A Framework for Efficient and Convenient Evaluation of Trajectory Compression Algorithms

Jonathan Muckell
Department of Informatics
University at Albany–SUNY
Albany, NY 12222, USA
jonmuckell@gmail.com

Paul W. Olsen Jr,
Jeong-Hyon Hwang, and S. S. Ravi
Department of Computer Science
University at Albany–SUNY
Albany, NY 12222, USA
{polsen, jhh, ravi}@cs.albany.edu

Catherine T. Lawson
Department of Geography and Planning
University at Albany–SUNY
Albany, NY 12222, USA
lawsonc@albany.edu

Abstract—**Trajectory compression algorithms eliminate redundant information in the history of a moving object. Such compression enables efficient transmission, storage, and processing of trajectory data. Although a number of compression algorithms have been proposed in the literature, no common benchmarking platform for evaluating their effectiveness exists. This paper presents a benchmarking framework for efficiently, conveniently, and accurately comparing trajectory compression algorithms. This framework supports various compression algorithms and metrics defined in the literature, as well as three synthetic trajectory generators that have different trade-offs. It also has a highly extensible architecture that facilitates the incorporation of new compression algorithms, evaluation metrics, and trajectory data generators. This paper provides a comprehensive overview of trajectory compression algorithms, evaluation metrics and data generators in conjunction with detailed discussions on their unique benefits and relevant application scenarios. Furthermore, this paper describes challenges that arise in the design and implementation of the above framework and our approaches to tackling these challenges. Finally, this paper presents evaluation results that demonstrate the utility of the benchmarking framework.**

I. Introduction

The availability of GPS-equipped mobile devices and development of geo-spatial applications have led to rapid adoption of location-based services. In the United States, 74% of smart phone users regularly use thier phones to get location-based information [1]. In particular, applications such as Four Square have seen a dramatic rise in usage, from 4% of the smart-phone users in 2011 to 10% in 2012. Many geo-spatial applications have difficulties in managing large volumes of GPS trajectory data. Common problems are long network transmission time, inefficient processing of trajectory data, and wasted memory and disk space.

To mitigate the above issues, a variety of trajectory compression algorithms have been developed [2], [3], [4], [5], [6], [7], [8]. Given an input trajectory, these algorithms produce an output trajectory that consists of a subset of the points from the input trajectory. These "lossy" compression algorithms can trade the accuracy of the output trajectory for a greater compression ratio or a shorter compression time. Since these algorithms have different trade-offs in terms of compression time, compression ratio (i.e., the size of the original trajectory divided by the size of the compressed representation of the trajectory), or diverse error metrics, it is difficult to select the most appropriate algorithm in practical situations. Furthermore, a standardized evaluation benchmark for comparing trajectory compression algorithms does not exist.

This paper presents a new benchmarking framework for efficiently, conveniently, and accurately comparing trajectory compression algorithms. Designing and implementing such a framework raises new challenges. First, the framework must support a wide spectrum of trajectory compression algorithms, metrics for evaluating these algorithms, and tools which facilitate the incorporation of newly developed or extended algorithms and metrics. Second, the framework must be able to provide a large number of trajectories that capture various modes of transportation as well as extreme conditions (e.g., drastic changes in speed and direction) for measuring the reliability of compression algorithms. Third, the framework should allow users to conveniently specify compression algorithms, evaluation metrics, and trajectory types of interest, and analyze the evaluation results. Fourth, the framework must be able to accurately evaluate compression algorithms on a fair basis despite their inherent differences (e.g., some algorithms strive to maximize compression ratio under a certain accuracy constraint whereas others aim at minimizing one type of error while guaranteeing a specific compression ratio). Fifth, the framework needs to efficiently support large-scale evaluations by taking advantage of the collective capability of a server cluster.

Our benchmarking framework meets the above requirements by adopting a highly extensible and scalable architecture. The design and implementation of this framework is motivated by our previous work [2], [3] which experimentally compared seven compression algorithms using real-world trajectory data and proposed a new algorithm, called SQUISH (Spatial QUalIty Simplification Heuristic), for achieving highly competitive compression accuracy with substantially lower overhead. Users of our framework can evaluate compression algorithms for various combinations of parameter values, data sets, and evaluation metrics by creating and submitting a concise configuration file. Then, the framework automatically schedules and efficiently carries out the specified evaluations using all servers available in the system. The results of these evaluations are stored in a shared database, thereby promoting efficient and convenient analysis of them. Our framework also supports one state-of-the-art trajectory generator capable of realistic modeling of traffic flow [9], as well as one extended version of a generator [10] and another newly developed generator which enables a tight control over the variation of speed and direction. The entire benchmarking framework is planned to be made available online as open-source by the summer of 2013.

The contributions made in this paper are as follows:

- the design and implementation of a highly extensible benchmarking framework which enables efficient and convenient evaluation of trajectory compression algorithms
- a detailed comparison of existing synthetic trajectory generators with a description of additional development and extension of generators to overcome previous limitations.
- a comprehensive overview of metrics for evaluating trajectory compression algorithms with an emphasis on the trade-offs and relevant use cases of these metrics
- a concise review and comparison of trajectory compression algorithms.
- evaluation results that demonstrate the utility of our benchmarking framework and the unique benefits of each compression algorithm and synthetic trajectory generator.

The remainder of the paper is organized as follows: Section II presents definitions of metrics for evaluating compression algorithms. Sections III and IV review the literature on synthetic trajectory generators and trajectory compression algorithms, respectively. Section V describes the architecture and features of our benchmarking framework. Section VI explains the development and extension of synthetic data generators for overcoming the limitations of existing generators. Section VII provides evaluation results obtained from our framework. Section VIII concludes this paper.

II. Metrics

This section describes metrics for evaluating trajectory compression algorithms. Given an input trajectory, these algorithms produce an output trajectory that consists of a subset of the points from the input trajectory. These "lossy" compression algorithms can trade the accuracy of the output trajectory for a shorter compression time or a greater compression ratio. This section presents a comprehensive survey of both accuracy metrics (Section II-A) and performance metrics (Section II-B), as well as a detailed discussion of these metrics (Section II-C).

A. Accuracy Metrics

Let $T = \langle (x_i, y_i, t_i) : i \in \{1, 2, \cdots, n\} \rangle$ denote a trajectory which consists of n points, where x_i and y_i represent the longitude and latitude, respectively, of a moving object at time t_i. Also, assume that the above trajectory is compressed into $T' = \langle (x_j, y_j, t_j) : j \in M \rangle$ for some $M \subset \{1, 2, \cdots, n\}$. Researchers have developed the following metrics for expressing the accuracy of T' with respect to a point in T.

1) Spatial Error: The spatial error of the compressed trajectory T' with respect to a point (x_i, y_i, t_i) in T is defined as the distance between the actual location (x_i, y_i) and estimated location (x'_i, y'_i) of that point [4]. If T' contains (x_i, y_i, t_i), then the spatial error of T' with respect to (x_i, y_i) is 0. Otherwise, two points in T' that are the closest predecessor and successor of (x_i, y_i, t_i) in T are found (denoted as $pred_{T,T'}(i)$ and $succ_{T,T'}(i)$, respectively). For example, if T' contains (x_1, y_1, t_1), (x_2, y_2, t_2), and (x_4, y_4, t_4), then $pred_{T,T'}(3) = (x_2, y_2, t_2)$ and $succ_{T,T'}(3) = (x_4, y_4, t_4)$. Then, (x'_i, y'_i) is the closest point to (x_i, y_i) along the line from $pred_{T,T'}(i)$ to $succ_{T,T'}(i)$.

2) Synchronized Euclidean Distance: A major limitation of spatial error is that it does not take temporal data into account. Synchronized Euclidean Distance (SED) [5] overcomes this limitation. As in the case of spatial error, the SED between an actual point (x_i, y_i, t_i) and an estimated point (x'_i, y'_i, t_i) is defined as the distance between (x_i, y_i) and (x'_i, y'_i). However, x'_i and y'_i are estimated via linear interpolation between $pred_{T,T'}(i)$ and $succ_{T,T'}(i)$. This interpolation preserves, across the longitude, latitude and time values, the ratio of the difference between $pred_{T,T'}(i)$ and (x'_i, y'_i, t_i) to the difference between (x'_i, y'_i, t_i) and $succ_{T,T'}(i)$.

3) Heading Error: The heading error of T' with respect to a point (x_i, y_i, t_i) in T is defined as the angular displacement between the movement from $(x_{i-1}, y_{i-1}, t_{i-1})$ to (x_i, y_i, t_i) and that from $(x'_{i-1}, y'_{i-1}, t_{i-1})$ to (x'_i, y'_i, t_i), where x'_{i-1}, x'_i, y'_{i-1}, and y'_i are estimated as in the case of SED. This error metric is particularly useful for detecting erratic behavior or disturbances in typical traffic flow [11].

4) Speed Errror: Travel speed is an important metric for a variety of transportation applications. For example, law enforcement utilizes speed information to derive speeding hot-spots [12]. Furthermore, acceleration/deceleration data is useful for identifying vehicles/drivers that are driving erratically [11]. Speed error is determined in a way similar to heading error, except that it captures the difference in travel speed between actual and estimated movements.

B. Performance Metrics

1) Compression Ratio: Compression ratio is defined as the size of the original trajectory divided by the size of the compressed representation of that trajectory. For instance, a compression ratio of 50 indicates that only 2% of the original points remain in the compressed representation of the trajactory.

2) Compression Time: Compression time refers to the amount of time that it takes to compress a trajectory.

C. Discussion

In contrast to spatial error, SED has the advantage of incorporating temporal data into accuracy calculation. Furthermore, there is a strong correlation between SED, heading, and speed errors (Section VII). For this reason, SED is considered a representative accuracy metric in the remainder of this paper.

In principle, applications require a sensible balance of accuracy, compression ratio, and compression time. As the following scenarios demonstrate, the significance of a metric may vary substantially according to the characteristics of applications:

Scenario 1 (Storage-Bound). Assume a fleet of $3,000$ trucks each of which generates trajectory data for 8 hours per day with a 1 second sampling rate and a sample size of 24 bytes (i.e., approximately 0.7 MB per truck per day or 2.1 GB per day). In this case, a 16 GB memory space and 1 TB disk space can hold the raw trajectory data collected for approximately 8 days and 1.3 years, respectively. The above periods can be extended to 76 days and 13 years by compressing trajectory data with a ratio of 10. This scenario represents a historical archive that requires a compact storage of trajectories using sufficient computational resources.

Scenario 2 (CPU-Bound). The above scenario can keep up with incoming trajectory data only if each trajectory is compressed within $0.48 = \frac{24*60}{3000}$ minutes on average. Furthermore, a higher compression speed is required as more computational resources are used to serve queries on stored trajectories. Compression time becomes the most crucial metric when the scarcest resource is the CPU(s).

Scenario 3 (Error-Bound). An application may tolerate only a small error in trajectory data. In this case, the accuracy metrics in Section II-A become a dominant factor, particularly if there are sufficient computational and storage resources.

III. Review of Synthetic Trajectory Generators

This section summarizes synthetic trajectory generators that are available in the literature. Sections III-A and III-B provide a categorization of these generators. Section III-C compares them with a focus on their unique advantages and limitations.

A. Free-Moving Trajectory Generators

There are software programs that create synthetic trajectories without relying on an underlying network [13], [14], [15]. These *free-moving trajectory generators* typically use parameterized random functions to change the direction and speed of trajectories.

B. Simulation-Based Trajectory Generators

Synthetic trajectories can be generated by using traffic simulators (refer to the SMARTEST project for a detailed evaluation of 58 simulators [16]). Most of these simulators, however, are proprietary, do not generate long trajectory histories, and are not easily extensible.

Our benchmarking framework takes advantage of the following two state-of-the-art open-source trajectory generators:

1) BerlinMOD: BerlinMOD provides moving object data (MOD) obtained by simulating trips around the Berlin metropolitan area [9]. BerlinMOD is based off a realistic model of movement which incorporates a realistic road network and statistics on home and work locations, as well as various types of trips (e.g., shopping, sports). The standard setting of BerlinMOD can create trajectories using $2,000$ vehicles simulated over 28 simulation days. The resulting data set contains a total of $292,693$ trajectories each of which represents a single trip in a travel mode (e.g., passenger car, truck, and bus). The total size of this data set is 19.45 GB.

2) Brinkhoff: Similar to BerlinMOD, the Brinkhoff trajectory generator [10] relies on traffic simulation over a road network. This generator, however, supports more detailed customization than BerlinMOD by allowing users to both change application parameters in the configuration file and override existing classes in the Brinkhoff API. In particular, this generator enables modeling of various impacts such as traffic and external events such as construction zones and weather conditions. This generator can also optimize each trip based on different criteria.

C. Discussion

Free-moving trajectory generators have the advantage of efficiently producing trajectories without any restriction on the variation of speed and direction. Due to the absence of an underlying network, however, free-moving generators have limitations in realistically representing the movement of a car or a pedestrian along a road network. To overcome this limitation, we have developed a free-moving trajectory generator which uses a movement model obtained by analyzing real trajectory data (refer to Section VI-B for further details). This generator is incorporated into our benchmark framework (Section V).

TABLE I. Summary of GPS Trajectory Algorithms (n: trajectory size, λ: target compression ratio, Ψ: maximum spatial error, μ: maximum SED error)

Algorithm	Param.	Online/Offline	Time
Uniform Sampling	λ	Online	$O(n)$
Douglas-Peucker	Ψ	Offline	$O(n^2)$
TD-TR	μ	Offline	$O(n^2)$
Open Window	Ψ	Online	$O(n^2)$
OPW-TR	μ	Online	$O(n^2)$
Dead Reckoning	μ	Online	$O(n)$
SQUISH(λ)	λ	Online/Offline	$O(n \log \frac{n}{\lambda})$
SQUISH(μ)	μ	Offline	$O(n \log n)$

The Brinkhoff generator is superior to other generators in terms of its customization capability. However, this generator is unable to produce detailed trajectories due to the lack of a model to capture small, local movements along road edges. For this reason, we have extended this generator to capture vehicle movement over short durations (Section VI-A). Our benchmark framework supports this extended version of the Brinkhoff generator.

In contrast to Brinkhoff and free-moving trajectory generators, BerlinMOD can produce long, detailed, and realistic trajectories of moving objects. Therefore, our benchmark framework does not require modifications to this generator. BerlinMOD configuration details are provided in Section VI-C.

IV. Trajectory Compression Algorithms

This section describes the trajectory compression algorithms that are supported by our benchmark and evaluated in this paper. Table I compares these algorithms in terms of their parameters and computational complexity. Algorithms that use λ, the target compression ratio, as the input parameter ensure that the ratio of the input trajectory size to the output trajectory size is at most λ, but provide no guarantees on the maximum error bound. Other algorithms strive to maximize the compression ratio while preserving the constraint that the maximum spatial error (or SED error) between the input and output trajectories must be less than Ψ (or μ). Trajectory compression algorithms can also be classified into online or offline algorithms depending on whether or not they process trajectory data as they arrive.

A. Uniform Sampling

Uniform Sampling simply takes every $\lfloor \lambda \rfloor$-th point in the input trajectory. Uniform sampling is fast and simple. It runs in an online fashion, but often results in large spatial and SED errors.

B. Douglas-Peucker

The Douglas-Peucker algorithm [4] approximates a trajectory using a series of line segments. This algorithm initially constructs an approximating line segment using the two end points of the input trajectory. It then recursively repeats the process of finding a point from the input trajectory which lies at the maximum distance from the closest approximating line segment, while using the point to split the line segment into two. The whole process stops when the maximum distance from the input trajectory to the closest line segment is less than Ψ.

The Douglas-Peucker algorithm in general achieves a high compression ratio with relatively high computational overhead.

It is not suitable for real-time applications since it is an offline algorithm (i.e., requires the entire trajectory before compression). Additionally, it does not allow users to set the desired compression ratio.

C. TD-TR

Meratnia and de By [6] indicate that line generalization algorithms, such as Douglas-Peucker, are not suitable for GPS trajectories since they do not take temporal data into account. The TD-TR algorithm [6] is similar to Douglas-Peucker except that it uses SED.

D. Opening Window

Opening Window algorithms [7] begin by creating an opening window anchored at the first point in the original trajectory and adding points to the window until the distance between the original trajectory and a line segment defined by the anchor and a point in the window is larger than Ψ. Then, they add to the output trajectory either the point causing the maximum error (Normal Opening Window Algorithm or NOWA) or the point just before the one that causes the maximum error (Before Opening Window or BOPW). Each point added to the output trajectory is also used as the next anchor of the opening window. The above process completes when the input trajectory ends.

E. OPW-TR

The standard Opening Window algorithms are unsuitable for compressing GPS trajectories because they ignore temporal data. OPW-TR [6] overcomes this limitation by using SED as in the case of TD-TR.

F. Dead Reckoning

Dead Reckoning [8] is an online algorithm that estimates the next point in the input trajectory based on the current point and velocity. The next point in the input trajectory is added to the output trajectory only if the distance between that point and the estimated point is larger than μ. Dead Reckoning repeats the above process until it reaches the end of the input trajectory.

The computational complexity of Dead Reckoning is $O(n)$, where n is the number of points in the input trajectory. This complexity is due to the fact that it takes only $O(1)$ time to compare each point with the corresponding predicted location. A major advantage of Dead Reckoning is that it runs in online mode. Dead Reckoning, however, does not allow users to set the compression ratio a priori and usually leads to a relatively low compression ratio.

G. SQUISH

We have developed a novel trajectory compression algorithm called SQUISH [2]. This algorithm includes points in a standard priority queue [17], with the priority of a point being an estimate of the maximum SED error that could be introduced if the point were to be removed. Given λ, the target compression ratio, SQUISH keeps only $\beta = n/\lambda$ points in the queue. If a point is inserted into the queue when the queue contains β points, the point with the lowest priority (i.e., smallest estimated error) is removed from the queue in $O(\log \beta)$ time and the priorities of the points that were adjacent to the removed point are updated in $O(1)$ time. If μ, the tolerable SED error, is given, SQUISH preserves in its queue only the points whose priority is larger than μ. Our previous work [2] has shown SQUISH to be comparable to TD-TR, the most accurate algorithm, while running at significantly faster speed.

```
# Directories
input=~/benchmark/data/original/
output=~/benchmark/data/compressed/ratio10/

# Executors with Compression Methods
executor=DefaultExecutor(SQUISH(*), 10.0)
......
executor=GuaranteedCompressionRatioExecutor(TDTR(*),10,0.5)

# Metrics
metrics=(SED, {Average, Maximum})
......
metrics=(HeadingError, {Average, Maximum})
```

Fig. 2. Example Configuration File

V. Benchmarking Framework

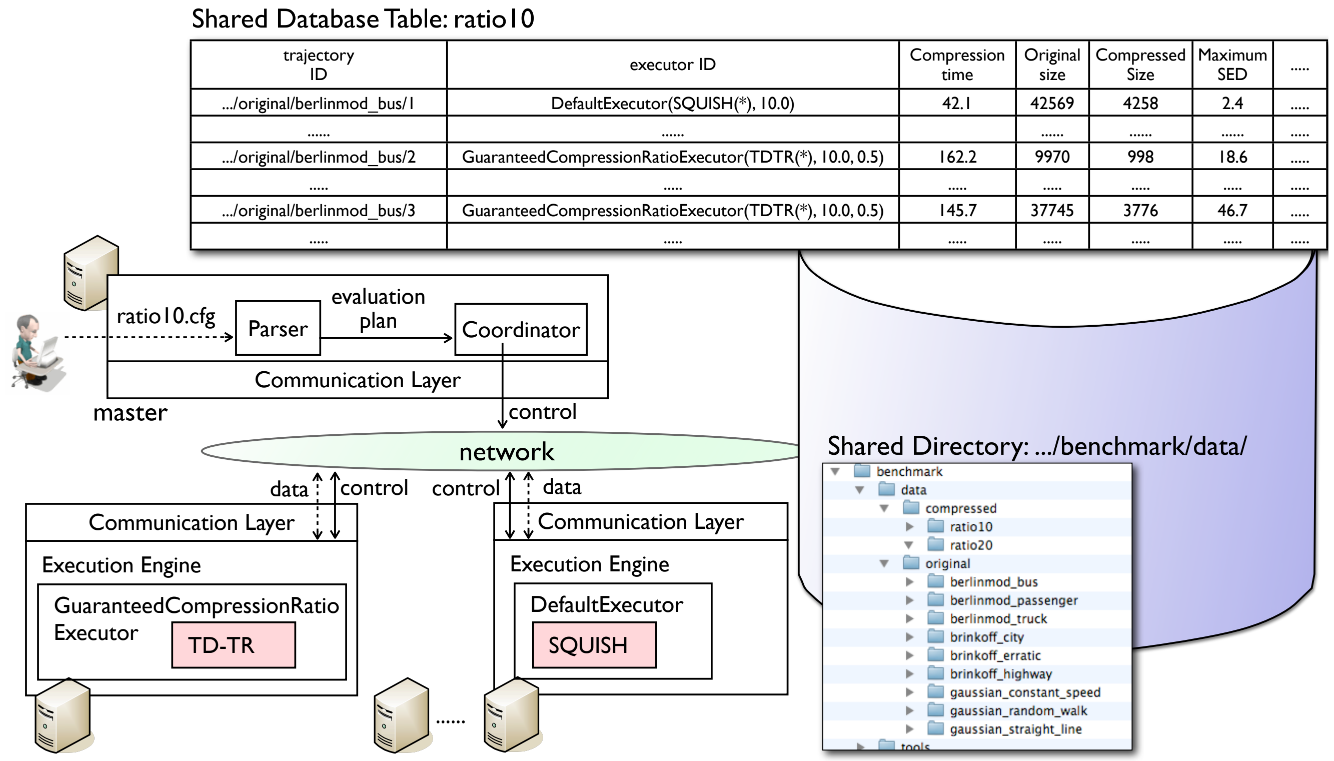
This section presents our benchmarking framework for comparing trajectory compression algorithms. Section V-A provides an architectural overview of the framework. Sections V-B and V-C explain the implementation details of trajectory compression algorithms and evaluation metrics.

A. Overview

Our benchmark framework supports a wide spectrum of trajectory compression algorithms, metrics for evaluating these algorithms, and tools which facilitate the incorporation of newly developed or extended algorithms and metrics. As Figure 1 shows, this framework consists of a master server which controls the overall system and worker servers that evaluate compression algorithms using a collection of trajectories. Whenever a trajectory is compressed, the values of performance metrics (i.e., compression time and compression ratio) as well as user-specified accuracy metrics (e.g., the maximum SED error) are obtained. Accuracy error metrics are measured by comparing the original trajectory and the compressed representation of the trajectory (Section II-A). Then these values are stored altogether as a record in a database table (Figure 1). To execute the benchmark, a user only needs to submit a configuration file (e.g., `ratio10.cfg` in Figure 1) to the master. Then, the master parses the configuration file, constructs an evaluation plan, and assigns evaluation tasks to available servers in the cluster.

Figure 2 shows an example configuration file. This file specifies the directory on the shared file system (e.g., `benchmark/data/original`) which contains the trajectory collections (e.g., `berlinmod_bus`, `berlinmod_passenger`, and others shown in Figure 1) for benchmarking. These trajectories can be produced by using the generators described in Section VI. In addition to the above input directory, the configuration file specifies the directory where the output trajectories are stored, modules (e.g., `DefaultExecutor` and `GuaranteedCompressionRatioExecutor`) for executing compression algorithms, as well as performance metrics (e.g., `SED`) and operations (e.g., `Average` and `Maximum`) for obtaining aggregate metric values computed over all of the points in the original trajectory.

The aforementioned `DefaultExecutor` and `GuaranteedCompressionRatioExecutor` modules are for accurately evaluating compression algorithms on a fair basis despite their differences. For examples, algorithms such



Shared Database Table: ratio10

trajectory ID	executor ID	Compression time	Original size	Compressed Size	Maximum SED	.....
.../original/berlinmod_bus/1	DefaultExecutor(SQUISH(*), 10.0)	42.1	42569	4258	2.4	.....
......	......		......	......	......	.....
.../original/berlinmod_bus/2	GuaranteedCompressionRatioExecutor(TDTR(*), 10.0, 0.5)	162.2	9970	998	18.6	.....
.....	.....	.....	.....	.....	.....	.....
.../original/berlinmod_bus/3	GuaranteedCompressionRatioExecutor(TDTR(*), 10.0, 0.5)	145.7	37745	3776	46.7	.....
.....	.....	.....	.....	.....	.....	.....

Fig. 1. Architecture of Benchmarking Framework

as SQUISH(λ) and Uniform Sampling compress trajectories while guaranteeing a specific compression ratio (Table I). In contrast, algorithms such as SQUISH(μ), Douglas-Peucker, and TD-TR strive to maximize compression ratio under a certain accuracy constraint. The `DefaultExecutor` on line 6 in Figure 2, runs the specified compression implementation (e.g., `SQUISH` which implements SQUISH(λ)) using the specified parameter value (e.g., `10.0`) when assigned to an available server (Figure 1). If the compression time is too short (e.g., 1 millisecond) to be considered a reliable measurement, this module repeatedly runs the compression algorithm up to a predefined amount of time (e.g., 10 seconds by default) and then uses the average compression time obtained over these recent executions . In contrast to `DefaultExecutor`, `GuaranteedCompressionRatioExecutor` varies the parameter value of the specified compression algorithm using binary search until the compression ratio is within the desired range (e.g., between 9.5 and 10.5 on line 8 in Figure 2).

```
public class SED implements ErrorMetric {

  public double evaluate(Trajectory c, Point p) {
    return c.estimatedPoint(p.t).distanceTo(p);
  }

}
```

Fig. 3. Implementation of `SED` Metric

```
public class Maximum extends AggregateOperator {

  double maximum = Double.NEGATIVE_INFINITY;

  public void update(double value) {
    maximum = Math.max(maximum, value);
  }

  public double aggregateValue() {
    return maximum;
  }

}
```

Fig. 4. Implementation of `Maximum` Aggregate Operator

B. *Integration of Compression Algorithms*

Our benchmarking framework represents each trajectory as a `Java` object of the `Trajectory` type. Each `Trajectory` object is an ordered collection of `Point` objects whose `x`, `y` and `t` attributes represent the longitude, latitude, and time, respectively, of a point in the trajectory. Each compression algorithm implements the `TrajectoryCompressor` interface which contains a method, `compress(Trajectory o)` method, where `o` is the trajectory to compress. This method returns a `Trajectory` object which is a compressed representation of the original trajectory `o`.

Integrating a new compression algorithm into our benchmarking framework requires only writing a `Java` class which implements the `TrajectoryCompressor` interface, and adding the name of that `Java` class and the parameter values in the configuration file for benchmarking.

C. *Integration of Evaluation Metrics*

Our benchmarking framework supports all of the evaluation metrics summarized in Section II. The actual code for the accuracy metrics in Section II-A is written in the form of `Java` classes which implement the `ErrorMetric` interface. This interface contains a method, `evaluate(Trajectory c, Point p)`, which returns a numeric error value derived from the compressed trajectory `c` with respect to a point `p` in the original trajectory. Figure 3 shows our implementation of the SED metric.

Our framework also provides aggregate operations for obtaining a representative value (e.g., maximum, average) computed with respect to all of the points in the original trajectory. The classes that implements these operations extend the `AggregateOperator` class. Figure 4 shows our code that implements the `Maximum` aggregate operation. Custom code for an error metric or an aggregate operation can be easily incorporated into our benchmarking framework by writing a `Java` class which implements the `ErrorMetric` or extends the `AggregateOperator` class. The name of the new class must then be added to the relevant configuration files.

VI. Our Trajectory Generators

This section describes the trajectory generators that were either modified or developed for our benchmarking framework. Each generator was chosen to fulfill a unique niche, targeting a particular aspect for compression. These generators include an extended version of the Brinkhoff generator that can produce relatively realistic trajectories with a tight control on the moving speed (Section VI-A), our Gaussian trajectory generator which can significantly vary speed and heading (Section VI-B), and the BerlinMOD generator which takes advantage of a mature road network model (Section VI-C).

A. *Modification of Brinkhoff Generator*

The original Brinkhoff generator cannot produce long, densely sampled trajectories. In particular, it can report vehicle locations only at the end points of road segments. Furthermore, it may drastically change the speed of a vehicle when it exits or enters a new road. We extended this Brinkhoff generator so that it can take into account arbitrary points on road segments and support smoothing of travel speed.

This extended Brinkhoff generator provides three modes: "Highway", "City", and "Erratic". The "Highway" mode determines the route of every trip with a preference to highways, meaning the production of relatively straight trajectories that represent fast movements. On the other hand, the "City" mode prefers local roads to highways and therefore tends to generate trajectories with more changes in speed and direction compared to the "Highway" mode. The third "Erratic" mode produces trajectories which contain drastic, unpredictable speed changes. All of these modes are implemented by changing the weight of each road segment when the routes between a pair of locations are determined using the A* algorithm.

B. *Gaussian Generator*

Due to the absence of an underlying network, free-moving trajectory generators have a limitation in modeling moving objects. To address this limitation, we developed a new free-moving trajectory generator which varies the speed and heading of an object according to statistics obtained from real-world trajectories. Our analysis of data collected from commuters in New York City [3] is shown in Figures 5 and 6, in which the distribution of changes in speed and heading are approximated using Gaussian distributions. For this reason, we call this generator the Gaussian trajectory generator.

Given the current location (Φ, Λ) in a trajectory, the Gaussian generator determines the next location (Φ', Λ') using parameters that define changes in speed and heading. These parameters include μ_{speed} and $\mu_{heading}$, which denote the mean of the changes in speed and in heading, respectively. Additional

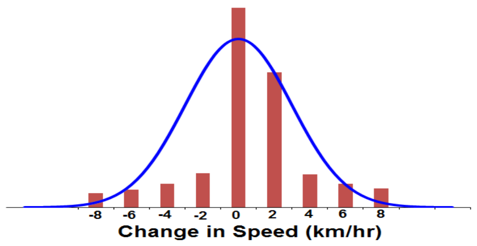


Fig. 5. Distribution of Changes in Speed

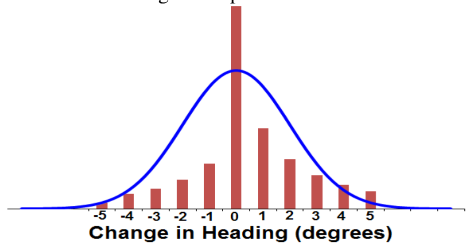


Fig. 6. Distribution of Changes in Heading

parameters are the standard deviation of the changes in speed and heading, which are denoted as σ_{speed} and $\sigma_{heading}$, respectively. The current speed (Δ) and heading (Θ) are determined by randomization formula described below (adjustments for keeping Δ within a certain range are omitted):

$$\Delta \leftarrow \Delta + \text{random}() \cdot \sigma_{speed} + \mu_{speed}$$
$$\Theta \leftarrow \Theta + \text{random}() \cdot \sigma_{heading} + \mu_{heading}$$

Then, the next location (Φ', Λ') is determined as follows:

$$\Phi' = \text{asin}(\sin(\Phi)\cos(d/R) + \cos(\Phi)\sin(d/R)\cos(\Theta))$$
$$\Lambda' = \Lambda + \text{atan2}(\omega, \zeta)$$

where R is the earth's radius, d is the product of the speed (Δ) and the sampling rate of the trajectory, $\omega = \sin(\Theta)\sin(d/R)\cos(\Phi)$, and $\zeta = \cos(d/R)\sin(\Phi)\sin(\Phi')$.

Our Gaussian generator supports the following modes:

Straight Line (speed). This mode constructs trajectories that maintain a constant heading, but contains Gaussian changes in speed. This mode is used for evaluations that focus on speed errors.

Constant Speed (heading). This mode produces trajectories that maintain a constant speed, but contains Gaussian changes in heading, which enables evaluations focused on heading errors.

Random Walk (SED). This mode generates trajectories that have Gaussian speed and heading changes. This mode is for evaluating trajectory compression algorithms with an emphasis on SED errors.

C. *BerlinMOD Generator*

The BerlinMOD trajectory generator uses a parameter called the scale factor Υ, which determines the duration of the simulation in terms of the number of simulation days. For our experiments, Υ was set to 0.05, which created a data set spanning 6 days and consisting of 447 vehicles traveling a total of $15,045$ kilometers. Short trips were excluded from compression, leaving $2,566$ trips, including 165 truck trips, 121 bus trips, and $2,280$ passenger trips. The average trip distance was 16.1 kilometers. The mean speed was 42.5 kilometers per hour, with a standard deviation of 18.2 kilometers per hour. The

TABLE II. COMPRESSION RESULTS FOR A COMPRESSION RATIO OF 10 (n: number of points in the input trajectory, β: number of points kept in the priority queue, ν: number of points kept in the window)

	US	**DR**	**SQ(λ)**	**SQ(μ)**	**DP**	**TD-TR**	**OPW**	**OPW-TR**	**Average**
SED (meters)	15.8	14.3	10.2	9.7	33.2	8.6	24.4	13.3	16.2
Spatial (meters)	5.7	6.0	4.6	4.6	4.1	4.2	6.5	5.6	5.2
Speed (meters/sec)	25.3	24.4	21.4	19.1	23.8	17.8	22.0	24.4	22.3
Heading (degrees)	17.5	18.9	14.7	13.4	23.1	12.6	20.2	17.5	17.24
Compression Time (ms)	0.6	3.5	59.4	42.4	55.0	151.3	12.8	164.2	62.1
Memory Usage	O(1)	O(1)	O(log $\frac{n}{\lambda}$)	O(n)	O(n)	O(n)	O(ν)	O(ν)	N/A

TABLE III. ACCURACY RANKINGS FOR A COMPRESSION RATIO OF 10

	US	**DR**	**SQ(λ)**	**SQ(μ)**	**DP**	**TD-TR**	**OPW**	**OPW-TR**
SED	6	5	3	2	8	1	7	4
Spatial	6	7	4	3	1	2	8	5
Speed	8	6	3	2	5	1	4	7
Heading	4	6	3	2	8	1	7	5
Overall Ranking	6.0	6.0	3.3	2.3	5.5	1.3	6.5	5.3

TABLE IV. PERFORMANCE RANKINGS FOR A COMPRESSION RATIO OF 10

	US	**DR**	**SQ(λ)**	**SQ(μ)**	**DP**	**TD-TR**	**OPW**	**OPW-TR**
Computation Time	1	2	6	4	5	7	3	8
Memory Usage	1	1	3	6	6	6	4	4
Overall Ranking	1.0	1.5	4.5	5.0	5.5	6.5	3.5	6.0

TABLE V. LIST OF COMPRESSION ALGORITHMS

Abbreviation	**Algorithm Name**
US	Uniform Sampling
DR	Dead Reckoning
SQ(λ)	SQUISH(λ)
SQ(μ)	SQUISH(μ)
DP	Douglas Peucker
TD-TR	TD-TR
OPW	Opening Window
OPW-TR	Opening Window (SED)

mean acceleration was 4.1 km/hr^2, with a standard deviation of 8.2 km/hr^2. In terms of the above statistics, there was no significant difference among different travel modes.

VII. BENCHMARKING RESULTS

This section presents evaluation results that compare all of the trajectory compression algorithms described in Section IV. We obtained these results by running our benchmark with a total of nine data sets from the three trajectory generators mentioned in Section VI. Each trajectory contained up to 37,000 points. In all of the evaluation cases, a compression ratio of 10 is achieved. Table II summarizes the characteristics of compression algorithms in terms of the error and performance metrics defined in Section II. Table V shows the abbreviations used to refer to the compression algorithms in Tables II, III, IV, and VI.

To highlight the benefits of each compression algorithm relative to the other algorithms, Tables III and IV present the rankings of these algorithms in terms of accuracy and performance, respectively. For example, Table III shows that, among the 8 compression algorithms that are compared, TD-TR achieves the smallest average SED error. In contrast, Douglas-Peucker results in the largest average SED error. The overall ranking of each compression algorithm in Table III is obtained by averaging the rankings of the algorithm in the SED, spatial, speed and heading error rows. In terms of this metric, the most accurate algorithm is TD-TR with a ranking of 1.3 and the second and third most accurate are SQUISH(μ) and SQUISH(λ) with rankings of 2.3 and 3.3, respectively. A strong correlation can be observed in Table III between an algorithm's rank in the SED row and the algorithm's ranks in the speed and heading rows. SED also has an advantage over spatial error in that it takes temporal data into account (Section II-C). Due to these benefits, the remainder of this section considers SED as a representative error metric.

Table IV ranks trajectory compression algorithms according to performance metrics, which indicate the average amount of time and space used for compressing trajectories. The algorithm that incurs the lowest computation and space overhead is Uniform Sampling, followed by Dead Reckoning and Opening Window. SQUISH(λ) ranks the fourth in overall performance, but achieves much more accurate compression than the above three. Furthermore, both SQUISH(λ) and SQUISH(μ) significantly outperform TD-TR, the most accurate algorithm, in terms of speed and memory efficiency at the expense of slightly lower accuracy.

Table VI presents the average SED achieved by each compression algorithm for a compression ratio of 10 and for each of nine data sets obtained from three data generators. The table shows that the average SED can vary significantly depending on the compression algorithm and the data set. For example, Douglas-Peucker results in an average SED of 95.1 meters for trajectories produced by our Gaussian generator in the "Random Walk" mode. In contrast, both SQUISH(μ) and TD-TR achieve an average SED of 0.9 meters for trajectories produced by the Brinkhoff generator in the "Highway" mode.

Table VI shows the advantage of our Gaussian trajectory generator which allows us to observe the effectiveness of compression algorithms under significant changes only in speed ("Straight Line"), only in direction ("Constant Speed"), and in both speed and direction ("Random Walk"). The benefit of BerlinMOD is that it more accurately models typical urban transportation patterns and fluctuations than other generators.

TABLE VI. AVERAGE SED (IN METERS) FOR EIGHT COMPRESSION ALGORITHMS AND NINE DATA SETS FROM THREE TRAJECTORY GENERATORS (compression ratio: 10)

	US	DR	SQ(λ)	SQ(μ)	DP	TD-TR	OPW	OPW-TR	Average
Gaussian - Random Walk	56.3	61.2	51.9	52.1	95.1	47.4	66.1	60.9	61.4
Gaussian - Constant Speed	15.2	16.7	15.0	13.5	22.6	10.9	17.5	18.3	16.2
Gaussian - Straight Line	15.1	20.0	13.0	11.9	0.0	10.3	0.0	17.0	14.6
BerlinMOD - Bus	10.3	9.3	4.6	3.4	41.8	3.2	22.7	7.8	12.9
BerlinMOD - Passenger	7.4	8.1	4.1	3.3	38.2	2.9	21.6	7.0	11.6
BerlinMOD - Truck	37.7	8.5	4.1	3.3	70.5	2.8	70.8	7.2	25.6
Brinkhoff - City	1.7	1.3	0.4	0.3	3.4	0.4	1.1	0.9	1.2
Brinkhoff - Highway	3.0	2.4	1.0	0.7	6.9	0.7	8.0	1.8	3.1
Brinkhoff - Erratic	10.4	14.9	8.4	11.9	20.7	7.2	11.8	11.9	12.1
Average	17.5	15.8	11.4	11.2	37.4	9.5	27.5	14.8	17.7

For this reason, the accuracy results from BerlinMOD in Table VI are consistent with our previous evaluation results which used actual GPS trajectory data [2], [3]. The Brinkhoff generator has the limitation that it cannot produce as realistic trajectories as BerlinMOD. However, our extension to Brinkhoff can produce trajectories with a tight control over the variation of speed. In particular, the "Erratic" mode of Brinkhoff produces trajectories with significantly varying speed. Therefore, when TD-TR, SQUISH(λ), and SQUISH(μ) compress these trajectories, the average SED values are relatively high compared to when they process trajectories produced in other modes of Brinkhoff, or by BerlinMOD. However, these SED values are still small compared to the case of compressing less realistic trajectories produced by our Gaussian generator. The reason behind this phenomenon is that Brinkhoff has the advantage of using an underlying network model, which avoids drastic changes in speed and direction.

VIII. CONCLUSION

Numerous trajectory compression algorithms have been proposed in the literature. This paper presents a new benchmarking framework that allows users to conveniently and efficiently evaluate these algorithms. Due to its extensible and scalable design, this framework facilitates the development and integration of new trajectory compression algorithms, and enables large-scale evaluations of compression algorithms using a possibly large number of servers. Furthermore, this framework can effectively measure the efficiency and reliability of compression algorithms using both realistic and irregular trajectories. This paper provides a comprehensive overview of trajectory compression algorithms and metrics for evaluating them. Their unique trade-offs and appropriate use cases are also discussed in detail according to actual evaluation results from the benchmarking framework. We intend to release the source code of our benchmarking framework by the summer of 2013.

IX. ACKNOWLEDGEMENTS

This work has been supported by the National Science Foundation under CAREER award IIS-1149372 as well as the Research and Innovative Technology Administration of the U.S. Department of Transportation through the Region 2 - University Transportation Research Centers Program.

REFERENCES

[1] Kathryn Zickuhr. Three-quarters of smartphone owners use location-based services. Technical report, Pew Research Center, 2012.

[2] Jonathan Muckell, Jeong-Hyon Hwang, Vikram Patil, Catherine T. Lawson, Fan Ping, and S. S. Ravi. SQUISH: an online approach for GPS trajectory compression. In *COM.Geo'11*, pages 13.1–13.8, 2011.

[3] Jonathan Muckell, Jeong-Hyon Hwang, Catherine T. Lawson, and S. S. Ravi. Algorithms for compressing GPS trajectory data: An empirical evaluation. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '10, pages 402–405, New York, NY, USA, 2010. ACM.

[4] D.H. Douglas and T.K. Peucker. Algorithms for the reduction of the number of points required to represent a line or its caricature. *The Canadian Cartographer*, 10(2):112–122, 1973.

[5] Michalis Potamias, Kostas Patroumpas, and Timos Sellis. Sampling trajectory streams with spatio-temporal criteria. In *18th International Conference on Scientific and Statistical Database Management (SSDBM'06)*, pages 275–284, 2006.

[6] Nirvana Meratnia and Rolf A. de By. Spatiotemportal compression techniques for moving point objects. In *Advances in Database Technology*, volume 2992, pages 551–562. Springer Berlin / Heidelberg, 2004.

[7] Eamonn J. Keogh, Selina Chu, David Hart, and Michael J. Pazzani. An online algorithm for segmenting time series. In *Proceedings of the 2001 IEEE International Conference on Data Mining*, ICDM '01, pages 289–296, 2001.

[8] Goce Trajcevski, Hu Cao, Peter Scheuermann, Ouri Wolfson, and Dennis 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 (MobiDE'06)*, pages 19–26, New York, NY, USA, 2006. ACM.

[9] Ralf H Guting, Thomas Behr, and Christian Duntgen. Second: A platform for moving objects database research and for publishing and integrating research implementations. In *Fachbereich Informatik*, 2010.

[10] Thomas Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6:153–180, 2002.

[11] M. Karpinski, A. Senart, and V. Cahill. Sensor networks for smart roads. *Pervasive Computing and Communications Workshops*, 2006.

[12] J.G. Harper. Traffic violation detection and deterrence: Implications for automatic policing. *Applied Ergonomics*, 22(3):189 – 197, 1991.

[13] Dieter Pfoser and Yannis Theodoridis. Generating semantics-based trajectories of moving objects. In *International workshop on emerging technologies for geo-based applications*, pages 59–76, 2000.

[14] Theodoros Tzouramanis, Michael Vassilakopoulos, and Yannis Manolopoulos. On the generation of time-evolving regional data. *GeoInformatica*, 6:207–231, 2002.

[15] Jean-Marc Saglio and José Moreira. Oporto: A realistic scenario generator for moving objects. *GeoInformatica*, 5:71–93, 2001.

[16] Jaime Barceló et al. Simulation modelling applied to road transport european scheme tests (SMARTEST) - Review of microsimulation models. In *Institute for Transport Studies, University of Leeds*, 1998.

[17] T. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw Hill, 2009.