Compression of GPS trajectory data: benchmarking framework and new approach

Jonathan Muckell
University at Albany, State University of New York, jonmuckell@gmail.com

The University at Albany community has made this article openly available. Please share how this access benefits you.

Follow this and additional works at: https://scholarsarchive.library.albany.edu/legacy-etd

Recommended Citation
https://scholarsarchive.library.albany.edu/legacy-etd/958

This Dissertation is brought to you for free and open access by the The Graduate School at Scholars Archive. It has been accepted for inclusion in Legacy Theses & Dissertations (2009 - 2024) by an authorized administrator of Scholars Archive.
Please see Terms of Use. For more information, please contact scholarsarchive@albany.edu.
COMPRESSION OF GPS TRAJECTORY DATA: BENCHMARKING FRAMEWORK AND NEW APPROACH

DISSERTATION

by

Jonathan Muckell

A Dissertation
Submitted to the University at Albany, State University of New York
in Partial Fulfillment of
the Requirements for the Degree of
Doctor of Philosophy

College of Computing and Information
Department of Informatics
University at Albany
Albany, New York
2013
COMPRESSION OF GPS TRAJECTORY DATA: BENCHMARKING FRAMEWORK AND NEW APPROACH

DISSERTATION

by

Jonathan Muckell

© Copyright 2013
To my lovely wife, Alison.
ABSTRACT

GPS-equipped mobile devices such as smart phones and in-car navigation units are collecting enormous amounts of spatial and temporal information that traces a moving object’s path. The exponential increase in the amount of such trajectory data has caused three major problems. First, transmission of large amounts of data is expensive and time-consuming. Second, queries on large amounts of trajectory data require computationally expensive operations to extract useful patterns and information. Third, GPS trajectories often contain large amounts of redundant data that waste storage and cause increased disk I/O time. These issues can be addressed by algorithms that reduce the size of trajectory data. This dissertation 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 dissertation presents a benchmarking framework for efficiently, conveniently, and accurately comparing trajectory compression algorithms. A key requirement for these algorithms is to minimize the loss of information essential to location-based applications. To address this requirement, this research introduces a new compression method called SQUISH (Spatial QUality Simplification Heuristic) that provides improved run-time performance and usability. A comprehensive comparison of SQUISH with other algorithms is carried out using the introduced benchmarking framework across three types of real-world datasets and three synthetic data generators.
ACKNOWLEDGMENT

This dissertation would not have been possible with the dedication and support of many individuals.

I offer my sincerest gratitude to my committee chair, Prof Catherine Lawson, who made this work possible by her unwavering commitment, encouragement and unique vision. My deepest appreciation to my other chairpersons. Prof Jeong-Hyon Hwang for his vast intelligence, clever solutions, and companionship during late night coding and writing sessions. Prof S.S. Ravi for steadfast dedication to learning (that has gone far beyond that of even the most committed educators), and for sharing his diverse computing knowledge.

During this work I have collaborated with many colleagues for whom I have great regard, and I wish to extend my warmest thanks to all those who have helped me with my work. Special thanks also to all my graduate friends, especially research group members; Dr. Fan Ping, Paul Olsen, and Vikram Patil for their contributions and valuable discussions. I would also like to thank researchers at General Electric Global Research Center; specifically, Dr. Patricia Mackenzie and Daniel Messier for their knowledge on supply-chain tracking, assistance in intellectual property and collaboration on grant writing. Special thanks to Prof Randolph Franklin and Prof Barb Culter at Rensselaer Polytechnic Institute and Prof Marcus Andrade at the Universidade Federal de Viçosa for initially introducing me to geographic information systems and compression algorithms. I would also like to thank my undergraduate adviser Prof Ed Harcourt for his outstanding advice in graduate school preparation and continuous support.

Above all, I appreciate the many years of love and support from my wife Alison, who selfishlessly sacrificed to make this dissertation possible. I will also be forever grateful to my parents for their encouragement, and for instilling the value of hard work and education.
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iv</td>
</tr>
<tr>
<td>ACKNOWLEDGMENT</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Contributions</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Dissertation Overview</td>
<td>4</td>
</tr>
<tr>
<td>2. Related Work</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Metrics</td>
<td>6</td>
</tr>
<tr>
<td>2.1.1 Accuracy Metrics</td>
<td>6</td>
</tr>
<tr>
<td>2.1.1.1 Spatial Error</td>
<td>7</td>
</tr>
<tr>
<td>2.1.1.2 Synchronized Euclidean Distance</td>
<td>7</td>
</tr>
<tr>
<td>2.1.1.3 Heading Error</td>
<td>8</td>
</tr>
<tr>
<td>2.1.1.4 Speed Error</td>
<td>8</td>
</tr>
<tr>
<td>2.1.2 Performance Metrics</td>
<td>8</td>
</tr>
<tr>
<td>2.1.2.1 Compression Ratio</td>
<td>8</td>
</tr>
<tr>
<td>2.1.2.2 Compression Time</td>
<td>8</td>
</tr>
<tr>
<td>2.1.3 Discussion</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Trajectory Compression Algorithms</td>
<td>9</td>
</tr>
<tr>
<td>2.2.1 Uniform Sampling</td>
<td>10</td>
</tr>
<tr>
<td>2.2.2 Douglas-Peucker</td>
<td>10</td>
</tr>
<tr>
<td>2.2.3 Top-Down Time Ratio (TD-TR)</td>
<td>11</td>
</tr>
<tr>
<td>2.2.4 Opening Window Algorithms</td>
<td>11</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Compression Accuracy Metrics</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Execution of Douglas-Peucker Algorithm</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Execution of STTrace Algorithm</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Execution of SQUISH Algorithm</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>SQUISH Correctness Proof Induction Step</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>Benefit of SQUISH((\lambda, \mu))</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Architecture of Benchmarking Framework</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>Example Configuration File</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>Implementation of SED Metric</td>
<td>38</td>
</tr>
<tr>
<td>4.4</td>
<td>Implementation of Maximum Aggregate Operator</td>
<td>38</td>
</tr>
<tr>
<td>4.5</td>
<td>Implementation of Maximum Aggregate Operator</td>
<td>38</td>
</tr>
<tr>
<td>4.6</td>
<td>Implementation of Maximum Aggregate Operator</td>
<td>38</td>
</tr>
<tr>
<td>4.7</td>
<td>Distribution of Changes in Speed</td>
<td>39</td>
</tr>
<tr>
<td>4.8</td>
<td>Distribution of Changes in Heading</td>
<td>39</td>
</tr>
<tr>
<td>5.1</td>
<td>Average Compression Time</td>
<td>40</td>
</tr>
<tr>
<td>5.2</td>
<td>Average SED Error</td>
<td>45</td>
</tr>
<tr>
<td>5.3</td>
<td>Average Spatial Error</td>
<td>46</td>
</tr>
<tr>
<td>5.4</td>
<td>Average SED Error (per Dataset)</td>
<td>47</td>
</tr>
<tr>
<td>5.5</td>
<td>Ranking in terms of Average SED Error</td>
<td>47</td>
</tr>
<tr>
<td>5.6</td>
<td>Average Speed Error</td>
<td>48</td>
</tr>
</tbody>
</table>
5.7 Comparison of the Previous Version SQUISH and SQUISH(λ) . . . . . 48
5.8 Cumulative Distribution of SED Errors . . . . . . . . . . . . . . . . 50
CHAPTER 1

Introduction

1.1 Motivation

In recent years, the number of GPS-enabled devices sold has drastically increased, following an impressive exponential trend. Canalys, an information technology firm that studies market patterns, reported a 116% increase in the number of GPS units sold between 2006 and 2007[1]. Location-based services and applications built from GPS-eqipped mobile devices represent a rapidly expanding consumer market. In 2009, there was an estimated 27 million GPS-equipped smart phones sold, bringing the world-wide GPS user-base to at least 68 million in the consumer market alone [2]. These devices have the ability to generate, store and transmit trajectory data. A trajectory is defined as a stream of 3-tuple records consisting of the position (latitude, longitude), along with the temporal information (when the moving object was at the location). In addition, trajectory data is commonly used in a variety of business and public section applications, such as supply-chain management, transportation planning and traffic modeling [3, 4, 5, 6, 7].

Three major problems currently exist in location-based applications that use trajectory data. First, storing the sheer volume of trajectory data can quickly overwhelm available data storage space. For instance, if data is collected without compression at 10 second intervals, 1 Gb of storage capacity is required to store just over 4,000 objects for a single day [8]. For this reason, efficiently storing and querying GPS trajectories is an area of active research [8, 9, 10, 11]. Second, the cost of sending a large amount of trajectory data over cellular or satellite networks can be expensive, typically ranging from $5 to $7 per megabyte [12]. Thus, for example, tracking a fleet of 4,000 vehicles for a single day would incur a cost of $5,000 to $7,000, or approximately $1,800,000 to $2,500,000 annually. Third, as the trajectory data size gets larger, it becomes more difficult to detect useful patterns from the data. Reducing the size of the trajectory data has the potential to accelerate the mining of trajectory
patterns [13].

To mitigate the above issues, a variety of trajectory compression algorithms have been developed [8, 14, 15, 16, 17, 18, 19]. 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.

Compression 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 thesis 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 [14, 15] 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 [20], as well as one extended version of a generator [21] and another newly developed generator which enables a tight control over the variation of speed and direction.

In addition to the benchmarking framework, this thesis presents a new compression algorithm SQUISH that overcomes the limitations of long network transmission time, inefficient processing of trajectory data, and wasted memory and disk space. This new algorithm provides provable guarantees on the error caused by compression. Given a trajectory $T$ and parameters $\lambda$ and $\mu$, this algorithm first compresses $T$ while striving to minimize the error due to compression and ensuring the compression ratio of $\lambda$. It then further compresses $T$ as long as this compression will not increase errors beyond $\mu$. In this way, the new SQUISH algorithm allows users to control compression with respect to both compression ratio and error, sharply contrasting with previous compression algorithms. In particular, setting $\mu$ to 0 causes this algorithm to minimize error while achieving the compression ratio of $\lambda$. When $\lambda$ is 1, this algorithm maximizes compression ratio while keeping errors under $\mu$.

1.2 Contributions

The contributions of this dissertation are:

- A review and comparison of trajectory compression algorithms.
- A comprehensive overview of metrics for evaluating trajectory compression algorithms with an emphasis on the trade-offs and relevant use cases of these
metrics.

- Guidelines for choosing a compression algorithm that is best suited to the characteristics of trajectories as well as the organizational and technical requirements.

- An introduction of a new algorithm called SQUISH, that compresses trajectories with provable guarantees on errors. This algorithm has the flexibility of tuning compression with respect to compression ratio and error.

- A formal proof of correctness of the new SQUISH algorithm.

- A detailed comparison of existing synthetic trajectory generators with a description of additional development and extension of generators to overcome previous limitations.

- The design and implementation of a highly extensible benchmarking framework which enables efficient and convenient evaluation 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.

1.3 Dissertation Overview

Chapter 2 describes previous work for compressing trajectories. First performance and accuracy metrics are discussed that are utilized for measuring the effectiveness of trajectory compression. Since applications that utilize trajectories have different goals and limitations, metrics are discussed in relation to application scenarios to understand compression trade-offs. A review of the literature of GPS compression algorithms is provided. Compression algorithms are compared based on error metrics, and additional characteristics such as complexity and run mode (online/batch).

In Chapter 3, the new SQUISH algorithm is introduced, including a benefits proof and discussion outlining the contributions of the new algorithm.
The GPS compression framework is introduced in Chapter 4. A detailed description is provided, specifying how error metrics and compression algorithms are incorporated. A new synthetic data generator (Gaussian) is introduced, along with the modifications of the Brinkhoff generator, and integration with the BerlinMOD generator.

An evaluation using real-world and synthetic datasets using the benchmarking framework is provided in Chapter 5. Comparisons are drawn using an analysis of performance and accuracy metrics to determine the most proper compression algorithms for various application scenarios. The dissertation concludes with recommendations for future work in Chapter 6.
CHAPTER 2
Related Work

Compression algorithms can be classified into two forms, namely lossless and lossy compression. Lossless compression enables exact reconstruction of the original data with no information loss. In contrast, lossy compression introduces inaccuracies when compared to the original data. The primary advantage of lossy compression is that it can often drastically reduce storage requirements while maintaining an acceptable degree of error. Due to this benefit, this paper focuses on lossy compression of trajectory data. This chapter presents a comprehensive survey of metrics for comparing lossy trajectory compression algorithms (Section 2.1) and a summary of these algorithms (Section 2.2).

2.1 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 2.1.1) and performance metrics (Section 2.1.2), as well as a detailed discussion of these metrics (Section 2.1.3).

2.1.1 Accuracy Metrics

Let $T = \langle (x_i, y_i, t_i) : i \in \{1, 2, \ldots, n\} \rangle$ 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 $M \subset \{1, 2, \ldots, n\}$. Researchers have developed the following metrics for expressing the accuracy of $T'$ with respect to a point in $T$. 
Figure 2.1: Compression Accuracy Metrics: Spatial error and SED are illustrated using a trajectory consisting of $P_1, P_2, \cdots, P_6$ and its compressed representation which contains $P_1, P_4$ and $P_6$.

2.1.1.1 Spatial Error

Given a trajectory $T$ and its compressed representation $T'$, the spatial error of $T'$ with respect to a point $P_i$ in $T$ is defined as the distance between $P_i(x_i, y_i, t_i)$ and its estimation $P'_i(x'_i, y'_i, t_i)$. If $T'$ contains $P_i$, then $P'_i = P_i$ (e.g., $P'_1 = P_1$ and $P'_4 = P_4$ in Figure 2.1(a) where a trajectory containing $P_1, P_2, \cdots, P_6$ is approximated using only $P_1, P_4$ and $P_6$). Otherwise, $P'_i$ is defined as the closest point to $P_i$ along the line between $\text{pred}_{T,T'}(P_i)$ and $\text{succ}_{T,T'}(P_i)$, where $\text{pred}_{T,T'}(P_i)$ and $\text{succ}_{T,T'}(P_i)$ denote $P_i$’s closest predecessor and successor among the points in $T'$. In Figure 2.1(a), $\text{pred}_{T,T'}(P_2) = P_1$ and $\text{succ}_{T,T'}(P_2) = P_4$. Therefore, the spatial error of $T'$ with respect to $P_2$ is the perpendicular distance from $P_2$ to line $P_1P_4$.

2.1.1.2 Synchronized Euclidean Distance

A major limitation of spatial error is that it does not take temporal data into account. Synchronized Euclidean Distance (SED) [17] 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 $\text{pred}_{T,T'}(i)$ and $\text{succ}_{T,T'}(i)$. This interpolation preserves, across the longitude, latitude and time values, the ratio of the difference between $\text{pred}_{T,T'}(i)$ and $(x'_i, y'_i, t_i)$ to the difference between $(x'_i, y'_i, t_i)$ and $\text{succ}_{T,T'}(i)$. For instance, if $t_i = i$ for $i \in \{1, 2, 3, 4\}$ in Figure 2.1(b), $P'_2$ and $P'_3$ correspond to the points that divide the line between $P_1$ and $P_4$ into three line segments of the same length.
2.1.1.3 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 [22].

2.1.1.4 Speed Error

Travel speed is an important metric for a variety of transportation applications. For example, law enforcement utilizes speed information to derive speeding hot-spots [23]. Furthermore, acceleration/deceleration data is useful for identifying vehicles/drivers that are driving erratically [22]. 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.

2.1.2 Performance Metrics

2.1.2.1 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 trajectory.

2.1.2.2 Compression Time

Compression time refers to the amount of time that it takes to compress a trajectory. This metric can be measured by a number of difference approaches. The first approach is to simply compress the trajectories to a certain size and measure the time it takes for the algorithm to complete. This is using the so-called wall clock time method. Additional means for computing the time complexity is using the number of CPU cycles required for the algorithm to finish.
2.1.3 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 5.3). For this reason, SED is considered a representative accuracy metric in the remainder of this dissertation.

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 $0.48 = \frac{24 \times 60}{3000}$ 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 2.1.1 become a dominant factor, particularly if there are sufficient computational and storage resources.

2.2 Trajectory Compression Algorithms

Various trajectory compression algorithms exist in the literature. Each offers a different trade-off among compression time, compression ratio, and accuracy. This section summarizes these algorithms.
Figure 2.2: Execution of Douglas-Peucker: Gray dots and dashed lines represent the original trajectory while black dots and solid lines represent the compressed representation of that trajectory. The compressed representation becomes a more accurate approximation of the original trajectory with the addition of a point whose absence caused the largest spatial error.

2.2.1 Uniform Sampling

Given a trajectory $T$ and the target compression ratio $\lambda$, Uniform Sampling down-samples $T$ at fixed time intervals in a manner that achieves the compression ratio of $\lambda$. Uniform sampling is fast, but often introduces large spatial and SED errors.

2.2.2 Douglas-Peucker

Given a trajectory $T$ and a parameter $\Psi$, the Douglas-Peucker Algorithm [16] constructs a new trajectory $T'$ by repeatedly adding points from $T$ until the maximum spatial error of $T'$ becomes smaller than $\Psi$. Figure 2.2 illustrates the operation of this algorithm. This algorithm initially approximates the original trajectory using the first and last points of the trajectory (Figure 2.2(b)). Then, it repeats the process of selecting the point that causes the largest spatial error (e.g., $P_3$ in Figure 2.2(b) and $P_5$ in Figure 2.2(c)) and using that point to more accurately approximate the original trajectory. This process stops when the maximum spatial error (e.g., the distance from $P_4$ to $P_3P_5$ in Figure 2.2(d)) is smaller than $\Psi$. 
The worst-case running time of the original Douglas-Peucker algorithm is $O(n^2)$, where $n$ is the number of original points. This running time can be improved to $O(n \log n)$ using an approach that involves convex hulls [24]. A primary advantage of this algorithm is the guarantee that the resulting spatial error is less than the user-specified bound $\Psi$. A major drawback of Douglas-Peucker is that it ignores temporal data due to the use of spatial error. Douglas-Peucker also does not allow users to set the desired compression ratio.

### 2.2.3 Top-Down Time Ratio (TD-TR)

The Douglas-Peucker algorithm has the limitation of ignoring temporal data. Top-Down Time Ratio (TD-TR) [8] overcomes this limitation by using SED instead of spatial error. The worst-case running time of TD-TR is $O(n^2)$ since it extends the original Douglas-Peucker algorithm. The $O(n \log n)$ implementation of Douglas-Peucker [24] takes advantage of geometric properties that do not hold for SED. Therefore, it cannot be applied to TD-TR.

### 2.2.4 Opening Window Algorithms

Similar to Douglas-Peucker, Opening Window algorithms [18] approximate each trajectory using an increasing number of points so that the resulting spatial error is smaller than a bound $\Psi$. A unique characteristic of Opening Window algorithms is that they slide a window over the points in the original trajectory. This window is initially anchored at the first point and gradually includes subsequent points until the distance from a point $P_j$ in the window to the line segment between the first and last points in the window becomes larger than $\Psi$. Next, either point $P_j$ is added to the compressed representation (Normal Opening Window Algorithm or NOWA) or the point right before $P_j$ is added (Before Opening Window or BOPW). Such an added point is then used as the anchor of the next opening window. The above process is repeated until the last point of the original trajectory is processed. The worst-case running time of Opening Window algorithms is $O(n^2)$ [18], where $n$ is the number of points in the original trajectory.
2.2.5 Opening Window Time Ratio (OPW-TR)

Opening Window Time Ratio (OPW-TR) [8] is an extension to Opening Window which uses SED instead of spatial error. Compared to Opening Window algorithms, OPW-TR has the benefit of taking temporal data into account. Just like Opening Window algorithms, OPW-TR has $O(n^2)$ worst-case running time.

2.2.5.1 Dead Reckoning

Dead Reckoning [19] stores the location of the first point $P_1$ and the velocity at $P_1$ in the compressed representation. It then skips every subsequent point $P_i$ ($i > 1$) whose location can be estimated from the information about $P_1$ within the SED of $\mu$. If $P_j$ is the first point whose location cannot be estimated as above, both the location of $P_j$ and the speed at $P_j$ are stored in the compressed representation, which are used for estimating the location of each point after $P_j$. This process is repeated until the input trajectory ends.

The computational complexity of Dead Reckoning is $O(n)$, where $n$ is the number of points in the original trajectory. This complexity is due to the fact that it takes only $O(1)$ time to compare the actual and estimated locations of each point. The primary disadvantages of Dead Reckoning are that it tends to achieve lower compression ratios than other techniques (Section 5.2.2) and it does not allow users to set the target compression ratio.

2.2.5.2 AmTree

The AmTree data structure [25, 26] is primarily based on the assumption that information value decays over time. AmTree stands for amnesic tree structure that reflects the data structures representation of time-decaying approximation while taking advantage of spatial redundancy. As time progresses, the AmTree will store the GPS trajectories past information at coarser and coarser resolutions. The root of the tree contains information every time-stamp message received, with each additional level from the root of the tree containing half the number of time-stamped messages over a given time interval.
Figure 2.3: The STTrace Algorithm predicts the behavior of the next points based on previous GPS trajectory points. Points within the predicted Safe Area are not stored.

2.2.5.3 STTrace

The STTrace algorithm [17] is designed to preserve spatio-temporal, heading and speed information in a trace. A vector defining the speed and direction between the two locations is used to predict the location of the next point. Two input parameters are used to make this prediction. One of these parameters is the speed tolerance which defines how much the speed can vary while still remaining in the predicted range. The other input parameter is the heading tolerance that defines how much the heading can vary while still remaining in the predicted range.

The two input parameters are used to construct a “safe area” polygon to adjust the degree to which the speed and heading can vary. If the third point in the GPS trajectory is inside this polygon, then it is considered to be within the predicted range and therefore not stored. Otherwise, the point is thought of as exhibiting unexpected behavior. Since such a point may contain information that is vital to the GPS trace, it is included in the compressed representation.

If the STTrace algorithm is implemented as described above, error propagation can be a major problem. This occurs when a point falls on the outer edge of each polygon; thus, it is possible for the error to accumulate, causing the compressed trace to become more and more inaccurate. Small changes at each time stamp over a long enough time period result in major deviations that need to be stored in the compressed version.

In order to resolve the issue of error propagation, STTrace uses a second safe area polygon, shown using horizontal slashes in Figure 2.3. (This figure is a simplified
version of Figure 4 in [17].) This safe area is defined by using two GPS points: the
point three time-steps back, and the point two time-steps back (points \( a \) and \( b \)). The
direction and speed are extrapolated over a two time-stamp distance to determine a
predicted range at a given time. Only points that are within both safe areas, shown
using diagonal slashes, are determined to be within the predicted range. If a point is
outside the intersection of the two polygons, it is stored in the compressed version.
This eliminates the problem of error propagation, at the cost of some additional
computation.

2.2.5.4 Bellman’s Algorithm

Bellman’s algorithm, based on dynamic programming [27, 28], also fits a se-
quence of line segments to a curve. The solution produced by the algorithm is provably
optimal; the algorithm minimizes the root mean square (RMS) error under specific
conditions. A straightforward implementation of the algorithm has a worst-case run-
ning time of \( O(n^3) \), where \( n \) is the number of points in the trajectory. This is a serious
drawback when large trajectories must be compressed. Using additional storage, the
running time of the algorithm can be reduced to \( O(n^2) \) [29].

The input to Bellman’s dynamic programming algorithm is a series of latitude
and longitude points that can be taken directly from the GPS loggers. An additional
positive value \( C \), which represents the penalty for introducing a new line segment into
the compressed representation, is also needed for the algorithm. More line segments
imply better accuracy; however, the corresponding compressed representation needs
more storage. Bellman’s algorithm creates an optimal fitting of the GPS trace using
line segments (where optimality is defined as minimizing the RMS error), taking into
account the penalty factor \( C \).

In order to optimally fit the line segments to the curve, Bellman’s algorithm as-
sumes that the input data is a valid (i.e., single-valued) function; thus, the trajectory
cannot contain no loops. However, due to inaccurate measurements by GPS devices,
loops are often present in trajectory data, causing the application of Bellman’s algo-

...
2.2.5.5 Statistical and Clustering Algorithms

Clustering and statistical aggregation methods that compress many GPS trajectories were not compared in this study due to the difficulty of appropriately measuring the effectiveness of such vastly different techniques. While the algorithms used in this study compress a single GPS trace, the algorithms in the literature that were not compared in this study operate on a GPS trajectory collection comprised of potentially hundreds or thousands of trajectories. A brief description of these other approaches is included below to provide a thorough literature review of available techniques.

Statistical methods are primarily focused on determining appropriate aggregation levels. These techniques are based on comparisons of statistical measures, and more recently, on wavelet decomposition. Data aggregation involves determining the proper sampling interval for storing the most significant spatial-temporal information. By exploiting the characteristics of the data, properly determined aggregation levels can lead to a drastic reduction in storage space. The optimal decomposition level is determined by finding the time interval with the most similarity and hence the least amount of variability during at each time step.

Implementations of statistical techniques utilize the minimal amount of sufficient statistics necessary to capture the full information contained within a parameter distribution. Some statistical solutions use a cross-validated mean square error, while others utilize an F-statistic computation to obtain the optimal aggregation level [30]. The decision as to which approach works best is often based on the most significant statistical approach, in conjunction with market-driven parameters such as the value of the data. Recent work using wavelet decomposition [31, 32, 33] is most often applicable for capturing significant information in intelligent transportation systems. Based on Shannon’s Theorem in information theory [34], wavelet analysis is useful for identifying trends.

Another class of algorithms exploits redundancy across a large set of trajectories. One such example is the Scalable Cluster-Based Algorithm (SCUBA) [35] which aggregates multiple GPS trajectories, instead of a single trace. This algorithm clusters similar GPS trajectories; this process can substantially reduce the space complexity by minimizing the storage of similar information.
Table 2.1: Summary of Trajectory Compression Algorithms ($n$: trajectory size, $\lambda$: target compression ratio, $\Psi$: spatial error bound, $\mu$: SED error bound)

<table>
<thead>
<tr>
<th>algorithm</th>
<th>param.</th>
<th>mode</th>
<th>time complexity</th>
<th>SED (same compres. ratio)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform Sampling</td>
<td>$\lambda$</td>
<td>online</td>
<td>$O(n)$</td>
<td>large</td>
</tr>
<tr>
<td>Douglas-Peucker</td>
<td>$\Psi$</td>
<td>offline</td>
<td>$O(n^2)$ [16], $O(n \log n)$ [24]</td>
<td>large</td>
</tr>
<tr>
<td>TD-TR</td>
<td>$\mu$</td>
<td>offline</td>
<td>$O(n^2)$</td>
<td>large</td>
</tr>
<tr>
<td>Opening Window</td>
<td>$\Psi$</td>
<td>online</td>
<td>$O(n^2)$</td>
<td>small</td>
</tr>
<tr>
<td>OPW-TR</td>
<td>$\mu$</td>
<td>online</td>
<td>$O(n^2)$</td>
<td>large</td>
</tr>
<tr>
<td>Dead Reckoning</td>
<td>$\mu$</td>
<td>online</td>
<td>$O(n)$</td>
<td>large</td>
</tr>
<tr>
<td>SQUISH($\lambda$)</td>
<td>$\lambda$</td>
<td>online/offline</td>
<td>$O(n \log \frac{n}{\lambda})$</td>
<td>small</td>
</tr>
<tr>
<td>SQUISH($\mu$)</td>
<td>$\mu$</td>
<td>offline</td>
<td>$O(n \log n)$</td>
<td>small</td>
</tr>
</tbody>
</table>

2.2.6 Discussion

As summarized in Table 2.1, trajectory compression algorithms have different characteristics in terms of their guarantees on compression (e.g., Uniform Sampling achieves the compression ratio of $\lambda$ while TD-TR limits SED under $\mu$) and compression speed. Douglas-Peucker and TD-TR are offline algorithms that begin compression only after obtaining all of the points from the input trajectory. On the other hand, Uniform Sampling, Opening Window, OPW-TR, and Dead Reckoning are online algorithms which compress each trajectory while they retrieve points from that trajectory. Such online algorithms have the advantages of supporting real-time applications and using a small amount of memory. Trajectory compression algorithms also introduce substantially different SED errors when their parameters are tuned so that they achieve the same compression ratio (Section 5).

Fast compression algorithms such as Uniform Sampling and Dead Reckoning tend to introduce large SED errors. Algorithms that use spatial error (i.e. Douglas-Peucker, Opening Window) also exhibit large SED errors since they ignore temporal data. Online compression algorithms tend to cause larger SED error than TD-TR. TD-TR, however, incurs high computational overhead (i.e., low compression speed) since it recomputes the SED of the current compressed representation with respect to multiple points whenever it adds a point into the compressed representation.

The above shortcomings of the previous algorithms motivated the development of our SQUISH algorithm which is further described in Section 3. This algorithm
supports online compression in that it starts removing points before it reaches the end of the input trajectory. It also has an aspect of an offline algorithm since it produces the final compressed representation only after it has accessed all of the points in the trajectory. Compared to TD-TR, SQUISH achieves competitive accuracy while incurring substantially lower computational and space overhead (Section 5).

The algorithms examined in this study approximate each trajectory using a subset of points from that trajectory. Among such point-selection algorithms, STTrace [17] and Bellman’s Algorithm [27] are not included in this study because of their significantly higher error rates and computational costs compared to other algorithms [36]. Another point-selection algorithm, explored by Feldman et al. [37], uses coresets for approximating a large collection of sensor data. Given a large set of points $S$, a coreset $C$ is a small subset $S$ such that $C$ can be used to approximate $S$ [38]. For practical problems where the number of points is very large, the corresponding algorithms that provide good approximations tend to be computationally expensive [37].

In contrast to point-selection approaches, semantic compression [39] enables extreme compression by storing only the points that represent key events such as transport stops. Semantic compression is effective for applications that have fixed requirements. On the other hand, point-selection approaches are applicable to broader situations since they use error metrics that consider only spatial and temporal data rather than other application-specific information. There has also been research on lossless compression of trajectory data. One such study by Lin et al. [40] uses inter-frame encoding. Lossless compression techniques in general achieve a lower compression ratio compared to lossy compression techniques. Lossy and lossless techniques can also be used in tandem by applying lossy compression and then lossless compression to each trajectory.

Additional lossy compression algorithms and data structures have been purposed. One approach by Kaul et al. [41] approximates trajectories using polynomial prediction. A key finding in this work is that linear prediction which relies on the two most recent GPS updates outperforms more complex polynomial prediction techniques. This approach is not as accurate as Douglas-Peucker and conceptually similar to Dead Reckoning which also uses linear prediction. In addition, this dissertation
was motivated by our prior research regarding algorithms used for compression of terrain [42], geological features [43] and data structures for efficient execution of algorithms on terrain [44].

2.3 Review of Synthetic Trajectory Generators

This section summarizes synthetic trajectory generators that are available in the literature. Sections 2.3.1 and 2.3.2 provide a categorization of these generators. Section 2.3.3 compares them with a focus on their unique advantages and limitations.

2.3.1 Free-Moving Trajectory Generators

There are software programs that create synthetic trajectories without relying on an underlying network [45, 46, 47]. These free-moving trajectory generators typically use parameterized random functions to change the direction and speed of trajectories.

2.3.1.1 GSTD

The GSTD generator [45] is one of the first generators to create trajectory data. The moving objects change direction, speed and size by parameterized random functions. An extension of the original work [48] implemented three major enhancements: agility that provides for movement to be weighted towards a particular direction, clustered movement that groups of trajectories with similar patterns, and infrastructure that are obstacles that constrain movement.

2.3.1.2 G-TERD

A framework called G-TERD [46] (generator of time-evolving regional data) provides for complex 2-D regional objects such as their color, maximum speed, the influence of other moving objects and statistical distributions for each change factor. The major limitation of G-TERD is that objects move absent of an underlying road network, and therefore are unrealistic in most real-world scenarios.
2.3.1.3 Oporto

A generator based on a spatio-temporal model of fishing boats was the motivation for Oporto [47]. This work is based on attractive and repulsive forces. Fishing boats are attracted to shoals of fish while trying to avoid storm areas. Shoals are themselves attracted by plankton areas. Ships are moving points; plankton or storm areas are regions with fixed center but moving shape; and shoals are moving regions. Although this generator incorporates movement with knowledge of the environment, it fails to force movement based on constraints of an underlying network. Therefore, this model is unsuitable for adaption to modeling roads, pedestrians and other modes of transportation where the movement is along a network.

2.3.2 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 [49]). Most of these simulators, however, are proprietary, do not generate long trajectory histories, and are not easily extensible.

Our benchmarking framework takes advantage of two state-of-the-art open-source trajectory generators: BerlinMOD (Section 2.3.2.3) and Brinkhoff (Section 2.3.2.4).

2.3.2.1 SUMO

SUMO (Simulation of Urban MObility) [50] is an open-source traffic simulator that supports multi-modal transportation and irregular departure times/routes. The transportation modes supported by this simulator include not only car movements within the city, but also public transport systems on the street network, including train networks. The simulator is microscopic, which means that the simulation represents a single discrete atomic unit of movement (in this case a single human being), in contrast to a macroscopic simulation that represents flows of movement through a network. Another major goal of this simulator is to address irregular departure times and routes which can make up approximately 60% of trips.
2.3.2.2 SMARTEST

The European Union’s 4th Framework Program funded the SMARTEST (Simulation Modeling Applied to Road Transport European Scheme Tests) project [49] to review existing micro-simulation traffic simulation models. The review compared 58 different micro-simulators. The goals varied substantially, from road designs, management strategies and evaluation of intelligent transportation systems. The primary objective of SMARTEST was to identify “gaps” in existing micro-simulation models, but which merited further study [51].

All these generators primarily targeted traffic planning applications, by determining general traffic flow based on observation of individual vehicles. They do not consider directions of travel, such as generating starting locations and destinations for trips, or long time observations of single moving objects. Also, many of these systems are not free, requiring the payment of license fees, and prohibiting researchers from reading or changing the implementation to suit their needs.

2.3.2.3 BerlinMOD

BerlinMOD provides moving object data (MOD) obtained by simulating trips around the Berlin metropolitan area [20]. 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.

2.3.2.4 Brinkhoff

Similar to BerlinMOD, the Brinkhoff trajectory generator [21] 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.

2.3.3 Discussion

The free-moving trajectory generators have the advantage of efficiently produc-
ing trajectories without any restriction on the variation of speed and direction. Due
to the absence of an underlying network, however, free-moving generators have lim-
itations 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 trajec-
tory generator which uses a movement model obtained by analyzing real trajectory
data (refer to Section 4.2.2 for further details). This generator is incorporated into
our benchmark framework (Section 4.1).

The Brinkhoff generator is superior to other generators in terms of its customiza-
tion 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 du-
urations (Section 4.2.1). 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 4.2.3.
CHAPTER 3
Spatial QUALity Simplification Heuristic (SQUISH)

A key contribution of our research is a novel approach called SQUISH for compressing trajectories. Given $\lambda$, the target compression ratio, our previous SQUISH algorithm [14] compresses a trajectory of length $n$ into a trajectory of length $\frac{n}{\lambda}$. This algorithm enables highly accurate compression of trajectories in a substantially shorter time than other techniques [14]. However, it lacks the capability of compressing trajectories while ensuring that SED error is within a user-specified bound. We developed a new version of SQUISH which overcomes this limitation. Algorithmic details and the correctness of this version are explained in Section 3.1 and 3.2, respectively. Section 3.3 discusses the novelty and unique benefits of this version.

3.1 Algorithm Description

Our new SQUISH algorithm requires a trajectory $T$ to compress, and two additional parameters $\lambda$ and $\mu$. It compresses trajectory $T$ while striving to minimize SED error and achieving the compression ratio of $\lambda$. Then, it further compresses $T$ as long as this compression will not increase SED error beyond $\mu$. Therefore, setting $\mu$ to 0 causes this algorithm to minimize SED error ensuring the compression ratio of $\lambda$. In this thesis, this case is referred to as SQUISH($\lambda$). SQUISH($\mu$) denotes another case where $\lambda$ is set to 1 and therefore SQUISH maximizes compression ratio while keeping SED error under $\mu$.

The key idea of SQUISH is to use a priority queue $Q$, where the priority of each point is defined as an upper bound on the SED error that the removal of that point would introduce. Therefore, SQUISH can find and remove a point from $Q$ that has the lowest priority (i.e., a point whose removal would increase SED error within the lower bound) in $O(\log |Q|)$ time, where $|Q|$ denotes the number of points stored in $Q$. By removing points in this way, SQUISH can effectively limit the growth of SED error.
Algorithm 1: SQUISH($T, \lambda, \mu$)

- **input**: trajectory $T$, lower bound $\lambda$ on compression ratio, upper bound $\mu$ on SED
- **output**: trajectory $T'$

1. $\beta \leftarrow 4; \quad // \text{the initial capacity of } Q \text{ is 4}$
2. for each point $P_i \in T$ do
3. \hspace{1em} if $i \lambda \geq \beta$ then
4. \hspace{2em} $\beta \leftarrow \beta + 1; \quad // \text{increase the capacity of } Q$
5. \hspace{1em} set_priority($P_i, \infty, Q$); \quad // enqueue $P_i$ with the priority of $P_i$ being $\infty$
6. \hspace{1em} $\pi[P_i] \leftarrow 0;$
7. \hspace{1em} if $i > 1$ then \quad // $P_i$ is not the first point
8. \hspace{2em} succ[$P_i-1$] $\leftarrow P_i; \quad // \text{register } P_i \text{ as } P_i-1 \text{'s closest successor}$
9. \hspace{2em} pred[$P_i$] $\leftarrow P_i-1; \quad // \text{register } P_i-1 \text{ as } P_i \text{'s closest predecessor}$
10. \hspace{1em} adjust_priority($P_i-1, Q, \text{pred, succ, } \pi$); \quad // Algorithm 3
11. \hspace{1em} if $|Q| = \beta$ then \quad // $Q$ is full
12. \hspace{2em} reduce($Q, \text{pred, succ, } \pi$); \quad // Algorithm 2
13. $p \leftarrow \min\text{priority}(Q); \quad // \text{find the lowest priority from } Q$
14. while $p \leq \mu$ do \quad // the lowest priority is not higher than $\mu$
15. \hspace{2em} reduce($Q, \text{pred, succ, } \pi$); \quad // Algorithm 2
16. \hspace{2em} $p \leftarrow \min\text{priority}(Q); \quad // \text{find the lowest priority from } Q$
17. return trajectory $T'$ comprising the points in $Q$ in the order reflected in the succ map;

<table>
<thead>
<tr>
<th>variable</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q$</td>
<td>priority queue</td>
</tr>
<tr>
<td>$\beta$</td>
<td>capacity of $Q$</td>
</tr>
<tr>
<td>$\text{pred}$</td>
<td>map storing, for each $P_i \in Q$, $P_i$'s closest predecessor among the points in $Q$</td>
</tr>
<tr>
<td>$\text{succ}$</td>
<td>map storing, for each $P_i \in Q$, $P_i$'s closest successor among the points in $Q$</td>
</tr>
<tr>
<td>$\pi$</td>
<td>map storing, for each $P_i \in Q$, the maximum of the priorities that the neighboring points of $P_i$ had when they were removed from $Q$</td>
</tr>
</tbody>
</table>

Table 3.1: Variables used in Algorithm 1

Algorithm 1 provides a detailed description of the SQUISH algorithm. Table 3.1 summarizes the variables used in Algorithm 1. Figure 3.1 illustrates the operation of SQUISH using an example. In SQUISH, variable $\beta$ stores the capacity of priority queue $Q$. The initial value of $\beta$ is 4 (line 1 in Algorithm 1) so that $Q$ can store the first four points (e.g., $P_1$, $P_2$, $P_3$ and $P_4$ in Figures 3.1(a)). The value of $\beta$ increases whenever $\frac{i}{\lambda} \geq \beta$, where $i$ denotes the number of points retrieved so far from trajectory $T$ (lines 3 and 4). Each point $P_i$ from trajectory $T$ (line 2) is initially assigned a priority of $\infty$ when it is inserted into $Q$ (line 5). If $P_i$ is not the first point
Figure 3.1: Execution of SQUISH: Black dots represent the points kept in priority queue $Q$. The numbers within a pair of parentheses are the coordinate values of the corresponding point. The number after a pair of parentheses is the current priority of the corresponding point.

(line 7), $P_i$ is registered as the closest successor of its previous point $P_{i-1}$ (line 8). $P_{i-1}$ is also registered as the closest predecessor of $P_i$ (line 9). Then, the priority of $P_{i-1}$ is adjusted to the SED error that the removal of $P_{i-1}$ would introduce (line 10). In Figure 3.1(a), the priority of $P_2$ is set to 0.5 since removing $P_2$ will cause the SED with respect to $P_2$ to be 0.5 (i.e., the distance between $P_2$ and $P'_2$ in Figure 3.1(b)). Further details of priority adjustment (Algorithm 3) are provided later in this section.

When $Q$ is full (i.e., contains $\beta$ points), SQUISH reduces $Q$ by removing from $Q$ a point that has the lowest priority (lines 11 and 12 in Algorithm 1). For example, in Figure 3.1(a), SQUISH would remove either $P_2$ or $P_3$ since they have the lowest priority of 0.5 among the four points in $Q$. Reducing $Q$ as above ensures that $Q$ keeps only $\beta - 1$ points out of the points seen so far in a manner that effectively limits the growth of SED error. Once all of the points in $T$ are processed, SQUISH finds the lowest priority $p$ obtained from the priorities of the points remaining in $Q$ (lines 13).
Algorithm 2: reduce(Q, pred, succ, π)

\[
\text{input} : \text{priority queue } Q, \text{ maps } \text{pred}, \text{succ} \text{ and } \pi \text{ (refer to Table 3.1)}
\]

1. \(P_j \leftarrow \text{remove_min}(Q); // \text{point } P_j \text{ which has the lowest priority is removed from } Q\)
2. \(\pi[\text{succ}[P_j]] \leftarrow \max(\text{priority}(P_j), \pi[\text{succ}[P_j]]); // \text{ensure that } \pi[\text{succ}[P_j]] \geq \text{priority}(P_j)\)
3. \(\pi[\text{pred}[P_j]] \leftarrow \max(\text{priority}(P_j), \pi[\text{pred}[P_j]]); // \text{ensure that } \pi[\text{pred}[P_j]] \geq \text{priority}(P_j)\)
4. \(\text{succ}[\text{pred}[P_j]] \leftarrow \text{succ}[P_j]; // \text{register } \text{succ}[P_j] \text{ as the closest successor of } \text{pred}[P_j]\)
5. \(\text{pred}[\text{succ}[P_j]] \leftarrow \text{pred}[P_j]; // \text{register } \text{pred}[P_j] \text{ as the closest predecessor of } \text{succ}[P_j]\)
6. \(\text{adjust_priority}(\text{pred}[P_j], Q, \text{pred}, \text{succ}, \pi); // \text{Algorithm 3}\)
7. \(\text{adjust_priority}(\text{succ}[P_j], Q, \text{pred}, \text{succ}, \pi); // \text{Algorithm 3}\)
8. remove the entry for \(P_j\) from \(\text{pred}, \text{succ}, \text{ and } \pi; // \text{garbage collection}\)

If \(p\) is not higher than \(\mu\) (i.e., removing a point which has the lowest priority will not increase SED error beyond \(\mu\)), SQUISH reduces \(Q\) by removing a point which has the lowest priority (lines 14 and 15). This process of reducing \(Q\) is repeated until every point remaining in \(Q\) has a priority higher than \(\mu\).

Algorithm 2 describes the details of reducing priority queue \(Q\). In this process, a point \(P_j\) which has the lowest priority is removed from \(Q\) (line 1). Next, the priority of \(P_j\) is used to update \(\pi[\text{succ}[P_j]]\) and \(\pi[\text{pred}[P_j]]\) (lines 2 and 3). For each point \(P_k \in Q\), \(\pi[P_k]\) stores the maximum of the priorities that the neighboring points of \(P_k\) had when they were removed from \(Q\). If none of \(P_k\)'s neighboring points has been removed, \(\pi[P_k] = 0\) (line 6 in Algorithm 1). For example, in Figure 3.1(a), \(P_2\) has the lowest priority of 0.5. Therefore, removing \(P_2\) causes the values of \(\pi[P_1]\) and \(\pi[P_3]\) to change from 0 to 0.5. As explained below, maintaining \(\pi[P_k]\) as above for each point \(P_k \in Q\) allows SQUISH to derive an upper bound on the SED error that the removal of \(P_k\) would introduce (a formal proof is provided in Section 3.2).

In addition to updating \(\pi[\text{succ}[P_j]]\) and \(\pi[\text{pred}[P_j]]\) as above, SQUISH registers \(\text{succ}[P_j]\) as the new closest successor of \(\text{pred}[P_j]\) (lined 4 in Algorithm 2) and registers \(\text{pred}[P_j]\) as the new closest predecessor of \(\text{succ}[P_j]\) (line 5). In Figure 3.1(b), due to the removal of \(P_2\), the closest successor of \(P_1\) becomes \(P_3\) (i.e., \(\text{succ}[P_1] = P_3\)) and the closest predecessor of \(P_3\) becomes \(P_1\) (i.e., \(\text{pred}[P_3] = P_1\)).
Algorithm 3: adjust_priority($P_j, Q, \text{pred, succ, } \pi$)

\begin{verbatim}
input : point $P_j$, priority queue $Q$, maps pred, succ and $\pi$ (refer to Table 3.1)
\begin{align*}
1 & \text{if } \text{pred}[P_j] \neq \text{null and succ}[P_j] \neq \text{null then} \\
2 & \quad p \leftarrow \pi[P_j] + \text{SED}(P_j, \text{pred}[P_j], \text{succ}[P_j]); \\
3 & \quad \text{set_priority}(P_j, p, Q);
\end{align*}
\end{verbatim}

of $\text{pred}[P_j]$ and that of $\text{succ}[P_j]$ are adjusted (lines 6 and 7 in Algorithm 2) using Algorithm 3. The reason for this adjustment is that the removal of $P_j$ affects the SED error with respect to $\text{pred}[P_j]$ and $\text{succ}[P_j]$. For example, in Figure 3.1(a), the calculation of $P_3$'s priority takes into account line segment $P_2P_4$. After removing $P_2$ (Figure 3.1(b)), that calculation needs to involve $P_1P_4$ rather than $P_2P_4$.

Algorithm 3 shows how SQUISH adjusts the priority of point $P_j$. If $P_j$ is the first point (i.e., $\text{pred}[P_j] = \text{null}$) or the last point (i.e., $\text{succ}[P_j] = \text{null}$) among the points in $Q$ (line 1), the priority of $P_j$ remains at its initial value $\infty$ (line 5 in Algorithm 1). Otherwise, the priority of $P_j$ is set to a new value $p$, which is the sum of $\pi[P_j]$ (i.e., the maximum of the priorities that $P_j$'s neighboring points had when they were removed) and the SED between $P_j$ and line segment $\text{pred}[P_j], \text{succ}[P_j]$ (line 2 in Algorithm 3). For instance, when point $P_2$ whose priority is 0.5 in Figure 3.1(a) is removed (Figure 3.1(b)), $\pi[P_3]$ becomes 0.5 and $\text{SED}(P_3, P_1, P_4)$ becomes the distance between $P_3(2, 5, 2)$ and $P'_3(2, 4, 2)$, which is 1. For this reason, the priority of $P_3$ is set to $0.5 + 1 = 1.5$. After obtaining $p$ as above, point $P_j$ is first removed from $Q$ and then inserted into $Q$ with priority $p$ (line 3). Section 3.2 provides a formal proof that $p$ is an upper bound on the SED that the removal of $P_j$ would introduce (Lemma 3.1).

3.2 Correctness of SQUISH

The $\lambda$ parameter of SQUISH indicates a lower bound on the compression ratio. When $\lambda$ is set to 1, the SQUISH algorithm must ensure that the actual SED error introduced during compression is no larger than $\mu$. Theorems 3.1 and 3.2 verify the correctness of this algorithm:

**Theorem 3.1. (Compression Ratio)** Given a trajectory $T$, SQUISH produces a compressed representation $T'$ of $T$ so that $\frac{|T'|}{|T|} \geq \lambda$, where $|T|$ and $|T'|$ denote the
lengths of $T$ and $T'$, respectively.

**Proof.** In Algorithm 1, for $\lambda \geq 1$, $\beta$ is incremented whenever $\frac{i}{\lambda} \geq \beta$ (lines 3 and 4), meaning that $\frac{i}{\lambda} < \beta \leq \frac{i}{\lambda} + 1$ after this step. Whenever $|Q| = \beta$, a point is removed from $Q$ (lines 11-12). More points may also be removed from $Q$ after the end of $T$ is reached (lines 13-16). Since $T'$ consists of the points that ultimately remain in $Q$, $|T'| = |Q| \leq \beta - 1 \leq (\frac{i}{\lambda} + 1) - 1 = \frac{i}{\lambda} = \frac{|T|}{\lambda}$. Therefore, $\frac{|T|}{|T'|} \geq \lambda$.

**Lemma 3.1.** After SQUISH reduces $Q$ (lines 12 and 15 in Algorithm 1) using Algorithm 2, for each $P_j \in Q$ such that $1 < j < i$ and for each point $P_k$ between $\text{pred}[P_j]$ and $\text{succ}[P_j]$ in the original trajectory $T$,

$$\text{priority}(P_j) \geq \text{SED}(P_k, \text{pred}[P_j], \text{succ}[P_j]) \quad (3.1)$$

where $\text{priority}(P_j)$ denotes the priority of point $P_j$ and $\text{SED}(P_k, \text{pred}[P_j], \text{succ}[P_j])$ denotes the SED between $P_k$ and the line segment joining $\text{pred}[P_j]$ and $\text{succ}[P_j]$.

**Proof.** We prove this lemma by induction as follows:

**Base case.** Suppose that both $P_{j-1}$ and $P_{j+1}$ are in $Q$, meaning that $\text{priority}(P_j)$ did not change after it was adjusted to $\pi[P_j] + \text{SED}(P_j, \text{pred}[P_j], \text{succ}[P_j]) = 0 + \text{SED}(P_j, P_{j-1}, P_{j+1})$ (line 2 in Algorithm 3 and line 10 in Algorithm 1). In this case, (3.1) holds.

**Induction step.** If $P_{j-1}$ is not in $Q$, then let $P_a$ be $\text{pred}[P_j]$, and $P_b$ be the most recently removed point among the points between $P_a$ and $P_j$ in the original trajectory $T$ (Figure 3.2). Also, assume, as an induction hypothesis, that when $P_b$ was removed,
priority($P_{b}$) $\geq SED(P_{k}, pred[P_{b}], succ[P_{b}])$ for each $P_{k}$ between $pred[P_{b}] = P_{a}$ and $succ[P_{b}] = P_{j}$ in $T$ (Figure 3.2). Let $P'_{k}$ denote the estimation of $P_{k}$ in the case of SED (Section 2.1.1) along line segment $P_{a}P_{j}$. Then, $\pi[P_{j}] = \pi[succ[P_{b}]] \geq priority(P_{b})$ (line 2 in Algorithm 2) and $priority(P_{b}) \geq SED(P_{k}, pred[P_{b}], succ[P_{b}]) = d(P_{k}, P'_{k})$. Therefore, $\pi[P_{j}] \geq d(P_{k}, P'_{k})$ (i). Let $P''_{k}$ and $P''_{j}$ represent the estimation of $P_{k}$ and that of $P_{j}$ along $P_{a}P_{j}$. Then, $SED(P_{j}, pred[P_{j}], succ[P_{j}]) = d(P_{j}, P''_{j}) > d(P'_{k}, P''_{k})$ since the angle between $P_{a}P''_{k}$ and $P_{a}P''_{j}$ and the angle between $P_{a}P'_{k}$ and $P_{a}P'_{j}$ are the same, $P'_{k}$ and $P''_{k}$ divide $P_{a}P'_{j}$ and $P_{a}P''_{j}$, respectively, in the same proportion by the definition of SED (Section 2.1.1), and $P_{a}P'_{j}$ is longer than $P_{a}P''_{j}$. At this point, due to line 2 in Algorithm 3, $priority(P_{j}) = \pi[P_{j}] + SED(P_{j}, pred[P_{j}], succ[P_{j}]) \geq d(P_{k}, P'_{k}) + d(P'_{k}, P''_{k})$ by (i) and (ii). Furthermore, $d(P_{k}, P'_{k}) + d(P'_{k}, P''_{k}) \geq d(P_{k}, P''_{k}) = SED(P_{k}, pred[P_{j}], succ[P_{j}])$ by triangle inequality. Therefore, (3.1) holds for each $P_{k}$ between $pred[P_{j}]$ and $P_{j}$ in $T$ (iii).

If $P_{j+1}$ is not in $Q$, then it can also be proven as in the derivation of (iii) that (3.1) holds for each $P_{k}$ between $P_{j}$ and $succ[P_{j}]$ in $T$ (iv). Since $priority(P_{j}) = \pi[P_{j}] + SED(P_{j}, pred[P_{j}], succ[P_{j}])$ (line 2 in Algorithm 3) and $\pi[P_{j}] \geq 0$ (line 6 in Algorithm 1 and lines 2 and 3 in Algorithm 3), (3.1) holds when $P_{k}$ is $P_{j}$ (v). By (iii), (iv), and (v), for each $P_{k}$ between $pred[P_{j}]$ and $succ[P_{j}]$ in $T$, (3.1) holds.

**Theorem 3.2. (SED Error)** Suppose that SQUISH compresses a trajectory $T$ into $T'$ with $\lambda$ set to 1. Then, $SED_{T'}(P_{i}) \leq \mu$ for every $P_{i} \in T$, where $SED_{T'}(P_{i})$ denotes the SED of $T'$ with respect to point $P_{i}$.

**Proof.** For $\mu \geq 0$ and for each point $P_{i} \in T'$, $SED_{T'}(P_{i}) = 0 \leq \mu$ (Section 2.1.1). For an arbitrary point $P_{i} \in T - T'$, let $P_{j}$ denote the most recently removed point such that $P_{i}$ was between $pred[P_{j}]$ and $succ[P_{j}]$. Then, $SED_{T'}(P_{i}) = SED(P_{i}, pred[P_{j}], succ[P_{j}]) \leq priority(P_{j})$ when $P_{j}$ was removed (Lemma 3.1) and $priority(P_{j}) \leq \mu$ (line 14 in Algorithm 1). Therefore, $SED_{T'}(P_{i}) \leq \mu$. 

28
(a) With $\lambda = 5$, SQUISH($\lambda$) achieves a compression ratio of 5.

(b) With $\lambda = 5$, $\mu = 10$ (meters), SQUISH($\lambda$, $\mu$) achieves a compression ratio higher than 5.

Figure 3.3: Benefit of SQUISH($\lambda$, $\mu$)

3.3 Discussion

A major benefit of the SQUISH algorithm is that it provides a provable guarantee on the SED error introduced during compression. It also has the benefit of supporting two special modes, namely SQUISH($\lambda$) which minimizes SED error while achieving the compression ratio of $\lambda$ and SQUISH($\mu$) which maximizes compression ratio while limiting SED error under $\mu$. The previous version of SQUISH [14] is similar to SQUISH($\lambda$) in nature, but different in that it lacks guarantees on SED error. Furthermore, the SQUISH algorithm overcomes the previous version’s limitation of ignoring compression opportunities once it achieves the compression ratio of $\lambda$. For instance, the previous SQUISH algorithm always compresses a trajectory $T$ into another trajectory whose length is $|T|/\lambda$ even if the input trajectory $T$ exhibits little variations in speed and heading. On the other hand, setting $\mu$ to a tolerable error value allows SQUISH to achieve a much higher compression ratio through additional removals which do not increase SED error beyond $\mu$ (Figure 3.3). Instead of using a fixed size priority queue which is constructed only after the length of the input trajectory and target compression ratio are given; SQUISH dynamically increases the size of the priority queue and thus can be used for trajectories that are provided in the form of streaming data.

The details of the SQUISH algorithm, such as updating variables and adjusting the priorities of points are shown in Algorithms 1, 2, and 3. These technical details
(Section 3.1) and the formal proofs on the correctness of SQUISH (Section 3.2) are key contributions of this dissertation.

Additionally, SQUISH has unique advantages over previous trajectory compression algorithms. First, SQUISH has the flexibility of controlling compression with respect to both compression ratio and SED error. Second, SQUISH enables fast compression. The operation of removing a point of the lowest priority from $Q$ takes $O(\log |Q|)$ time, where $|Q|$ is the number of points in $Q$. Given a trajectory $T$ of length $n$, the running time and space overhead of SQUISH($\lambda$) are $O(n \log \frac{n}{\lambda})$ and $O(\frac{n}{\lambda})$ since $|Q| \leq \frac{n}{\lambda}$. In the case of SQUISH($\mu$), the running time and space overhead are $O(n \log n)$ and $O(n)$, respectively, since $|Q| \leq n$. In addition to the above benefits, SQUISH achieves low SED error since it removes points whose removal will increase SED within the lowest bound. These characteristics of SQUISH are summarized in Table 2.1 and experimentally demonstrated in Section 5.
CHAPTER 4
Benchmarking Framework

4.1 Introduction to Framework

This section presents a benchmarking framework for comparing trajectory compression algorithms. Section 4.1.1 provides an architectural overview of the framework. Sections 4.1.2 and 4.1.3 explain the implementation details of trajectory compression algorithms and evaluation metrics.

4.1.1 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 4.1 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 2.1.1). Then these values are stored altogether as a record in a database table (Figure 4.1). To execute the benchmark, a user only needs to submit a configuration file (e.g., ratio10.cfg in Figure 4.1) 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 4.2 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 4.1) for benchmarking. These trajectories can be produced by using the generators described in Section 4.2. In addition to the above input directory, the con-
Figure 4.1: Architecture of Benchmarking Framework

Figure 4.2: Example Configuration File

The aforementioned DefaultExecutor and GuaranteedCompressionRatioExecutor modules are for accurately evaluating compression algorithms on a fair basis despite their differences. For examples, algorithms such as SQUISH(λ) and Uniform Sampling compress trajectories while guaranteeing a specific compression ratio (Table 2.1). In contrast, algorithms such as SQUISH(μ), Douglas-Peucker, and TD-TR strive to max-
imize compression ratio under a certain accuracy constraint. The DefaultExecutor on line 6 in Figure 4.2, 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 4.1). 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 4.2).

4.1.2 Integration of Compression Algorithms

The 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.

4.1.3 Integration of Evaluation Metrics

The benchmarking framework supports all of the evaluation metrics summarized in Section 2.1. The actual code for the accuracy metrics in Section 2.1.1 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 5.2 shows our implementation of the SED metric.

The 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 4.6 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.

### 4.2 Synthetic 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 4.2.1), our Gaussian trajectory generator which can significantly vary speed and heading (Section 4.2.2), and the BerlinMOD generator which takes advantage of a mature road network model.
4.2.1 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 [52].

4.2.2 Developed 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 [15] is shown in Figures 4.7 and 4.8, 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 \((\Phi, \Lambda)\) in a trajectory, the Gaussian generator determines the next location \((\Phi', \Lambda')\) using parameters that define changes in speed and heading. These parameters include \(\mu_{\text{speed}}\) and \(\mu_{\text{heading}}\), which denote the mean of the changes in speed and in heading, respectively. Additional parameters are the
standard deviation of the changes in speed and heading, which are denoted as $\sigma_{\text{speed}}$ and $\sigma_{\text{heading}}$ respectively. The current speed ($\Delta$) and heading ($\Theta$) are determined by randomization formula described below:

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

$$\Theta \leftarrow \Theta + \text{random()} \cdot \sigma_{\text{heading}} + \mu_{\text{heading}}$$
Then, the next location \((\Phi', \Lambda')\) 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 (\(\Delta\)) 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.

### 4.2.3 Integrated BerlinMOD Generator

The BerlinMOD trajectory generator uses a parameter called the scale factor \(\Upsilon\), which determines the duration of the simulation in terms of the number of simulation days. For our experiments, \(\Upsilon\) 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 car 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 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.
4.3 Discussion

This chapter introduced a new benchmarking framework that allows users to conveniently and efficiently evaluate GPS trajectory 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 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. A comprehensive overview of trajectory compression algorithms and metrics for evaluating them were provided in Chapter 2. The trade-off and appropriate use cases are discussed in detail according to actual evaluation results from the benchmarking framework in the next chapter.
CHAPTER 5

Evaluation

This chapter experimentally evaluates algorithms that compress trajectories by removing a subset of points from them. Our evaluation does not include STTrace [17] and Bellman’s algorithm [27] which introduce larger errors despite longer running times compared to TD-TR [8] (refer to Section 2.2.6 for further details). The techniques that are evaluated in this chapter have unique benefits in balancing compression time and the degree of error (Table 2.1).

Section 5.2.1 describes the data sets used for evaluating trajectory compression algorithms. Sections 5.1.1 and 5.2.2 compare the trajectory compression algorithms in terms of compression time (Section 2.1.2) and accuracy/error metrics (Section 2.1.1), respectively. Section 5.2.3 presents application-specific recommendations on choosing trajectory compression algorithms.

Figure 5.1: Average Compression Time
5.1 Evaluation Based on Performance Metrics

5.1.1 Evaluation Based on Compression Times

Figure 5.1 depicts the actual execution times of each trajectory compression algorithm. This experiment used 71 trajectories, each containing a minimum of 20,000 points. To measure the effect of the trajectory size on compression time, we obtained shorter trajectories by taking the first 5,000, 10,000, 15,000, and 20,000 points from each of the original trajectories. To compare the algorithms on a fair basis, the compression ratio was set to 10 for all of these algorithms. However, most algorithms including Douglas-Peucker, Dead-Reckoning, Opening Window and SQUISH(\(\mu\)) do not take a target compression ratio as an input parameter (Table 2.1). Therefore, we used an approach which repeatedly executed such an algorithm, modifying the error bound parameter as in binary search, until the desired compression ratio was achieved. Once a relevant error bound is found, to obtain reliable results on compression time despite short compression runs, we repeatedly ran each algorithm on the same trajectory using the same parameter values until the accumulated execution time became greater than 1 second. Then, we averaged the execution times of these iterations to obtain the compression time for that combination of algorithm and trajectory. In this evaluation study, we used one core of a Xeon E5430 2.67 GHz CPU for each run.

In Figure 5.1, the fastest algorithm is Uniform Sampling which compressed trajectories consisting of 20,000 points within 36 microseconds on average. For the same trajectories, the average execution time of TD-TR, the slowest among the 8 algorithms, was 85 milliseconds, which was more than 2,000 times longer than that of Uniform Sampling. As Figure 5.1 shows, TD-TR incurs significantly higher computational overhead than Douglas-Peucker although they are identical except for the error metrics that they use. The reason behind this difference is that SED error (TD-TR) requires more computation than spatial error (Douglas-Peucker) particularly due to the Haversine distance calculation between two points on a sphere [53]. For the same reason, OPW-TR is also substantially slower than Opening Window. The benefits of using SED error with respect to error metrics are experimentally demonstrated in Section 5.2.2. In our result, Dead Reckoning and Opening Window were the fastest.
algorithms after Uniform Sampling. Dead Reckoning has linear time overhead like Uniform Sampling (Table 2.1). In theory, Opening Window may exhibit quadratic time overhead (Table 2.1). However, it demonstrated a substantially lower overhead in our study. Despite this speed benefit, Uniform Sampling, Dead Reckoning, and Opening Window have fundamental limitations in controlling the growth of errors during compression (Section 5.2.2).

Both SQUISH(λ) and SQUISH(μ) demonstrate significantly higher compression speed than other algorithms that use SED. In particular, they are approximately 4-6 times faster than TD-TR and often faster than Douglas-Peucker which uses a less expensive error metric (spatial error). This benefit in compression speed is due to the use of a priority queue which enables both fast and effective removal of points (Section 3.1). Furthermore, whenever a point is removed from the priority queue, SQUISH needs to update information about up to two points. On the other hand, both Douglas-Peucker and TD-TR tend to recalculate, for each addition of a point, error values with respect to a large number of points (Section 2.2.2). SQUISH(μ) is slightly slower than SQUISH(λ) since it can further remove redundant points as long as this removal increases SED error under a tolerable bound.

Although there were significant differences in run-time performance between algorithms, we did not observe the same amongst different datasets (i.e., travel modes). In general, particularly in the case of Uniform Sampling and SQUISH, compression time is affected directly by the trajectory size and compression ratio rather than variations in speed and heading. In contrast, such characteristics of trajectories may have a significant impact on the errors introduced during compression (Table 5.3 and Figure 5.4).

5.2 Evaluation using Real-World Datasets

5.2.1 Real-World Datasets

Table 5.1 summarizes the data sets used for evaluating trajectory compression algorithms. Each data set represents a different transportation mode (multi-modal, bus, and urban commuter).
<table>
<thead>
<tr>
<th>data set</th>
<th>location</th>
<th>mode(s)</th>
<th>trajectories</th>
<th>points</th>
<th>size</th>
<th>sampling rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeoLife</td>
<td>Beijing, China</td>
<td>multi-modal</td>
<td>6,923</td>
<td>12,847</td>
<td>641Mb</td>
<td>1 – 5 seconds</td>
</tr>
<tr>
<td>Bus</td>
<td>Albany, NY</td>
<td>bus</td>
<td>52</td>
<td>4,608</td>
<td>7Mb</td>
<td>5 seconds</td>
</tr>
<tr>
<td>NYMTC</td>
<td>New York City</td>
<td>urban commuter</td>
<td>30</td>
<td>2,902</td>
<td>52Mb</td>
<td>5 seconds</td>
</tr>
</tbody>
</table>

Table 5.1: GPS Trajectory Data sets

<table>
<thead>
<tr>
<th>data set</th>
<th>avg(speed)</th>
<th>std(speed)</th>
<th>avg(Δ speed)</th>
<th>std(Δ speed)</th>
<th>std(Δ direction)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeoLife</td>
<td>20.7</td>
<td>18.8</td>
<td>7.5</td>
<td>5.9</td>
<td>27.5</td>
</tr>
<tr>
<td>Bus</td>
<td>15.7</td>
<td>18.4</td>
<td>4.9</td>
<td>7.9</td>
<td>27.8</td>
</tr>
<tr>
<td>NYMTC</td>
<td>10.2</td>
<td>19.2</td>
<td>2.6</td>
<td>3.7</td>
<td>29.9</td>
</tr>
</tbody>
</table>

Table 5.2: Statistics of Data sets

5.2.1.1 Microsoft GeoLife Dataset

The Microsoft GeoLife [54, 55] dataset was obtained from 178 individuals over a period of two years (from April 2007 to August 2009). This dataset includes various transportation modes such as biking, walking, and rail. Approximately 91% of the trajectories have a sampling rate of 1 - 5 seconds and others have higher sampling rates. Most of the data collection occurred around Beijing, China and a small number of trajectories were obtained in the United States and Europe.

To properly evaluate trajectory compression algorithms, we cleaned the GeoLife dataset. For example, we removed every part in the trajectories which was collected while the corresponding individual remained stationary for over 20 minutes. The reason for this removal is to prevent compression results from being skewed by unusually high data redundancy. We also split each trajectory whenever we detected a gap in recorded time which was at least 20 minutes long. Furthermore, points that exhibit unrealistic speed (e.g., 1000 km/hr) were removed from the trajectories. As a result, we obtained a total of 6,923 trajectories containing 12,847 points on average.
<table>
<thead>
<tr>
<th>Dataset</th>
<th>Avg(SED)</th>
<th>Avg(spatial)</th>
<th>Avg(speed)</th>
<th>Avg(heading)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeoLife</td>
<td>75.0</td>
<td>0.066</td>
<td>24.1</td>
<td>25.7</td>
</tr>
<tr>
<td>Bus</td>
<td>33.1</td>
<td>0.050</td>
<td>6.28</td>
<td>42.3</td>
</tr>
<tr>
<td>NYMTC</td>
<td>12.7</td>
<td>0.043</td>
<td>2.39</td>
<td>52.5</td>
</tr>
</tbody>
</table>

Table 5.3: Impact of Datasets on Compression Errors at a Fixed Compression Ratio of 10

5.2.1.2 Albany Bus Transit Dataset

The Bus dataset in Table 5.1 was obtained from buses traveling along four routes in Albany, New York, over a period of 12 weeks (October to December, 2009) [4]. This data set consists of 52 trajectories with a combined total of roughly 240,000 spatio-temporal points collected every 5 seconds.

5.2.1.3 New York Metropolitan Transportation Council Dataset

This dataset consists of trajectories collected at the sampling rate of 5 seconds by 24 volunteers from the New York Metropolitan Transportation Council (NYMTC) [56]. These trajectories reflect the movements of individuals commuting into New York City for work and returning home at the end of the day.

5.2.1.4 Discussion

Each real-world dataset included in this study represents a unique data profile. Travel mode is a crucial predictor of such profiles. On average, as Table 5.2 shows, the GeoLife dataset has the highest degree of error, followed by the Bus and NYMTK datasets. The GeoLife dataset represents a multitude of different travel modes that occurred in the complex urban environment of Beijing, China. Although filters were applied to clean the dataset, significant inaccuracies in the data are inevitable due to the urban canyon effect and obstructions that affect GPS readings. The Bus dataset represents noticeable changes in speed that took place as buses stopped due to traffic or the arrival at designated stops, or as GPS units inside a bus had an obstructed view of the sky. On the other hand, a large number of trajectories in the NYMTK dataset represent the urban commuter mode which typically involves a relatively constant ride, typically of a car or rail with smooth stops.
Table 5.3 shows how the characteristics of datasets affect the errors introduced during compression. The results in the table were obtained across all of the compression algorithms evaluated in this study. Specific breakdowns of the results per compression algorithm are provided and discussed in Section 5.2.2. Trajectories from the GeoLife dataset are by far the most difficult to compress with high accuracy due to the high average speed and large variations in speed. In contrast, trajectories from the NYMTC dataset were compressed with lowest SED, speed, and spatial errors. Table 5.3 shows that the ranking of accuracy is in general consistent across SED, speed, and spatial errors. On the other hand, NYMTC dataset contains relatively large variations in speed. Compressing trajectories from the NYMTC dataset also resulted in large heading errors.

5.2.2 Evaluation Based on Accuracy Metrics

This section experimentally compares trajectory compression algorithms across multiple metrics including SED, spatial, and speed errors. Figures 5.2 and 5.3 contrast compression algorithms in terms of SED and spatial errors, respectively, for compression ratios 5 through 30. These results are a composition from the three data sets mentioned in Section 5.2.1. In terms of overall accuracy, TD-TR and SQUISH($\mu$) are clearly the most accurate with the differences being insignificant in most cases. TD-TR is slightly more accurate than SQUISH($\mu$) in many cases since it strives to minimize the maximum SED error by taking many points into account. As mentioned in Section 5.1.1, SQUISH is 4-6 times faster than TD-TR because it updates infor-
Figure 5.3: Average Spatial Error

In Figures 5.2 and 5.3, algorithms that are faster than SQUISH (Figure 5.1) have substantial limitations in controlling errors during compression. For example, Uniform Sampling had about 2 times higher SED and spatial errors than SQUISH(µ). Surprisingly, both Dead Reckoning and Opening Window introduced larger SED and spatial errors than Uniform Sampling. These figures also illustrate that SED error has benefits over spatial error. For example, algorithms, such as Douglas-Peucker and Opening Window, that are optimized for spatial error tend to introduce the largest SED errors (Figure 5.2). On the other hand, algorithms which uses SED error, including TD-TR, SQUISH(λ), SQUISH(µ), and OPW-TR keep spatial error at a relatively low level. As explained in Section 2.1.1, SED error has the benefit of incorporating temporal data into error calculation.

The compression algorithms examined in this study attempt to minimize specific error metrics, as indicated in Table 2.1. Douglas-Peucker is optimized for spatial error, while TD-TR, a modification of Douglas-Peucker, is optimized for SED error. Not surprisingly, Douglas-Peucker and TD-TR enabled most accurate compressions in terms of spatial error and SED error, respectively. In the case of Opening Window, a significant improvement can be observed for OPW-TR compared to Opening Window in Figure 5.2.
Figure 5.4: Average SED Error (per Dataset)

Figure 5.5: Ranking in terms of Average SED Error

Figure 5.4 demonstrates significant differences between various trajectory compression algorithms, as well as the datasets explained in Section 5.2.1. For this result, all algorithms were compared at a common compression ratio of 10. On average, the GeoLife dataset had the highest degree of error, followed by the bus and urban commuter dataset respectively (Figure 5.4). The GeoLife dataset represents a multitude of different forms of travel modes that occurred in the complex urban environment of Beijing, China. Although filters were applied to clean the dataset, significant inaccuracies in the data are inevitable due to the urban canyon effect and obstructions that affect GPS readings. Similarly, the bus dataset also contained features that make highly accurate compression difficult. First, GPS units inside a bus have an obstructed view of the sky, causing errors due to horizontal dilution of precision (HDOP). The second reason is that it is common for buses to stop either due to
traffic or due to designated stops. High error, combined with frequent stops, causes random fluctuations and noise to be introduced into the GPS trajectory, resulting in less redundancy and higher measured error between the original and the compressed trajectories. A large number of trajectories in the urban commuter dataset contained rail travel modes. These trajectories are easy to compress due to the relatively small amount of change in speed and direction.

Figure 5.5 provides an additional comparison which uses an ordered ranking of the algorithms in terms of average SED error. A ranking of 1 for a particular trajectory implies that the algorithm had the lowest SED error compared to the other algorithms. Similarly, an algorithm that had a ranking of 8 had the highest SED error compared to other algorithms.

The results presented in Figure 5.5 was obtained over all of the three datasets.
mentioned in Section 5.2.1 and across various compression ratios that ranged from 5 to 30. TD-TR, SQUISH(μ) and SQUISH(λ) are shown to be consistently the most accurate algorithms with a ranking of nearly 1, 2 and 3 respectively. Douglas-Peucker and Opening Window are the worst performers, presumably because they are not optimized for SED errors. Surprisingly, Uniform Sampling had a better average ranking compared to Dead Reckoning.

Figure 5.6 shows the average speed error for each algorithm over various compression ratios. TD-TR and SQUISH(μ) were nearly tied for the most accurate algorithm in terms of speed error. Uniform Sampling has the third best performance. The performance of all of the other algorithms was nearly identical. When comparing maximum speed error, SQUISH(μ) and TD-TR were the best algorithms. The effectiveness of SQUISH(μ) and TD-TR is due to the fact that they use temporal information in deciding which points to remove/add from/to the compressed representation.

Our original implementation of SQUISH allowed significant error propagation at high compression ratios. In particular, it was not be able to provide provable guarantees on SED errors that the removal of points would introduce. Our extended version of SQUISH overcomes the above problems with a different way of adjusting the priority of points. The correctness of this new approach is presented in Section 3.2. Figure 5.7, shows the difference in between these two versions. The new version of SQUISH enables a more tight control over the growth of SED error particularly under high compression ratios.

Figure 5.8 shows the distribution of SED errors for the compression algorithms at a compression ratio of 10. Each curve in the figure depicts the probability that the actual SED error will be less than or equal to the error value. For example, the curve labeled “Douglas-Peucker” illustrates that the average SED error introduced by the Douglas-Peucker is less than 100 meters for about 80% of trajectories that were compressed. This distribution also indicates that there is no significant difference in terms of SED error between TD-TR and SQUISH(μ).
5.2.3 Discussion

In comparing these different algorithms side-by-side, the trade-offs between accuracy and computation time can be evaluated on an equal footing. By understanding how well these compression algorithms work on trajectories corresponding to different transportation modes, application-specific compression can be used to determine the best algorithm for a specific context.

Despite the results indicating significant differences in accuracy amongst the compression algorithms, the clear overall winners are TD-TR and SQUISH. Significant differences are noted previously in Figure 5.4. In this figure, it is observed that Douglas-Peucker, Opening-Window, OPW-TR, and Dead Reckoning were the least accurate on the Bus data set. In contrast, Douglas-Peucker, Opening-Window, OPW-TR and Uniform Sampling were the least accurate on the GeoLife data set. However, the figure clearly indicates that the most accurate algorithms across all three data sets was TD-TR and the two implementations of SQUISH.

The impact of compression on specific applications and queries largely depends on the accuracy of core metrics. Although the overall winners based on the three data sets above are TD-TR and SQUISH, a one-size-fits-all approach is not applicable here. If an application requires high spatial accuracy, Douglas-Peucker or SQUISH would be the best choice. Other considerations would be whether or not the compression is needed on an online fashion. In this case, Uniform Sampling or SQUISH(λ) would be viable options.
Table 5.4: Compression Results for a Compression Ratio of 10 ($n$: number of points in the input trajectory)

<table>
<thead>
<tr>
<th></th>
<th>US</th>
<th>DR</th>
<th>SQ($\lambda$)</th>
<th>SQ($\mu$)</th>
<th>DP</th>
<th>TD-TR</th>
<th>OPW</th>
<th>OPW-TR</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SED (meters)</td>
<td>15.8</td>
<td>14.3</td>
<td>10.2</td>
<td>9.7</td>
<td>33.2</td>
<td>8.6</td>
<td>24.4</td>
<td>13.3</td>
<td>16.2</td>
</tr>
<tr>
<td>Spatial (meters)</td>
<td>5.7</td>
<td>6.0</td>
<td>4.6</td>
<td>4.6</td>
<td>4.1</td>
<td>4.2</td>
<td>6.5</td>
<td>5.6</td>
<td>5.2</td>
</tr>
<tr>
<td>Speed (meters/sec)</td>
<td>25.3</td>
<td>24.4</td>
<td>21.4</td>
<td>19.1</td>
<td>23.8</td>
<td>17.8</td>
<td>22.0</td>
<td>24.4</td>
<td>22.3</td>
</tr>
<tr>
<td>Heading (degrees)</td>
<td>17.5</td>
<td>18.9</td>
<td>14.7</td>
<td>15.4</td>
<td>29.1</td>
<td>12.6</td>
<td>26.2</td>
<td>17.5</td>
<td>17.24</td>
</tr>
<tr>
<td>Compr. Time (ms)</td>
<td>0.6</td>
<td>3.5</td>
<td>59.4</td>
<td>42.4</td>
<td>55.0</td>
<td>151.3</td>
<td>12.8</td>
<td>164.2</td>
<td>62.1</td>
</tr>
<tr>
<td>Memory Usage</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Most applications require a balance between accuracy, performance and usability. Often times, SQUISH would be a strong candidate to meet these characteristics. In terms of accuracy, SQUISH is slightly less accurate in terms of SED and speed error, while slightly more accurate in terms of spatial error. Applications that have very short sampling intervals under long durations might be willing to substitute accuracy for faster processing. Since SQUISH($\lambda$) processes data in an online fashion, and SQUISH($\lambda$, $\mu$) performs the majority of the computation online, the differences in performance is more apparent compared to batch methods. The result is that SQUISH executes much faster than TD-TR. Additionally, SQUISH is able to allow the user to compress to a desired size or compress to within a certain error threshold. In most applications, the user would set both parameters. This fulfills an important niche in the literature, since other algorithms lack this flexibility.

5.3 Benchmark Evaluation

This section presents evaluation results that compare all of the trajectory compression algorithms described in Section 2.2 using data generated from the synthetic data generators. We obtained these results by running our benchmark with a total of nine data sets from the three trajectory generators mentioned in Section 4.2. Each trajectory contained up to 37,000 points. In all of the evaluation cases, a compression ratio of 10 is achieved. Table 5.4 summarizes the characteristics of compression algorithms in terms of the error and performance metrics defined in Section 2.1. Table 5.5 shows the abbreviations used to refer to the compression algorithms in Tables 5.4, 5.6, 5.7, and 5.8.

To highlight the benefits of each compression algorithm relative to the other algorithms, Tables 5.6 and 5.7 present the rankings of these algorithms in terms of
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Algorithm Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>US</td>
<td>Uniform Sampling</td>
</tr>
<tr>
<td>DR</td>
<td>Dead Reckoning</td>
</tr>
<tr>
<td>SQ(\lambda)</td>
<td>SQUISH(\lambda)</td>
</tr>
<tr>
<td>SQ(\mu)</td>
<td>SQUISH(\mu)</td>
</tr>
<tr>
<td>DP</td>
<td>Douglas Peucker</td>
</tr>
<tr>
<td>TD-TR</td>
<td>TD-TR</td>
</tr>
<tr>
<td>OPW</td>
<td>Opening Window</td>
</tr>
<tr>
<td>OPW-TR</td>
<td>Opening Window (SED)</td>
</tr>
</tbody>
</table>

Table 5.5: List of Compression Algorithms

<table>
<thead>
<tr>
<th></th>
<th>US</th>
<th>DR</th>
<th>SQ(\lambda)</th>
<th>SQ(\mu)</th>
<th>DP</th>
<th>TD-TR</th>
<th>OPW</th>
<th>OPW-TR</th>
</tr>
</thead>
<tbody>
<tr>
<td>SED</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>Spatial</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>Speed</td>
<td>8</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>Heading</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Overall</td>
<td>6.0</td>
<td>6.0</td>
<td>3.3</td>
<td>2.3</td>
<td>5.5</td>
<td>1.3</td>
<td>6.5</td>
<td>5.3</td>
</tr>
</tbody>
</table>

Table 5.6: Accuracy Rankings for a Compression Ratio of 10

<table>
<thead>
<tr>
<th></th>
<th>US</th>
<th>DR</th>
<th>SQ(\lambda)</th>
<th>SQ(\mu)</th>
<th>DP</th>
<th>TD-TR</th>
<th>OPW</th>
<th>OPW-TR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation Time</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Memory Usage</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Overall</td>
<td>1.0</td>
<td>1.5</td>
<td>4.5</td>
<td>5.0</td>
<td>5.5</td>
<td>6.5</td>
<td>3.5</td>
<td>6.0</td>
</tr>
</tbody>
</table>

Table 5.7: Performance Rankings for a Compression Ratio of 10

accuracy and performance, respectively. For example, Table 5.6 shows that, among the eight 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 5.6 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(\mu) and SQUISH(\lambda) with rankings of 2.3 and 3.3, respectively. A strong correlation can be observed in Table 5.6 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 2.1.3). Due to these benefits, the remainder of this section considers SED as the representative error metric.

Table 5.7 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(\(\lambda\)) ranks the fourth in overall performance, but achieves much more accurate compression than the above three. Furthermore, both SQUISH(\(\lambda\)) and SQUISH(\(\mu\)) significantly outperform TD-TR, the most accurate algorithm, in terms of speed and memory efficiency at the expense of slightly lower accuracy.

Table 5.8 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(\(\mu\)) and TD-TR achieve an average SED of 0.9 meters for trajectories produced by the Brinkhoff generator in the “Highway” mode.

Table 5.8 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. For this reason, the accuracy results from BerlinMOD in Table 5.8 are consistent with our previous evaluation results which used actual GPS trajectory data [14, 15]. 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.
CHAPTER 6
Conclusion and Future Work

This dissertation presents a new approach to GPS trajectory compression. This approach, called SQUISH (Spatial QUalIty Simplification Heuristic), inserts points from a trajectory into a priority queue. The priority of each point in the queue is set to an upper bound on the error that the removal of that point would introduce. In this way, it can quickly remove points while effectively bounding the growth of error caused by the removal of points. This technique also allows users to control compression while striking a balance between compression ratio and accuracy.

To compare SQUISH and other GPS trajectory compression algorithms, a new benchmarking framework was presented. This framework 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 thesis 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.

Utilizing the benchmark framework, a comprehensive evaluation study was conducted using a combination of real-world and synthetically generated datasets. This evaluation measures the effectiveness of trajectory compression algorithms in terms of performance and various error metrics. SQUISH achieves a very accurate compression in a substantially shorter time than other techniques.

Lossy compression removes noise and identifies critical features, therefore the end result is a concise aggregation of the most important information. Since trajectories often share similar patterns (i.e. similar road segments, travel time) compression
can be applied to compression of multiple trajectories to achieve drastic reduction in data size. This aggregation can be used for data mining and database queries without requiring processing of the entire original dataset. Most spatio-temporal queries and patterns are derived from a large number of trips, therefore expanding this research to multiple trajectories has great potential.

The exponential growth in the collection and use of GPS data has exposed a number of important research opportunities for handling these large datasets. Specifically, compression techniques can be utilized to identify critical features of a trajectory and assist in deriving semantic information. By understanding the context (semantic) information about what individuals are doing, applications can be written that better utilize predictive behavior and individual preferences. Examples of extracted semantic information include: reason for travel, reason for stopping at a particular location and mode of travel (i.e. car, bus, bicycle). Combining spatio-temporal data with additional information, such as sensors and social connections/messages can lead to much more accurate predictive models. Another avenue of future work would be to expand the benchmarking framework to include queries that serve as metrics to measure the accuracy of compression.
BIBLIOGRAPHY


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


