The results on the other datasets are qualitatively similar and thus omitted.

(2) Effectiveness evaluation (comparison with existing approximate algorithms). We randomly sample 1,000 trajectories T from a dataset and vary the storage budget W from $0.1 \times |T|$ to $0.5 \times |T|$ by following [21]. Figure 3 show the results for both online and batch modes on the Geolife dataset. Overall, the results clearly show that RLTS consistently outperforms existing algorithms under all error measurements for both online and batch modes and on all datasets. For RLTS-Skip, it beats all baselines for the online mode, and provides comparable performance in the batch mode.

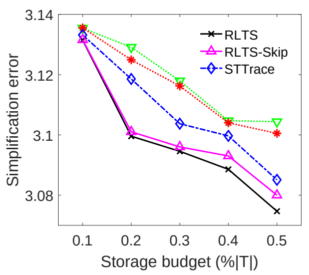
We next discuss the results in more details. (1) The online mode: Among the baselines, SQUISH-E performs the best in terms of SED and PED, and STTrace performs the best in terms of DAD and SAD. This could be because their human-crafted rules are more suitable for a particular distance measure and this finding is consistent with what are reported in the evaluation study [1]. RLTS consistently outperform all the baselines. For example, at $W = 0.3$, RLTS outperforms SQUISH-E (the second best under SED and PED) by 37% (resp. 35% and 30%) for SED on Geolife (resp. T-Drive and Truck); it outperforms STTrace (the second best under DAD and SAD) by 11% (resp. 1% and 12%) for DAD on Geolife (resp. T-Drive and Truck). In addition, the effectiveness of RLTS-Skip is slightly worse than RLTS, but still better than



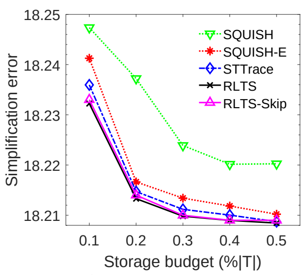
(a) SED Error



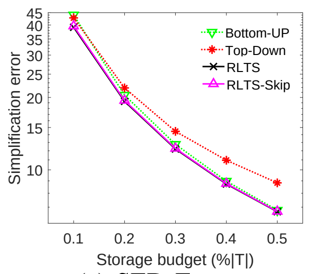
(b) PED Error



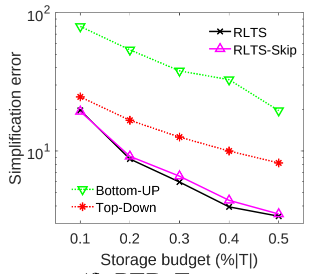
(c) DAD Error



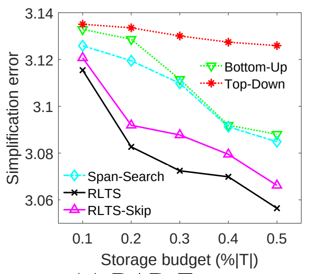
(d) SAD Error



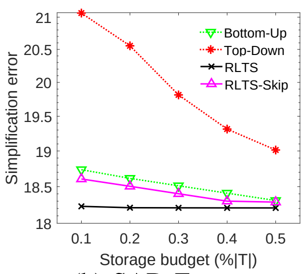
(e) SED Error



(f) PED Error

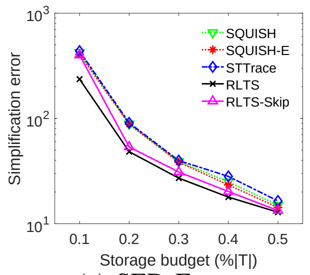


(g) DAD Error

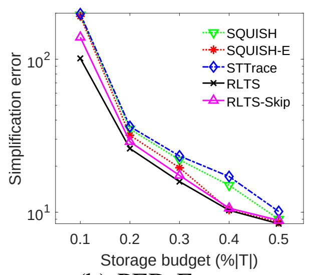


(h) SAD Error

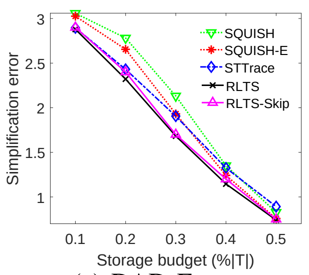
Fig. 4. Effectiveness evaluation with varying W ((a)-(d): Online mode, (e)-(h): Batch mode, T-Drive).



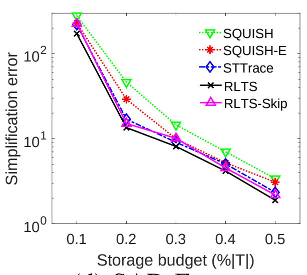
(a) SED Error



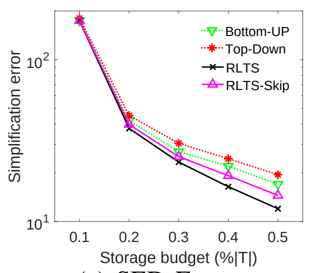
(b) PED Error



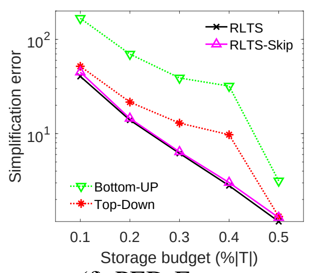
(c) DAD Error



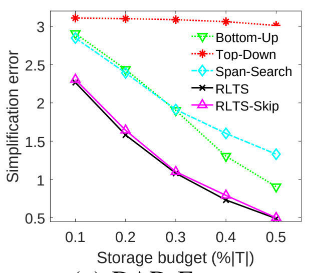
(d) SAD Error



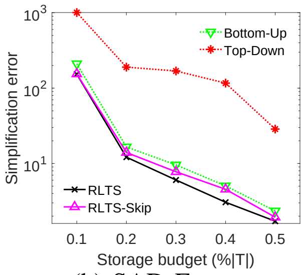
(e) SED Error



(f) PED Error



(g) DAD Error



(h) SAD Error

Fig. 5. Effectiveness evaluation with varying W ((a)-(d): Online mode, (e)-(h): Batch mode, Truck).

those baselines due to the fact that it is based on a learned policy for dropping points. (2) The batch mode: Among the baselines, Bottom-Up is the best for SED and SAD; Top-Down is mostly the best for PED especially for vehicle datasets such as T-Drive or Truck; and Bottom-UP performs similarly as Span-Search in terms of DAD. Again, RLTS consistently outperforms all the baselines. For example, at $W = 0.3$, RLTS outperforms Bottom-Up by 19% (resp. 5% and 24%) for SED on Geolife (resp. T-Drive and Truck), Top-Down by 68% (resp. 53% and 52%) for PED on Geolife (resp. T-Drive and Truck), and Span-Search by 49% (resp. 2% and 43%) for DAD on Geolife (resp. T-Drive and Truck). The reason that Bottom-Up performs better than Top-Down for most measurements could be because merging is more effective to guide trajectory simplification than splitting. Another finding is that Span-Search performs reasonably well for DAD. This is because the algorithm is specially designed for minimizing the DAD error. If we adapt it for the other three measurements, its performance becomes unsatisfactory as reported in [1], and thus we do not adapt it for the other three measurements. Similarly, RLTS-Skip has its effectiveness slightly worse than RLTS in the batch mode (due to a smaller space of simplified trajectories explored).

(3) Effectiveness evaluation (with and without the learned policy). We conduct an ablation experiment by replacing the policy with one that greedily drops a point with the smallest

TABLE II
IMPACTS OF THE LEARNED POLICY (GEOLIFE).

Dataset	Geolife				T-Drive				Truck			
Simplification Mode	Online		Batch		Online		Batch		Online		Batch	
Error Measurement	SED	PED	SED	PED	SED	PED	SED	PED	SED	PED	SED	PED
RLTS	**4.603**	**2.611**	**3.467**	**1.667**	**51.439**	**51.445**	**39.083**	**19.593**	**236.167**	**101.268**	**173.371**	**40.465**
RLTS-Greedy	8.434	3.301	3.538	1.765	71.977	104.019	39.133	19.672	372.077	149.229	173.527	40.730

TABLE III
IMPACTS OF THE PARAMETER k (GEOLIFE).

Parameter	$k=2$				$k=3$				$k=4$				$k=5$			
Mode	Online		Batch		Online		Batch		Online		Batch		Online		Batch	
Measurement	SED	PED	SED	PED	SED	PED	SED	PED	SED	PED	SED	PED	SED	PED	SED	PED
Error	4.802	2.743	3.808	1.962	**4.603**	2.611	**3.467**	**1.667**	4.779	**2.655**	4.018	1.995	4.759	2.769	4.230	2.182
Time (ms)	**0.576**	**0.456**	**0.688**	**0.547**	0.580	0.461	0.692	0.551	0.585	0.467	0.697	0.555	0.590	0.471	0.702	0.560

TABLE IV
IMPACTS OF SKIPPING STEPS J FOR RLTS-SKIP (GEOLIFE).

Mode	Online					Batch				
Skipping Steps	$J=0$	$J=1$	$J=2$	$J=3$	$J=4$	$J=0$	$J=1$	$J=2$	$J=3$	$J=4$
SED Error	4.603	5.367	5.761	6.565	9.482	3.467	3.640	3.716	4.329	4.557
Time (ms)	0.580	0.525	0.508	0.493	0.462	0.692	0.652	0.628	0.610	0.580
Skipped Pts	0%	2.7%	3.8%	5.2%	14.9%	0%	0.6%	1.9%	2.8%	3.4%

value in the buffer (called RLTS-Greedy). Table II reports the average error on 1,000 randomly sampled trajectories (W is fixed at $0.1|T|$) under the SED error. In the online mode, the learned policy contributes significantly to the effectiveness, e.g., the mean error of RLTS-Greedy is around 2 times larger than that of RLTS on Geolife. In the batch mode, the improvement due to the learned policy is not as much as in the online mode, which could be possibly explained by that in the batch mode, more information is available for making decisions and a simple greedy heuristic based on our defined state information works quite well.

(4) Effectiveness evaluation (varying the parameter k). Table III shows the results for both online and batch modes on Geolife under the SED error. As expected, as k grows, the running time becomes larger. We also observe that the accuracy becomes better as k grows from 2 to 3, and drops slightly for the batch mode when k becomes 5. This is probably because a big k would make it difficult to train the model while a small k would miss some potential candidate points. This is in line with our intuition. Therefore, we set the k at 3 in our experiments because it strikes a good balance between effectiveness and efficiency.

(5) Effectiveness evaluation (varying the parameter J). We report the effects of J, i.e., the maximum number of points that could be skipped, for both online and batch modes on Geolife under the SED error in Table IV. As expected, we observe that the general trend of RLTS-Skip is that the effectiveness degrades but the efficiency improves as J increases. This is because with a larger J, RLTS-Skip would tend to skip more points, which would lead to (1) the space of possible simplified trajectories would be smaller and (2) the efforts of deciding and taking actions would be less. We also report the portion of skipped points in the 1000 random trajectories. We choose $J=2$ as the default setting for other experiments because it gives a reasonable trade-off between effectiveness and efficiency.

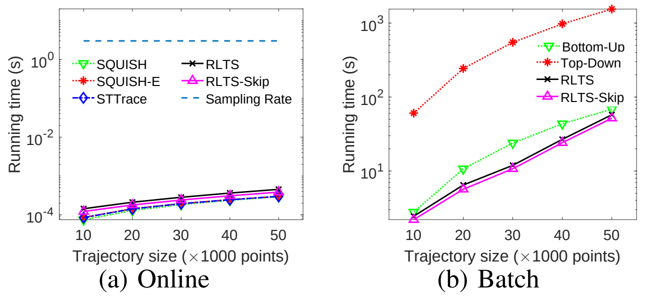


Fig. 6. Efficiency evaluation (varying $|T|$) on Truck.

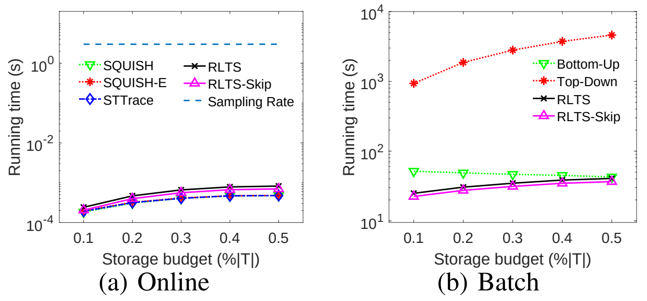


Fig. 7. Efficiency evaluation (varying W) on Truck.

(6) Efficiency evaluation (varying the trajectory length $|T|$). We follow [1], [21] by varying $|T|$ from 10,000 to 50,000. For each setting, we randomly select 100 trajectories with the size around the setting from Truck. We fix W at $0.1|T|$. Figure 6 shows the results for both online and batch modes under SED. In the online mode, we show the average running time per point because the processing time of a single point is important for an online scenario. In addition, the sampling rate (3s) of the Truck dataset is shown with a dotted line in the figure for reference. We observe that RLTS and RLTS-Skip are slightly slower than the three baseline algorithms though

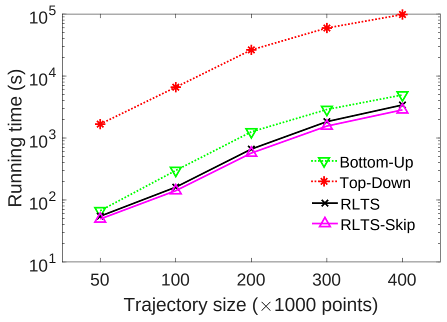


Fig. 8. Scalability test on Truck.

all of them have the same time complexity. This is because learning-based algorithms employ the learning models to make the decision (i.e., dropping or skipping) while the other three algorithms use a simple comparison operation for the same task. In addition, RLTS-Skip runs faster than RLTS since the time cost of constructing states and choosing actions are saved for those points that have been skipped. Overall, RLTS and RLTS-Skip are fast enough (very close to other algorithms) and far meets the practical needs, e.g., for a trajectory with about 10,000 points, they take less than 0.15ms per point, which is 20,000 times faster than the sampling rate (3s). In the batch mode, both RLTS and RLTS-Skip are faster than Top-Down and Bottom-Up, and the gaps of efficiency are aligned with the time complexities. We omit the running time of Span-Search because it has been shown to be slower than Top-Down in the existing studies [1], [21]. The results on the other datasets under other error measurements are qualitatively similar as those reported in Figure 6 and are omitted.

(7) Scalability test. Figure 8 shows the scalability test results on some long trajectories on Truck. We vary trajectory size from around 50,000 to 400,000 and report the average running time per trajectory. According to the results, we observe our RL-based methods are scalable. e.g., for the longest trajectory with about 383,000 points, RLTS-Skip takes 2,843s while Bottom-Up and Top-Down take 4,952s, 98,427s, respectively.

(8) Efficiency evaluation (varying the budget size W). We vary W from $0.1|T|$ to $0.5|T|$ on Truck and fix $|T|$ at 40,000 using SED as the error measurement. Figure 7 shows the result for both the online and batch modes. Similarly, in the online mode, RLTS and RLTS-Skip are a bit slower than SQUISH, SQUISH-E and STTrace, but run reasonably fast. For example, they take less than 1ms per point point for a trajectory with 40,000 points. The running times of all the methods slightly increase with W. In the batch mode, both RLTS and RLTS-Skip run faster than Top-Down by around two orders of magnitude. They also run faster than Bottom-Up, and gap reduces as W increases since they take $O(\log W)$ time to construct states while Bottom-Up takes $O(\log n)$ time to decide which segments to merge.

(9) Case study. In Figure 9, blue solid lines indicate a raw trajectory and red dashed lines indicate its simplified trajectories by different algorithms in the online mode. We label the SED errors with black dashed lines. The results clearly show that our RL-based methods return better results than baselines. For example, the SED of RLTS ($\epsilon = 2.851$) is around half of those of SQUISH and SQUISH-E ($\epsilon = 5.987$), and STTrace ($\epsilon = 5.866$). In addition, we notice the results in the batch mode are generally better than the online mode. This is because the whole trajectory is accessible during the simplification process in the batch mode instead of point by point in an online fashion. Again, RL-based methods RLTS and RLTS-Skip consistently outperform other baselines.

Summary on the experimental results. Regarding the effectiveness, RLTS and RLTS-Skip are better than existing approximate algorithms consistently on both online and batch modes, for all four error measurements, on all three datasets. Regarding the efficiency, RLTS and RLTS-Skip both run comparably fast as other algorithms for the online mode and consistently faster for the batch mode. For RLTS and RLTS-Skip, the former has slightly better effectiveness while the latter has better efficiency.

VII. Conclusion

In this paper, we study the trajectory simplification problem in both online and batch modes. We propose a reinforcement learning (RL)-based method called RLTS for both modes. Compared with existing algorithms, which are mainly heuristic-based, our RLTS method is data-driven and can adapt to different dynamics of the underlying points. We conduct extensive experiments, which show that RLTS computes simplified trajectories with consistently lower errors and runs comparably fast in the online mode and faster in the batch mode, compared with existing algorithms. One interesting direction for future research is to explore reinforcement learning methods for other related problems that could be modeled as a sequential decision process such as trajectory segmentation.

References

[1] D. Zhang, M. Ding, D. Yang, Y. Liu, J. Fan, and H. T. Shen, "Trajectory simplification: an experimental study and quality analysis," *Proceedings of the VLDB Endowment*, vol. 11, no. 9, pp. 934–946, 2018.

[2] M. Potamias, K. Patroumpas, and T. Sellis, "Sampling trajectory streams with spatiotemporal criteria," in *18th International Conference on Scientific and Statistical Database Management (SSDBM'06)*. IEEE, 2006, pp. 275–284.

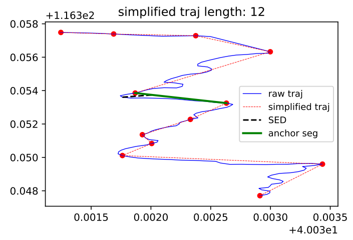
[3] J. Muckell, J.-H. Hwang, V. Patil, C. T. Lawson, F. Ping, and S. Ravi, "Squish: an online approach for gps trajectory compression," in *Proceedings of the 2nd International Conference on Computing for Geospatial Research & Applications*. ACM, 2011, p. 13.

[4] J. Muckell, P. W. Olsen, J.-H. Hwang, C. T. Lawson, and S. Ravi, "Compression of trajectory data: a comprehensive evaluation and new approach," *GeoInformatica*, vol. 18, no. 3, pp. 435–460, 2014.

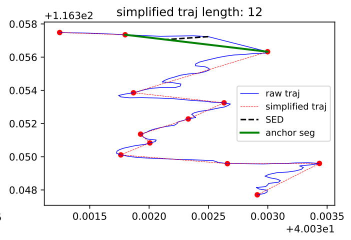
[5] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[6] A. Ilyas, L. Engstrom, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry, "A closer look at deep policy gradients," *International Conference on Learning Representations*, 2020.

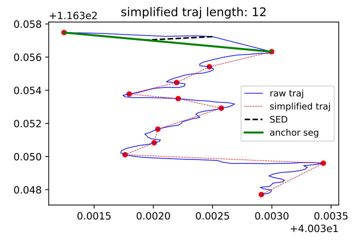
[7] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.



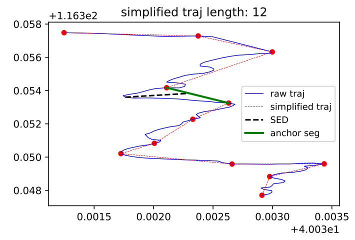
(a) RLTS ($\epsilon = 2.851$)



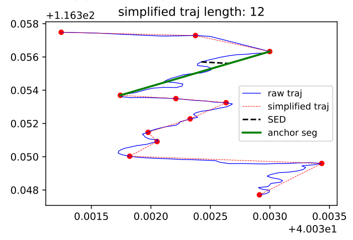
(b) RLTS-Skip ($\epsilon = 3.987$)



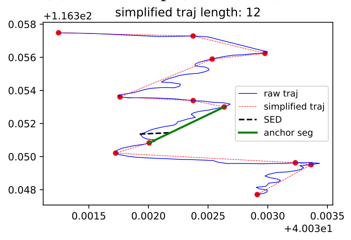
(c) SQUISH, SQUISH-E ($\epsilon = 5.987$)



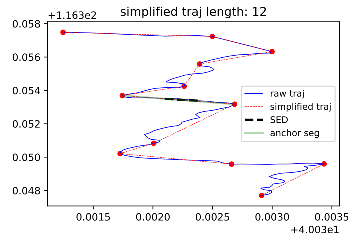
(d) STTrace ($\epsilon = 5.866$)



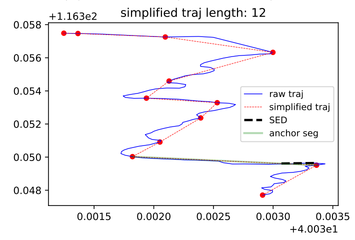
(e) RLTS ($\epsilon = 2.508$)



(f) RLTS-Skip ($\epsilon = 2.700$)



(g) Bottom-UP ($\epsilon = 3.009$)



(h) Top-Down ($\epsilon = 2.853$)

Fig. 9. Case Study ((a)-(d): Online mode, (e)-(h): Batch mode, Geolife).

[8] H. Li, L. Kulik, and K. Ramamohanarao, "Spatio-temporal trajectory simplification for inferring travel paths," in *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2014, pp. 63–72.

[9] N. Meratnia and A. Rolf, "Spatiotemporal compression techniques for moving point objects," in *International Conference on Extending Database Technology*. Springer, 2004, pp. 765–782.

[10] J. Liu, K. Zhao, P. Sommer, S. Shang, B. Kusy, and R. Jurdak, "Bounded quadrant system: Error-bounded trajectory compression on the go," in *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 2015, pp. 987–998.

[11] J. Liu, K. Zhao, P. Sommer, S. Shang, B. Kusy, J.-G. Lee, and R. Jurdak, "A novel framework for online amnesic trajectory compression in resource-constrained environments," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 11, pp. 2827–2841, 2016.

[12] B. Ke, J. Shao, Y. Zhang, D. Zhang, and Y. Yang, "An online approach for direction-based trajectory compression with error bound guarantee," in *Asia-Pacific Web Conference*. Springer, 2016, pp. 79–91.

[13] B. Ke, J. Shao, and D. Zhang, "An efficient online approach for direction-preserving trajectory simplification with interval bounds," in *2017 18th IEEE International Conference on Mobile Data Management (MDM)*. IEEE, 2017, pp. 50–55.

[14] W. Cao and Y. Li, "Dots: An online and near-optimal trajectory simplification algorithm," *Journal of Systems and Software*, vol. 126, pp. 34–44, 2017.

[15] X. Lin, S. Ma, H. Zhang, T. Wo, and J. Huai, "One-pass error bounded trajectory simplification," *Proceedings of the VLDB Endowment*, vol. 10, no. 7, pp. 841–852, 2017.

[16] X. Lin, J. Jiang, S. Ma, Y. Zuo, and C. Hu, "One-pass trajectory simplification using the synchronous euclidean distance," *The VLDB Journal*, vol. 28, no. 6, pp. 897–921, 2019.

[17] G. Trajcevski, H. Cao, P. Scheuermanny, O. Wolfsonz, and D. Vaccaro, "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, 2006, pp. 19–26.

[18] P. Katsikouli, R. Sarkar, and J. Gao, "Persistence based online signal and trajectory simplification for mobile devices," in *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2014, pp. 371–380.

[19] R. Bellman, "On the approximation of curves by line segments using dynamic programming," *Communications of the ACM*, vol. 4, no. 6, p. 284, 1961.

[20] E. Keogh, S. Chu, D. Hart, and M. Pazzani, "An online algorithm for segmenting time series," in *Proceedings 2001 IEEE international conference on data mining*. IEEE, 2001, pp. 289–296.

[21] C. Long, R. C.-W. Wong, and H. Jagadish, "Trajectory simplification: on minimizing the direction-based error," *Proceedings of the VLDB Endowment*, vol. 8, no. 1, pp. 49–60, 2014.

[22] D. H. Douglas and T. K. Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature," *Cartographica: the international journal for geographic information and geovisualization*, vol. 10, no. 2, pp. 112–122, 1973.

[23] M. Chen, M. Xu, and P. Franti, "A fast $o(n)$ multiresolution polygonal approximation algorithm for gps trajectory simplification," *IEEE Transactions on Image Processing*, vol. 21, no. 5, pp. 2770–2785, 2012.

[24] C. Long, R. C.-W. Wong, and H. Jagadish, "Direction-preserving trajectory simplification," *Proceedings of the VLDB Endowment*, vol. 6, no. 10, pp. 949–960, 2013.

[25] Y. Chen, K. Jiang, Y. Zheng, C. Li, and N. Yu, "Trajectory simplification method for location-based social networking services," in *Proceedings of the 2009 International Workshop on Location Based Social Networks*, 2009, pp. 33–40.

[26] Y. Zhao, S. Shang, Y. Wang, B. Zheng, Q. V. H. Nguyen, and K. Zheng, "Rest: A reference-based framework for spatio-temporal trajectory compression," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2018, pp. 2797–2806.

[27] T. Li, R. Huang, L. Chen, C. S. Jensen, and T. B. Pedersen, "Compression of uncertain trajectories in road networks," *Proceedings of the VLDB Endowment*, vol. 13, no. 7, pp. 1050–1063, 2020.

[28] C. Chen, Y. Ding, X. Xie, S. Zhang, Z. Wang, and L. Feng, "Trajcompressor: An online map-matching-based trajectory compression framework leveraging vehicle heading direction and change," *IEEE Transactions on Intelligent Transportation Systems*, vol. 21, no. 5, pp. 2012–2028, 2019.

[29] Y. Han, W. Sun, and B. Zheng, "Compress: A comprehensive framework of trajectory compression in road networks," *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 2, pp. 1–49, 2017.

[30] X. Yang, B. Wang, K. Yang, C. Liu, and B. Zheng, "A novel representation and compression for queries on trajectories in road networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 4, pp. 613–629, 2017.

[31] R. Song, W. Sun, B. Zheng, and Y. Zheng, "Press: A novel framework of trajectory compression in road networks," *Proceedings of the VLDB Endowment*, vol. 7, no. 9, p. 661–672, 2014.

[32] S. Koide, Y. Tadokoro, C. Xiao, and Y. Ishikawa, "Cinct: Compression and retrieval for massive vehicular trajectories via relative movement labeling," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 1097–1108.

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

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

[35] W. Kong, C. Liaw, A. Mehta, and D. Sivakumar, “A new dog learns old tricks: Rl finds classic optimization algorithms,” 2018.

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

[37] R. Marcus and O. Papaemmanouil, “Deep reinforcement learning for join order enumeration,” in *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. ACM, 2018, p. 3.

[38] J. E. Hershberger and J. Snoeyink, *Speeding up the Douglas-Peucker line-simplification algorithm*. University of British Columbia, Department of Computer Science, 1992.