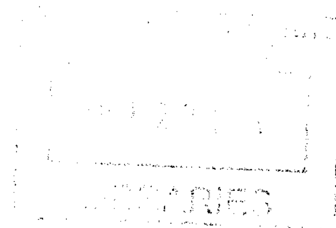
GPSZip: Semantic Representation and Compression System for GPS using Coresets

by

Cathy Wu
B.S., Massachusetts Institute of Technology (2012)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science



at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2013
[June 2013]

© Massachusetts Institute of Technology 2013. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 24, 2013

Certified by ....
Daniela Rus
Professor
Thesis Supervisor

Accepted by ....
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

GPSZip: Semantic Representation and Compression System for GPS using Coresets

by

Cathy Wu

B.S., Massachusetts Institute of Technology (2012)

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2013, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We present a semantic approach for compressing mobile sensor data and focus on GPS streams. Unlike popular text-compression methods, our approach takes advantage of the fact that agents (robotic, personal, or vehicular) perform tasks in a physical space, and the resulting sensor stream usually contains repeated observations of the same locations, actions, or scenes. We model this sensor stream as a Markov process with unobserved states, and our goal is to compute the Hidden Markov Model (HMM) that maximizes the likelihood estimation (MLE) of generating the stream. Our semantic representation and compression system comprises of two main parts: 1) trajectory mapping and 2) trajectory compression. The trajectory mapping stage extracts a semantic representation (topological map) from raw sensor data. Our trajectory compression stage uses a recursive binary search algorithm to take advantage of the information captured by our constructed map. To improve efficiency and scalability, we utilize 2 coresets: we formalize the coreset for 1-segment and apply our system on a small k-segment coreset of the data rather than the original data. The compressed trajectory compresses the original sensor stream and approximates its likelihood up to a provable $(1 + \varepsilon)$-multiplicative factor for any candidate Markov model.

We conclude with experimental results on data sets from several robots, personal smartphones, and taxicabs. In a robotics experiment of more than 72K points, we show that the size of our compression is smaller by a factor of 650 when compared to the original signal, and by factor of 170 when compared to bzip2. We additionally demonstrate the capability of our system to automatically summarize a personal GPS stream, generate a sketch of a city map, and merge trajectories from multiple taxicabs for a more complete map.

Thesis Supervisor: Daniela Rus
Title: Professor

Acknowledgments

I would like to thank Professor Daniela Rus for giving me great motivation, energy and enthusiasm throughout this year. Her ideas and vision guided me and positively affected the outcome of this thesis. I would also like to thank postdoctoral researcher Dr. Dan Feldman for his excellent guidance and help, in particular in the area of coresets.

I would also like to express my gratitude to all of the Distributed Robotics Laboratory (DRL) in CSAIL, MIT for their unconditional support and help in any topic. I am so grateful to call DRL my home. I especially thank Dr. Sejoon Lim, Brian Julian, and Cynthia Sung for their technical guidance, ideas and mentorship.

Finally, I would like to thank my parents, Gody and Jeb, my siblings, Nancy and Joey, and the rest of my family for their absolute support, encouragement, and understanding.

Contents

1 Introduction 15

1.1 Motivation 16

1.2 Main Contributions 19

1.3 Applications 20

1.4 Relation to SLAM 21

1.5 Extension to higher dimensions 22

1.6 Organization 22

2 Semantic representation and compression system overview 23

2.1 Preprocessing 24

2.2 Trajectory mapping 25

2.3 Trajectory compression 25

2.4 Additional features 26

3 Related Work 27

3.1 Coreset for k-segment mean 27

3.2 Trajectory clustering 28

3.2.1 (k, m)-segment mean 28

3.2.2 Map inference 28

3.2.3 Other techniques 29

3.3 Trajectory compression 30

3.3.1 Text-based compression 30

3.3.2 Hidden Markov map matching 31

3.4 SLAM 31

4 Problem Statement **34**

4.1 Definitions 34

4.2 Problem definitions 36

5 Preprocessing **38**

5.1 Coreset for k-segments 38

5.2 Data patcher 39

6 Trajectory mapping **40**

6.1 (k, m)-segment mean algorithm 40

6.2 Optimal algorithm for 2-segment mean 43

6.2.1 Informal overview 43

6.2.2 Coreset for 1-segment mean 44

6.2.3 Algorithm for $(2, m)$-segment mean 46

6.2.4 Operation reduction 48

6.3 Partitioning (expectation) step algorithm 50

6.3.1 Naive algorithm 50

6.3.2 Using 2-segment mean algorithm 51

6.3.3 Extension for parallelism 51

6.4 Parameter estimator 51

6.5 Map construction 56

7 Trajectory compression **59**

7.1 Trajectory compression using map matching 59

7.2 Trajectory decompression 61

8 Experiments **62**

8.1 Overview 62

8.2 Data sets 62

8.3 Processing 64

8.4 Compression quality: trajectory mapping results 66
8.5 Compression size: trajectory compression results 70
8.6 Intuition for trajectory compression 73

9 Conclusion **75**

List of Figures

1-1 A point density map of taxicab GPS points (projected onto lat-lon space). Hints of streets are visible in areas where there are fewer points per area, however in denser city areas (zoomed in area), it is hard to identify streets. Moreover, this is a raw pixel image rather than a map (graph). 17

1-2 *Left*: Input points, shown in (lat,lon,time). *Left Middle*: Segment clustering, fitting k segments to the input points to minimize error. *Right Middle*: Trajectory clustering, fitting k segments to the input points to minimize error, with the additional constraint that we may only use m different segments (indicated by the m colors). *Right* In other words, the projection of a trajectory clustering onto x-y consists only of m segments (m colors). . . 18

1-3 *Left*: Stream of GPS (lat,lon) points over time. Even when the object moves in straight line segments, the input is not smooth due to GPS noise and small movements. *Top Right*: Even after smoothing, the same trajectory projected onto different segments in (lat,lon) space. This is because the trajectory is not repeated exactly the same (e.g. different lane on a road, small deviations in motion). Position stream shown projected onto (lat,lon) dimensions. Our work is in efficiently clustering these repeated line segments. 19

1-4 *Left*: The input is a sequence of position points, i.e. from GPS devices. *Right*: The output is a graph (map) that captures with error parameters the movement of an agent in its environment, i.e. a taxicab in Singapore. 20

2-1 Semantic representation and compression system. The output of the trajectory clustering & map stage is useful for real-time decision making, whereas the output of the trajectory compression stage is useful for data logging. 24

3-1 Dynamic Bayesian network. In our system, our input measurements are GPS points taken over time. Our map M is a connected directed graph whose nodes are features defined by long corridors of space (segments). Our output poses X are a sequence of these segments. 32

5-1 *Left*: Shown here is a raw GPS trajectory (blue) projected onto x-y. The GPS drops are indicated by the black connections between points.*Right*: Using the map of Massachuetts, we performed map matching to guess what roads the agent took between GPS signal gaps, and these data patches are shown in red. As shown, data patching, eliminates most of the black connections. 39

6-1 *(a)* Shows the input, which consists of n points, partitioned into k parts, each with an initial assignment of to one of m clusters. *(b)* We split the input into $\lceil \frac{k}{2} \rceil$ 2-segment mean problems and we show the steps of computation of one of the subproblems in *(c)-(g)*. *(c)* We initialize our right 1-segment coreset in $O(n)$ time to compute the cost for all of the n points. *(d)* We update our right coreset by removing the leftmost point and we update/initialize our left coreset by adding the leftmost point, which takes O(1) time. We compute the cost using the coreset, which costs O(1). *(e)* We update our right coreset by removing its leftmost point and we update our left coreset by adding the next point to its right, which takes O(1) time. We compute the cost using the coreset, which costs O(1). *(f)* We continue in this manner until the right coreset is empty. Now we have n costs. *(g)* Finally, we compute the partition that yields the minimum cost. 52

6-2 The partitioning stage of the (k,m)-segment mean algorithm consists of finding the best partition of the input into k parts given m possible segments to use. Instead of a computationally expensive full search, we shift the partition bars locally. We **parallelize** this process by shifting all the odd numbered partitions at once (red), then the even partitions (blue). Here, we show the k parts of the input; the first few parts contain points for demonstrative purposes (orange and purple). Note that the points stay in place between iterations, but the selected segments are scaled accordingly (to the length of the partition) 53

6-3 Shown is a sample m vs cost curve. The parameter estimator searches for the elbow of the curve by using a trinary search to look for the point farthest away from the reference line (red). Distances are shown in light blue. In this example, $m = 7$ is located at the elbow. 54

6-4 Shown here is the full original indoors trajectory (blue) projected onto x-y, as well as the m segments that form the map (colored segments) for several different m. Notice the trade-off between compression size and compression quality. Smaller m yields larger errors. Our parameter estimator selected $m = 19$. 56

6-5 Shown here is an example of map construction, which is the last step of the trajectory mapping stage. The resulting information-rich map is then used to further compress the input stream. *Top*: The (k, m)-segment mean produced by the trajectory clustering algorithm, when projected onto x-y, consists of m directed segments. *Middle*: The m directed segments (blue edges) from the (k, m)-segment mean form part of the map, but they lie disconnected. The endpoints of the m clusters form the nodes of the graph (purple). *Bottom*: By following the order of the k segments (each of which is one of the m clusters), the transitions between one cluster to the next yields additional auxiliary edges (orange) to the graph that connect the m clusters. The resulting approximation of the real map is a graph with nodes (end points of the segments (purple)) and edges (the segments (blue) and any transitions between the segments that may have occured during the trajectory (orange)). 57

8-1 *Top:* Our quadrotor flying robot equipped with an onboard computer and smartphone collects GPS data in a courtyard. *Bottom:* An omnidirectional ground robot equipped with laptop and scanning laser rangefinder performs SLAM in an academic building. The resulting occupancy grid map and recorded path of the ground robot. 65

8-2 Raw input location stream from the quadrotor in courtyard area (left), the corresponding generated map (center), and a view of the courtyard where the data was collected (right). Repeated movements are captured in the map as a single movement, as shown above, thereby reducing the noise from the original data. 66

8-3 Shown here is the full original trajectory in time (blue), as well as the m segments that form the map (not blue). Paths that are traversed several times are automatically summarized with a single path that represents all the traversals (with bounded error). 67

8-4 Semantic compression of personal GPS trajectory. 67

8-5 Semantic compression of a Singapore taxicab trajectory. 68

8-6 Merging capability of our semantic representation and compression system. The trajectories of two taxis (top) are merged into a single more complete map (bottom). Note the circled area, which is lacking in segments in the top left taxi's trajectory but abundant in the top right taxi's trajectory. Together (bottom), their map is more complete. 69

8-7 *Left:* The average error drops off exponentially as the size of the map increases (for k=200). *Right:* The average error drops off exponentially as the size of the coreset increases. The trend is independent of the choice of m, as shown for k=200. 69

8-8 We compress subsets of the ground robot data set to demonstrate how our system scales over time and to simulate the online operation of our system. *Left*: Growth of input size vs output size as the task progresses. *Right*: Compression ratio trend and growth of map size as task progresses for the ground robot experiment. 73

List of Tables

8.1 Experimental results. Our semantic compression performs consistently better than bzip2 (in size) and may be layered with bzip2 for further compression. The **coreset** column indicates the size of the k-coreset computed in the preprocessing stage, and **comp** indicates the length of the resulting compressed trajectory. The **improvement** metric is the measure of how well semantic+bzip2 compresses as compared to just bzip2, or in other words, how much using our compression on top of bzip2 improves the baseline compression. For smaller error parameters, the improvement will be smaller. For applications that tolerate larger error, the improvement may be drastic. Except for the quadcoptor experiment (trajectory clustering, map construction, and trajectory compression only), the listed **running time** is for the entire semantic representation and compression system, including preprocessing, trajectory mapping (and parameter search), and trajectory compression. 71

8.2 Ground robot: As more of the robot's positions and movement patterns are captured, the compression ratio improves (60%-100%). Our compression also works well for simple paths (first 30%). No k-coreset was computed for this experiment. 72

List of Algorithms

1 km_seg_EM($k, m, input$) 41

2 2_seg_mean(P, M) 47

3 Find_best_m($m_0, m_n, k_0, k_n, input$) 54

4 Find_best_k($m_0, k_0, k_n, input$) 55

5 Make_map($input, k, m$) 58

6 Traj_compress(seq, map) 60

Chapter 1

Introduction

This thesis describes an efficient semantic representation and compression system for sensor streams. Our system efficiently computes topological maps for mobile agents with long-term continuous operation, when the trajectory of the agent includes repeating segments. These agents may be robots performing surveillance tasks, people performing daily routines, or vehicles picking up passengers. A long-duration task gives the mobile agent the opportunity to collect historical data streams from its onboard sensors (e.g. GPS units, smart phones). This data can be useful in real-time decision making but the big challenge is dealing with the large data volumes. This is especially challenging for mapping from higher dimensional data, such as camera and LIDAR data.

One approach to solving the large data problem is to use text-based compression methods, more powerful batteries, better hardware, etc. Instead, we present a semantic representation and compression system that 1) quickly identifies the repeated data in the data stream by line-fitting and clustering and thus greatly reduces the input size to an arbitrary decision making algorithm, and 2) further losslessly compresses the repeated data using the topological map for offline storage. The process of extracting the repeated data in a data stream is called trajectory mapping, and the process of using the topological map to compress a trajectory is called trajectory compression. The compressed trajectory compresses the original sensor stream and approximates its maximum likelihood up to a provable $(1 + \varepsilon)$-multiplicative factor for any candidate Markov model.

In this thesis we describe the semantic approach to mapping, present our semantic rep-

resentation and compression system and its algorithms, show experimental results, and present intuition for compression effectiveness for long-term operation. Abstractly, an agent collects a sequence of location-tagged points over a long period of operation. In subsequent visits to the same location, the recorded points are are observed and recorded within some error parameters. Despite this uncertainty, the agent is able to identify the segments that are repeated as it moves in the environment. Our semantic approach to mapping is applicable for large data streams when the mobile agent has access to location information and repeatedly travels along segments in the trajectory (although the trajectory segments may be traversed in different order).

We present results from a series of experiments, each of which far out-perform bzip2, an industry standard lossless text compression tool. In an outdoors robotic experiment we compressed GPS data collected from a quadrotor robot, producing a 60-fold compression on top of bzip2. In an indoors robotics experiment we used a ground robot with a laser scanner, which localized to a SLAM map for global positioning. The raw data set consisting of $72,000$ points was compressed to a coreset of size 52, along with a small map (170x smaller than just bzip2). We additionally demonstrate the capability of our system to automatically summarize a personal GPS stream, generate a sketch of a city map, and merge trajectories from multiple taxicabs for a more complete map.

1.1 Motivation

Driven by the near-ubiquitous availability of GPS sensors in a variety of everyday devices, the past decade has witnessed considerable interest in the automatic inference and construction of road maps from GPS traces. As shown in Figure 1-1, a point density map of a taxicab GPS signal implies the underlying streets. However, zooming in on a portion of the map only magnifies the noise, where we would hope to see smaller streets. Our work and many related works address the problem of semantic representation (the map) underlying GPS signals. Additionally motivated by the expansion of robotics into everyday life, the problem of generating topological maps for any arbitrary or specialized setting or scope is becoming increasingly important.

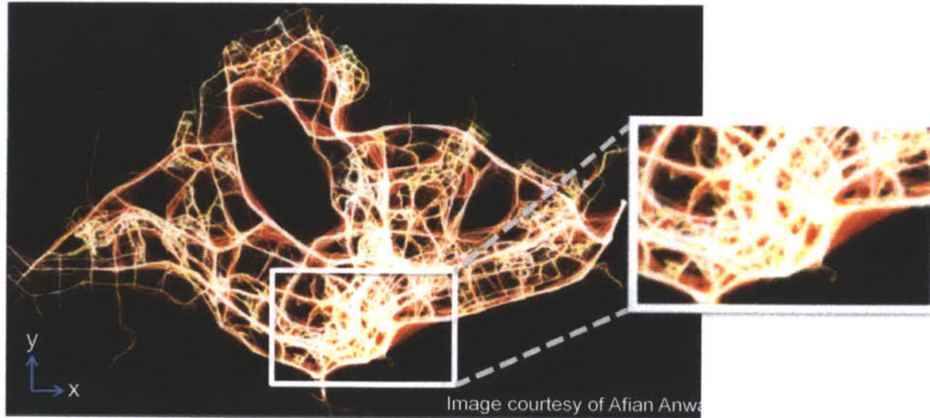


Figure 1-1: A point density map of taxicab GPS points (projected onto lat-lon space). Hints of streets are visible in areas where there are fewer points per area, however in denser city areas (zoomed in area), it is hard to identify streets. Moreover, this is a raw pixel image rather than a map (graph).

The most prevalent topological maps are road maps, and it will be crucial to maintain them for efficient and safe long-term operation of consumer autonomous vehicles. Accurate road maps are crucial for travel efficiency and safety. Existing maps are mostly drawn from geographical surveys. Such maps are updated infrequently due to the high cost of surveys, and thus lag behind road construction substantially. This problem is most prominent in developing countries, where there is often a combination of inferior survey quality as well as booming growth. Even for developed countries, frequent road recongurations and closures often confuse GPS navigation, and have caused fatal accidents even with experienced drivers [34]. As an alternative, OpenStreetMap (OSM) has attracted an online community to manually draw street maps from aerial images and collected GPS traces [1]. Though less expensive, this volunteer-driven process is extremely labor intensive. It also suffers from signicant variability in both the skill level and availability of volunteers in dierent parts of the world.

In other settings, such as within buildings and unmapped outdoors places (e.g. parks, stadiums, courtyards), topological maps simply do not exist, thereby limiting the operation of robots there. Our work is motivated by the difficulty of generating and updating topological maps for arbitrary settings and scopes, in the absense of other maps.

Our work draws inspiration from computational geometry shape-fitting algorithms because mobile agents typically move in straight segments. For example, indoors, people

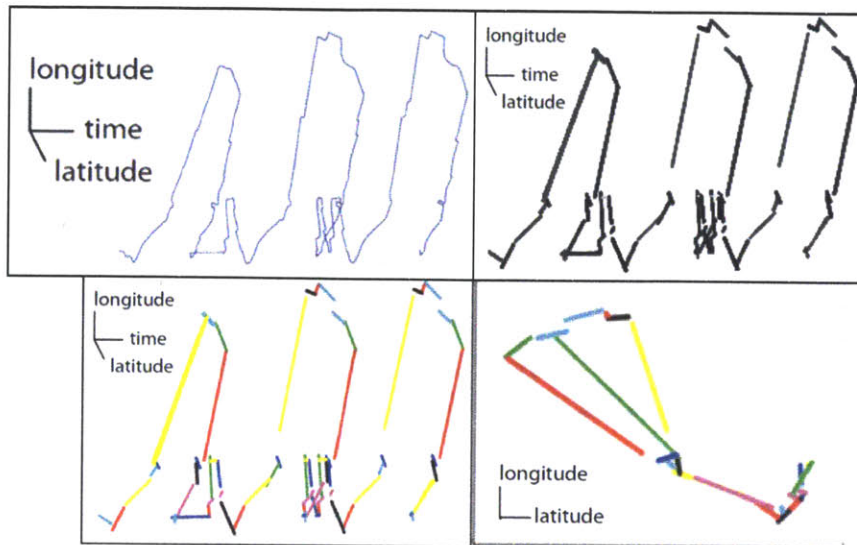


Figure 1-2: *Left*: Input points, shown in (lat,lon,time). *Left Middle*: Segment clustering, fitting k segments to the input points to minimize error. *Right Middle*: Trajectory clustering, fitting k segments to the input points to minimize error, with the additional constraint that we may only use m different segments (indicated by the m colors). *Right* In other words, the projection of a trajectory clustering onto x-y consists only of m segments (m colors).

and robots will follow straight hallways. Outdoors, mobile agents will follow sidewalks or roads. Counter to the case of the traveling salesperson, many applications require mobile agents to visit and revisit locations, resulting in segments that repeat themselves, which lend themselves well to clustering approaches. Additionally, it is known that the additive noise of GPS points is Gaussian [33], which allows us to model the error of our resulting representation of the sensor stream and define the maximum likelihood of a candidate solution based on the input.

We focus on detecting and clustering these repeated segments, called trajectory clustering and then using that representation for trajectory compression. Although both trajectory clustering and segment clustering compute a sequence of segments from a sensor stream, they are different in the following way: Segment clustering (also called the k-segment problem) is the problem of optimally partitioning a sequence into k segments to minimize some fitting cost function. Trajectory clustering similarly partitions a sequence into k segments while minimizing some cost function but has the additional constraint that the k segments, projected onto the space, consists of only m segments, where $m \leq k$. This is called the (k, m)-segment mean problem (defined in Section 4.1). The difference between between

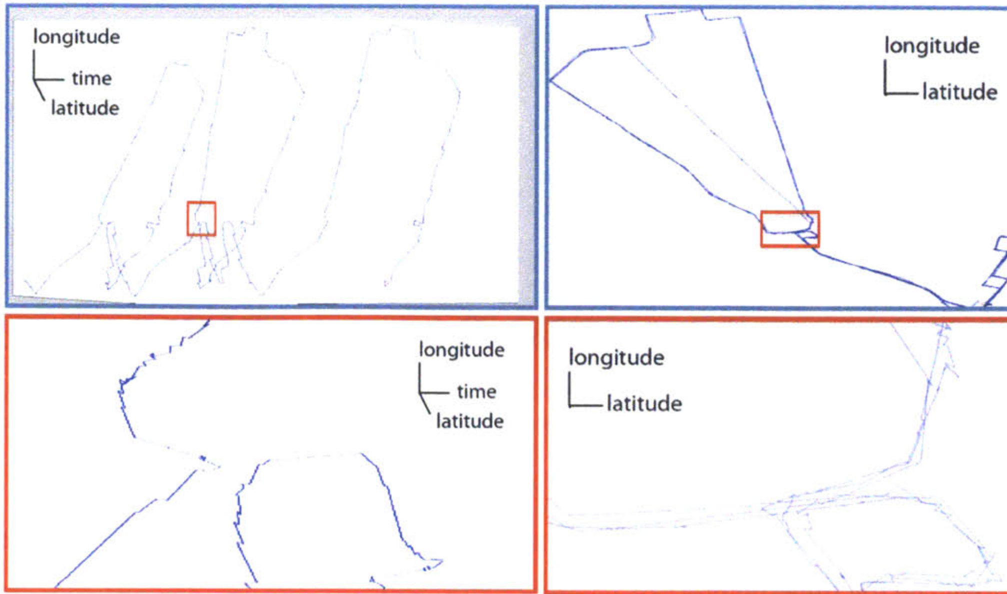


Figure 1-3: *Left*: Stream of GPS (lat,lon) points over time. Even when the object moves in straight line segments, the input is not smooth due to GPS noise and small movements. *Top Right*: Even after smoothing, the same trajectory projected onto different segments in (lat,lon) space. This is because the trajectory is not repeated exactly the same (e.g. different lane on a road, small deviations in motion). Position stream shown projected onto (lat,lon) dimensions. Our work is in efficiently clustering these repeated line segments.

trajectory clustering and segment clustering is illustrated in Figure 1-2, and motivation for this problem is shown in Figure 1-3.

1.2 Main Contributions

The main contributions of this work are as follows:

1. **Trajectory mapping**: Linear-time algorithm for (k, m)-segments mean problem in the number of input points with local optimum guarantees. See Figure 1-3 for intuition on the trajectory clustering problem.

2. **Trajectory compression**: Novel technique for reducing the redundant information in a trajectory, by utilizing information in a topological map.

3. A practical **system-level implementation** of both above modules, which takes as input a sequence of timestamped position points from an agent and produces as output a graph (map) and a compressed trajectory of the agent (see Figure 1-4).

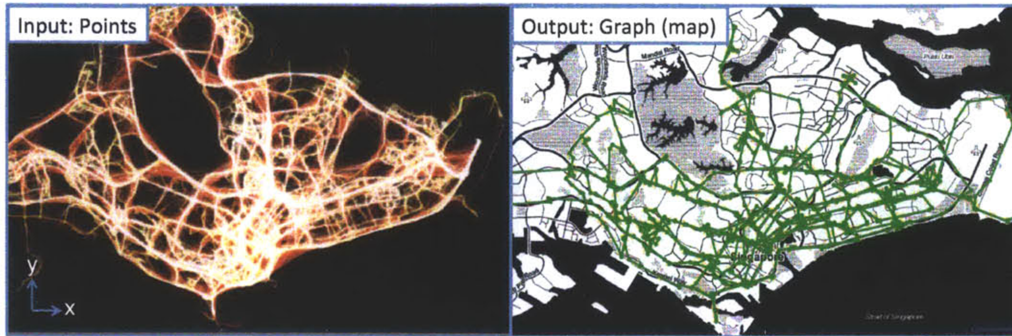


Figure 1-4: *Left*: The input is a sequence of position points, i.e. from GPS devices. *Right*: The output is a graph (map) that captures with error parameters the movement of an agent in its environment, i.e. a taxicab in Singapore.

4. Building- and city-scale **experimental results** for constant frequency position streams (i.e. (lat,lon,time) points).

1.3 Applications

The semantic representation that our system produces is useful for building topological maps in the absence of other maps or for real-time decision making in a variety of mobile applications, including updating maps, incident reporting and monitoring for patrolling, building tours, automated escorts, travel anomalies, road closure detection, and prediction and pattern recognition.

Surveillance and patrolling: For example, a security robot patrolling a museum at night logs locations, actions, and events within a known floor plan. In addition, this robot needs to act in realtime on the incoming sensor streams, relying on the interpretations of historical events. Typically the mobile agent first processes these streams online for decision making using our semantic compression algorithm, then employs lossless compression techniques for data logging. To revisit and reprocess historical data, the log is first decompressed back into its semantic (losslessly) or original raw form (lossy with bounded error) (See Figure 2-1).

Building and updating topological maps: In an unknown environment, for example parks or indoors, where topological maps are not available, generating a map automatically from a raw position stream in the absence of suitable maps is a first step to life-long opera-

tion in the area (for robotic and non-robotic agents alike). In known environments, such as cities, the network topology is constantly changing and the ability to automatically detect network changes (e.g. permanent or temporary road closures, new roads) would help in city planning and traffic control.

Prediction and pattern recognition By running machine learning algorithms on the semantic data (segment popularity, velocity), engineers and city planners may learn for example cluster speeds of drivers and aggregate uses of land for different parts of the day or year.

1.4 Relation to SLAM

Our trajectory clustering algorithm may be seen as a Dynamic Bayesian Network (DBN) for SLAM applications (see Figure 3-1). Unlike most existing SLAM methods, our algorithm clearly defines an objective function, provably converges to a local minimum, and provides provable and configurable error bounds. Additionally, many existing SLAM methods take as input rich sensor data and produce dense representations using heuristic methods that are computationally expensive (polynomial time). Our algorithm instead takes as input simple (x,y) points and produces a small representation (sparse topological map) in linear time. Our system has the additional capability to tweak the error parameters to produce a larger (more precise) or smaller (less precise) topological map. Our system may additionally be layered on top of other SLAM algorithms for compression purposes. For example, in our indoors experiment (where GPS is not available), we use instead a position stream localized using a SLAM algorithm. Traditionally, the map produced by SLAM is used for planning and navigation of robots; the uses for our map is broader: real-time decision making and trajectory compression (offline storage of the input stream). For more information on the relation of our work to SLAM, see Section 3.4.

1.5 Extension to higher dimensions

In this work, our discussion and experiments focus on $\mathbb{R}^2$ space for GPS and general (x,y) position streams. However, our methods are generalizable to signals of any integer $d > 0$ dimensional signal, including high-dimensional data, such as camera and LIDAR signals. Applications such as compression, denoising, activity recognition, road matching, and map generation become especially challenging for existing methods on higher dimensional data. Our algorithms are suitable for handling such large streaming data and can be run in parallel on networks or clouds.

1.6 Organization

Chapter 2 introduces the semantic representation and compression system, and Chapter 3 discusses the related work for each part of the system. Chapter 4 defines the problems formally. The system is comprised of preprocessing steps (Chapter 5) and the 2 major subcomponents: the trajectory mapping (Chapter 6) and the trajectory compression (Chapter 7). The results for our system are discussed in Chapter 8. We conclude in Chapter 9.

Chapter 2

Semantic representation and compression system overview

The main problem we address in this thesis is efficient signal compression by exploiting the semantic structure within the signal. Our system thus both extracts the semantic representation and computes a compressed signal as output. We call the problem of extracting the semantic representation trajectory mapping because we essentially produce a map of the agent's world from its observations. We call the problem of compressing the signal using the semantic representation trajectory compression. Additionally, we perform some preprocessing steps to the original input for computational speedup or to clean the data. The power of our system is that it is based on computational geometry and shape fitting algorithms, rather than string compression algorithms that seem to be less natural for sensor data from moving agents.

In this chapter, we present a system (Figure 2-1) for learning and compressing structure of the mobile agent's motion patterns (as a sequence of states) from its sensor signals. The system is composed of three separate modules: (i) Preprocessing, (ii) Trajectory Mapping, and (iii) Trajectory Compression. The result is a small sequence of points and a weighted directed graph (the map). We first give a non-formal description of these modules, with more details in Section 6.

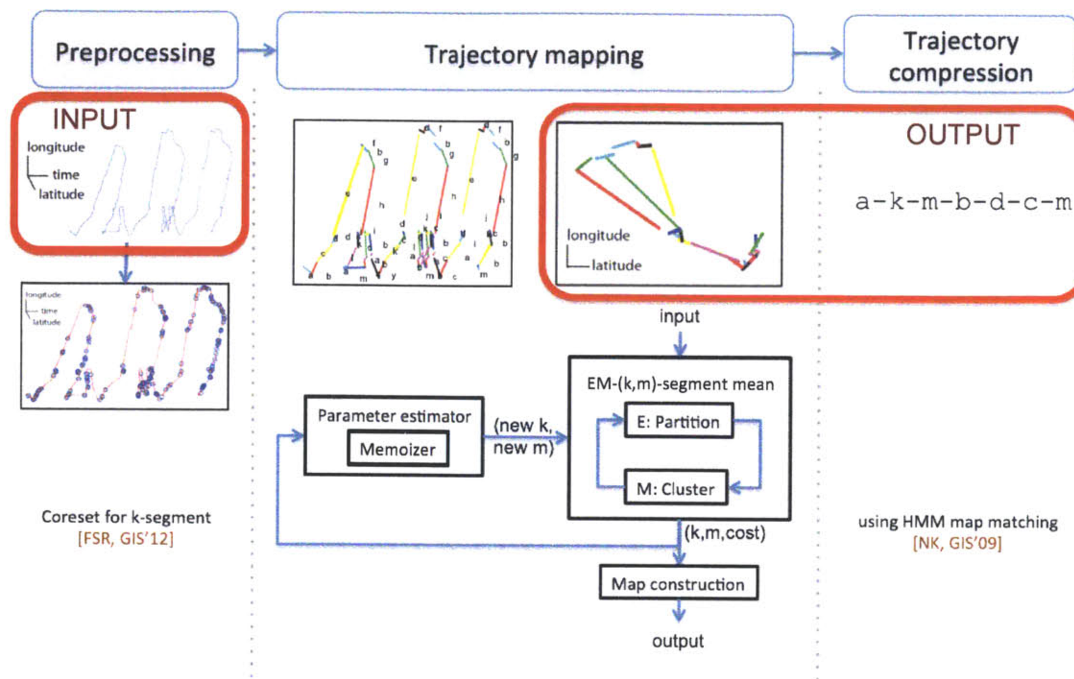


Figure 2-1: Semantic representation and compression system. The output of the trajectory clustering & map stage is useful for real-time decision making, whereas the output of the trajectory compression stage is useful for data logging.

2.1 Preprocessing

Coreset construction [16]: In order to apply the above algorithms on scalable data, we do not run them directly on the original input points. Instead, we apply them on a *coreset* of the original points. A coreset for HMM is a small set of points which is also a semantic compression of the original points in a way that running modules (ii) and (iii) as mentioned above on the coreset of the input stream of points, instead of the original points would produce approximately the same result.

Data patcher: In cases where there are large gaps in the input stream (as is common with GPS streams) *and* we have access to an existing topological map of the environment, we may patch the gaps of the input data by applying Hidden Markov map matching [25] to the input stream. The map matching algorithm gives a possible and plausible sequence of segments (i.e. streets) for each gap, which can be interpolated and inserted into the original sequence. See Figure 5-1 for an example.

2.2 Trajectory mapping

This module computes a map (graph) and a trajectory that consists of nodes on the map that maximizes the likelihood estimation of the input stream to some error parameters.

First, we perform **trajectory clustering** on the input, solving what we call the (k,m)-segment mean problem (see Algorithm 1 and see Section 4.1 for definitions). This algorithm is a partial map maker that gets a sequence of n GPS points $(lat, lon, time)$ from a mobile agent that travels over a path of size $k \ll n$ on an (unknown) map that consists of $m \ll k$ disconnected linear (geometric) segments on the plane. The algorithm outputs a set of m segments labeled from 1 to m, that approximates, together with a sequence of k integers between 1 to m which are the approximated path.

We introduce the **coreset for 2-segment mean**, which is the key component that allows us to efficiently compute in linear time a local optimal k-partitioning of the input, with the constraint that their projection onto Euclidean space consists of only m segments. See Section 6.2 for more details.

Here, we also perform **paramester estimation** on k and m by binary searching the parameter spaces in a nested manner. For a given m, there is an optimal k. See Algorithms 3 and 4 for more details.

Finally, the **map construction** step turns the m disconnected segments of the (k,m)-segment mean into a connected map, and the path into a valid path. It first connects every two segments that are consecutive on the path of the agent to turn the map into a set of connected segments. Each state corresponds to 2 nodes and an edge on the map. Each unique state transition adds an extra edge to the map. This trajectory map is the semantic representation of the agent's world.

2.3 Trajectory compression

We may further apply several lossless compression techniques for logging purposes. Our system applies a binary recursive **trajectory compression** of the compressed path against the map. For example, if the only (or shortest) path from segment a in the map to segment

f is through segments b, c, d, e, then we can compress the sub-sequence (a, b, c, d, e, f) into (a, f). If there are other paths from a to f, then we subdivide the problem and try again (See Algorithm 6). We call the recursively compressed output the compressed trajectory.

We may further apply state-of-the-art **string compression** algorithms such as bzip2 on the recursively compressed path (which we call the semantic+bzip2 output).

2.4 Additional features

- Supports streaming computation (through application of the k-segment coreset [16]).
- Supports parallel computation of trajectory mapping. The trajectory clustering algorithm may be run on up to $\frac{k}{2}$ machines (see Section 6.3.3), and the parameter estimation algorithm can be run on $\log^2(n)$ machines (one machine for each set of (k, m) parameters), and their results may be combined in a MapReduce manner.
- Supports extension to multi-agent systems. The map of multiple users may be combined to form a single map representing all users. For example, the map from the first agent merged with the map of the second agent will yield a more complete map of the multi-agent system's environment. Our system may then compress the data from all the users as a single trajectory or as individual trajectories.

Chapter 3

Related Work

3.1 Coreset for k-segment mean

A coreset is a small semantic compression (data structure) of an original signal P, such that every k-segment or (k, m)-segment has the same cost with respect to the original signal as to the coreset, up to $(1 + \varepsilon)$-multiplicative error, for a given $\varepsilon > 0$ and arbitrary constants k and m (refer to Section 4.1 for definitions). Coresets provide shorter running time and allow for streaming and parallel computing. This coreset is a smart compression of the input signal that can be constructed in $O(n \log n)$ time [16].

We use this result to reduce our input from size $O(n)$ to $O(\frac{k}{\varepsilon^2})$ to more efficiently solve the trajectory clustering problem below, where ε is our error parameter. Our trajectory clustering solution additionally supports streaming, as a property of this coreset.

In our work, we additionally formalize and implement the 1-segment coreset, which solves the 2-segment mean problem optimally and efficiently. We use this result to obtain a parallelizable linear-time trajectory clustering algorithm. We demonstrate our results for extracting trajectory clusters from GPS traces. However, the results are more general and applicable to other types of sensors.

3.2 Trajectory clustering

3.2.1 (k, m)-segment mean

Signal or trajectory clustering is significantly harder than segment clustering (lime simplification). In signal clustering, we wish to cluster sets of traces-points from arbitrary time intervals, which is more computationally challenging than clustering consecutive points in time as in line simplification.

Our previous work uses the coreset from [16] and shows how to apply it for computing a local minima for the (k, m)-segment mean problem (see Section 4.1 for definition) and compressing signals. In [16], an algorithm for computing maps (albeit disconnected) was also suggested; however, it uses dynamic programming and takes cubic time in n, as compared to our improved linear time algorithm. The previous work presented small preliminary experimental results, but larger experiments were practically intractable. This thesis improves the efficiency of the algorithm and implementation allowing for more scalable experiments.

3.2.2 Map inference

Due to the availability of GPS sensors in a variety of everyday devices, GPS trace-data is becoming increasingly abundant. One potential use of this wealth of data is to infer, and update, the geometry and connectivity of road maps, using what are known as map generation or map inference algorithms. These algorithms offer a tremendous advantage when no existing road map data is present rather than incur the expense of a complete road survey, GPS trace-data can be used to generate entirely new sections of the road map at a fraction of the cost. Road map inference can also be valuable in cases where a road map is already available. Here, they may not only help to increase the accuracy of drawn road maps, but also help to detect new road construction and dynamically adapt to road closures–useful features for digital road maps being used for in-car navigation. A recent survey of the existing literature [8] identified *trace merging algorithms* as a category of map inference algorithms.

Trace merging algorithms generally accumulate traces into a growing road network, where each addition meets the location and bearing constraints of existing roads so far. A post-processing phase removes relatively unsupported road findings. Representative algorithms in this class include [12] and [26], and similar methods include [37] and [13].

Our approach is similar to the trace merging algorithms in the literature, however current methods are heuristic and do not have defined optimization functions. Additionally, these methods do not use the results as a method of compression for the original trajectory. Experiments using controlled GPS traces in known environments (such as road maps) are common. By contrast, we address the more general problem of generating topological maps for arbitrary settings and scopes, which may or may not have available ground truth information.

3.2.3 Other techniques

Our approach to signal compression builds on line simplification. Line simplification is a common approach to compressing and denoising continuous signals and has already been well studied for purposes such as cartography [17] and digital graphics [6].

There exists a body of signal clustering work [14, 22, 36] whose goal is to translate a time signal into a map of commonly traversed paths. In particular, Sacharidis et al. [15] process GPS streams online to generate common motion paths and identify "hot" paths. However, the majority of this previous work either present an algorithm without defining any optimization problem, or present quality functions based on a ad-hoc tuning variables whose meaning are not clear. Furthermore, these works do not preserve the temporal information in the original data. No current work in trajectory clustering uses the results as a method of compression for the original input trajectory.

In general, the classes of existing solutions may be characterized by at least one of the following:

- No optimization problem is defined, i.e. no criteria or objective function.
- Optimization problem is defined, but there are no local minimum guarantees.
- No streaming/parallel support and practically cannot run on large datasets.

We propose a solution with properties of all three: optimization problem defined with local mimimum guarantees, streaming/parallel support, and practical for large datasets.

3.3 Trajectory compression

There is an increasing number of rapidly growing repositories capturing the movement of people in space-time. Movement trajectory compression becomes an obvious necessity for coping with such growing data volumes. [24] examines algorithms for compressing GPS trajectories, including Douglas-Peucker Algorithm, Bellman's Algorithm, STTrace Algorithm and Opening Window Algorithms, and compares them empirically.

[20], [29] address trajectory compression under network constraints by applying map matching algorithms. In our work, we similarly explore trajectory compression under network constraints; however, our system does not require that it be given the network structure (map), since it is capable of extracting out the network structure (map) from raw data.

3.3.1 Text-based compression

Current common practice for compressing sensor streams is simply applying popular text-based compression algorithms on the data stream, like bzip2. Bzip2 is the standard used by Robot Operating System (ROS) libraries [3]. Lossless compression reduces bits by identifying and eliminating statistical redundancy and is possible because most real-world data has statistical redundancy. Lossy compression reduces bits by identifying marginally important information and removing it.

Bzip2, which we use for comparison, is a free and open source implementation of the Burrows-Wheeler algorithm [11] and is a lossless data compression algorithm. Bzip2 incorporates many well-studied compression ideas [18, 7, 19] and is the only supported compression format for ROS .bag files, commonly used for storing location data in robotic applications [3].

As used in our system, compression algorithms can be layered on top of one another, as long as the redundancy eliminated by the different algorithms is different. We layer bzip2 on top of our algorithm to achieve further non-lossy compression.

3.3.2 Hidden Markov map matching

The problem of matching measured latitude/longitude points to roads is very important for mobile agents. Map matching has been extensively studied, implemented, tested, and adapted for a wide number of applications [25, 35, 10, 2, 27]. Hidden Markov map matching uses a Hidden Markov Model (HMM) to find the most likely route through a given map or network. The route is represented by a time-stamped contiguous sequence of edges on the map. The HMM elegantly accounts for measurement noise and the topology of the network. Hidden Markov map matching also employs A*, a heuristic shortest path algorithm, to reproduce parts of the route that are only implied by a sparse data input.

The ability to reproduce information from a map using this algorithm lends itself well to compression and, as a side effect, may reduce the noise in the sensor signal [9, 21]. Hidden Markov map matching is a key component to our trajectory compression algorithm. We additionally use HMM map matching to patch gaps within GPS signals in our preprocessing step, where a prior map is available.

While map matching algorithms assume that the map is given, and uses HMM only for detecting the path, our trajectory mapping module is the first that estimates the map from the input data, which can then be used to compress the trajectory.

3.4 SLAM

Simultaneous localization and mapping (SLAM) is a technique usually used in robotics to build up a map within an unknown environment (without a priori knowledge), or to update a map within a known environment (with a priori knowledge from a given map), while at the same time keeping track of their current location. Mapping is the problem of integrating the information gathered with the robot's sensors into a given representation. Central aspects in mapping are the representation of the environment and the interpretation of sensor data. Localization is the problem of estimating the pose of the robot relative to a map. All that is given are measurements $z_{1:t}$ and controls $u_{1:t}$, and the output is a map M and the posterior over the poses $x_{1:t}$ [30, 31].

Our trajectory clustering algorithm may be seen as a Dynamic Bayesian Network (DBN)

for SLAM applications (see Figure 3-1). The input measurements are GPS points taken over time and no control inputs are required. Our map M is a connected directed graph whose nodes are features defined by long corridors of space (segments), rather than visual or pose features. Respectively, our output poses X are a sequence of these segments. The output poses are conditioned on the agent position (GPS point readings), as well as on the map, which constrains the free space of the agent. The map is also conditioned on the agent position (GPS point readings). The output poses affect the map by adjusting the segment locations, and the map affects the output poses by providing the limited set of segment choices, and thus we perform simultaneously localization and mapping to yield our k output poses and our map of size $O(m^2)$.

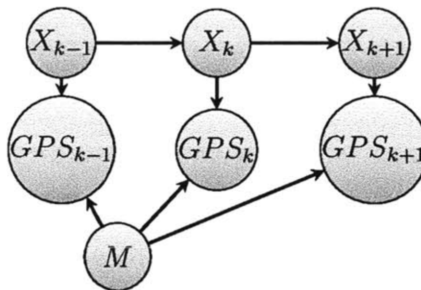


Figure 3-1: Dynamic Bayesian network. In our system, our input measurements are GPS points taken over time. Our map M is a connected directed graph whose nodes are features defined by long corridors of space (segments). Our output poses X are a sequence of these segments.

Whereas SLAM methods typically do not have defined objective functions or local minimum guarantees, our objective function is defined as the (k, m)-segment mean problem (see Section 4.1) and our algorithm will provably reach a local minimum (see Chapter 6).

Our work is a type of topological SLAM [32], which are SLAM methods whose underlying map representation is a topological map. Topological maps are concise and often sparse maps that represent an environment as a graph, whose vertices are interesting places and whose edges represent the paths between them. The advantages of topological maps are their computational efficiency, their reduced memory requirements, and their lack of dependence on metric positioning.

Whereas existing topological SLAM methods use local sensor information (e.g. camera) to simultaneously map and localize heuristically, our work uses globally positioned sensor information (e.g. GPS) to produce topological maps with provable and configurable

error bounds, while localizing the points to the map. Our map is then used for real-time decision algorithms or for compressing the original input (trajectory compression).

Note that since our SLAM algorithm takes only globally positioned data as input (e.g. GPS), we may layer our system on top of another SLAM implementation that takes in local sensor input and produces globally positioned data.

Chapter 4

Problem Statement

4.1 Definitions

Definition 4.1.1 (Signal) *A signal $P = \{p_{t_1}, \cdots, p_{t_n}\}$ in $\mathbb{R}^d$ is a set of n points in $\mathbb{R}^d$, where each point $p_t \in P$ is assigned an integer timestamp t.*

Definition 4.1.2 (Segment) *A segment $X = (x_1, x_2)$ in $\mathbb{R}^d$ is two points in $\mathbb{R}^d$.*

Let $\tau(P, X) \geq 0$ be the *fitting cost* for a segment X and a signal P. Suppose that X fits to P in time by associating the endpoints of the X with the endpoints of P (p_{t_1} and p_{t_n}). We call the sum of the time-wise Euclidean distances between X and P the fitting cost. Since the P may contain more points than X (two, by definition), we first interpolate within the endpoints of X for the additional points in P in order to compute the fitting cost. Note that our fitting cost is different from Euclidean distance (though we make use of it), which does not consider time by itself. More generally:

Definition 4.1.3 (Fitting cost) *For an integer $k \geq 1$, a* k-segment in $\mathbb{R}^d$ *is a piecewise linear function $f : \mathbb{R} \to \mathbb{R}^d$ of k linear segments. The* fitting cost *of f for a given signal P in $\mathbb{R}^d$ is the sum of squared distances to its k segments,*

$$\tau(P, f) := \sum_{p_t \in P} \|f(t) - p_t\|^2 .$$

Here, $\|x - y\|$ is the Euclidean distance between the x and y in $\mathbb{R}^d$.

Definition 4.1.4 (**k-segment mean**) *For an integer $k \geq 1$, a* k-segment in $\mathbb{R}^d$ *is a piecewise linear function $f : \mathbb{R} \to \mathbb{R}^d$ of k linear segments. The* k-segment mean f^* *of P minimizes the fitting cost $\tau(P, f)$ over every possible k-segment f.*

To deal with compressed signals, we need a more robust definition of discrete sequences, where we might assign higher weights for more important points, and even negative weights for points that should be far from the segments.

Definition 4.1.5 (Weighted signal) *A* weighted signal (C, w) *is a pair of a signal C, and a weight function $w : C \to \mathbb{R}$. The* weighted error *from (C, w) to $f : \mathbb{R}^d \to \mathbb{R}$ is*

$$\tau_w(C, f) := \sum_{c_t \in C} w_c \cdot \|f(t) - c_t\|^2 .$$

Given a signal P, a coreset C for P approximates its fitting cost to any k-segment up to $(1 \pm \varepsilon)$-multiplicative factor, as formally defined below.

Definition 4.1.6 (**(k, ε)-coreset**) *Let $k \geq 1$ be an integer and $\varepsilon > 0$. A weighted subsequence (C, w) is a* (k, ε)-coreset *for a sequence P if for every k-segment f in $\mathbb{R}^d$ we have*

$$(1 - \varepsilon)\tau(P, f) \leq \tau_w(C, f) \leq (1 + \varepsilon)\tau(P, f).$$

Definition 4.1.7 (**(k, m)-segment mean**) *For a pair of integers $k \geq m \geq 1$, a* (k, m)-segment *is a k-segment f in $\mathbb{R}^d$ whose projection $\{f(t) \mid t \in \mathbb{R}\}$ is a set of only m segments in $\mathbb{R}^d$. Given a sequence P in $\mathbb{R}^d$, the* (k, m)-segment mean f^* *of P minimizes $\tau(P, f)$ among every possible (k, m)-segment f in $\mathbb{R}^d$.*

Definition 4.1.8 (**(k, m)-segment map**) *A* (k, m)-segment map *is a directed graph G that encodes the m segments as well as the $k - 1$ historically valid transitions between the segments. Let M_s be the set of start points and M_e is the set of end points in $\mathbb{R}^d$ for the m segments. The vertices are $V := \{v : v \in M_s \cup v \in M_t\}$. Let $\{K_{si}\}$ be the sequence of start points and $\{K_{ei}\}$ is the set of end points for the k segments and $i = [t_1 ... t_n]$. The edges are $E := \{e : e \in m \cup e \in (K_{ei}, K_{s(i+1)})\}$.*

Definition 4.1.9 (Trajectory compression) *Let $Q = (q_1, q_2, \cdots, q_n)$ be a path on an Euclidean graph G. Let $\text{path}(p, q)$ denote the shortest path between two nodes p, q in G, where ties are broken according to lexicographical order on the coordinates of the vertices. A* compression *of Q is a sequence $T = (p_1, p_2, \cdots, p_m)$ of nodes in G such that concatenating the shortest path between consecutive pairs in T yields Q, i.e.*

$$(q_1, q_2, \cdots, q_n) = \big((\text{path}(p_1, p_2), \text{path}(p_2, p_3), \cdots, \text{path}(p_{m-1}, p_m)\big).$$

That is, a compression is efficient if $m \ll n$.

4.2 Problem definitions

We now define more formally the problems that we need to solve for each module of the system.

The trajectory mapping module approximates the trajectories of the agent by linear segments and attaches for every observation a segment. In general, the segments are not connected and the list of states does not describe continuous trajectory. This module additionally aims to solve this problem by turning the segments into a map of connected segments, and finding a continuous (connected) path on this map that maximizes the likelihood of generating the observations. Here, the likelihood is defined using distances between segments as defined in [25]; see reference for more details and motivation.

Problem 4.2.1 (Trajectory mapping) *Given a signal P in $\mathbb{R}^d$ (observations), and a pair of integers $k, m \geq 1$, compute a graph (state machine, Hidden Markov Model) and a path (Markov chain) that maximizes the likelihood of generating the observations, among every path of length k that consists of m states ((k, m)-segment mean).*

The next challenge is to compress the string that represents the path. Note that, since this string represents a path on a map, we can do more than usual string-compression algorithms; see example in Figure 2-1.

Problem 4.2.2 (Trajectory compression) *Given a map (graph) and a path on the map, compute a representation that uses the minimum amount of memory.*

Problem 4.2.3 (Semantic representation and compression system) *Our system integrates trajectory clustering and trajectory compression. It takes as input a signal P and produces a map M and a sequence of segments k', such that $k' < k$, such that the intermediate (k, m)-segment mean may be recovered losslessly.*

Chapter 5

Preprocessing

This chapter details the algorithms in the Preprocessing block of the system (see Figure 2-1). The algorithms in the Preprocessing block are optional but useful for some types of inputs. For computation speedup (and applications which may afford a larger error parameters), we reduce the input data from n points to $O(\frac{k}{\varepsilon^2})$ points using the coreset of k-segments, where ε is the error parameter. For choppy signals as is common with GPS (and where an existing map is available), we may use the map to patch the input signal with plausible intermediate trajectories.

5.1 Coreset for k-segments

In order to deal with large dataset of points, we use the (k, ε)-coreset from [16] (see Section 4.1), and observe that it approximates the log-likelihood of every (k, m)-segment mean of at most k states.

Theorem 5.1.1 ([16]) *Let P be a set of n observations, and let $\varepsilon > 0$, $k \geq 1$ be constants. Let (C, w) be the output of a call to* CORESET(P, k, ε). *Then (C, w) is a (k, ε)-coreset for P, with high probability, and its construction time is* $O(n \log n)$.

Proof: See [16] for the full algorithm pseudocode and proof. □

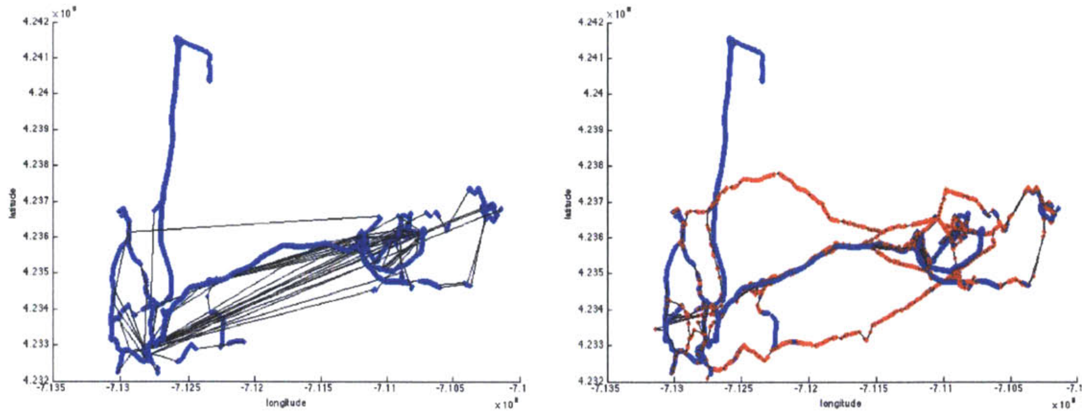


Figure 5-1: *Left*: Shown here is a raw GPS trajectory (blue) projected onto x-y. The GPS drops are indicated by the black connections between points.*Right*: Using the map of Massachuetts, we performed map matching to guess what roads the agent took between GPS signal gaps, and these data patches are shown in red. As shown, data patching, eliminates most of the black connections.

5.2 Data patcher

Since some of our experiments use GPS data, which is prone to signal drops and thereby gaps in the input stream, we introduce a preprocessing step to patch gaps in the GPS data stream by using map matching. We used a HMM-based map matching implementation, detailed in [25], which takes as input a sequence of GPS points and a map.

Algorithm: The output is the most likely sequence of edges (i.e. streets) traversed on the map. When a map is available and there are large gaps in the input stream, we apply this map matching algorithm on our input stream to generate the most likely sequence of edges missed during the signal outage for each gap, which we convert into sequences of points to patch our original sequence. Figure 5-1 demonstrates an example of the effectiveness of data patching in cleaning up signal discontinuities.

Analysis: The complexity of the map matching algorithm is $O(n \cdot \|V\|^2 \cdot \|E\|)$, since its subcomponents Viterbi and A* require $O(n \cdot \|V\|^2)$ and $O(\|E\|)$, respectively. n is the length of the input, $\|V\|$ is the number of nodes in our map, and $\|E\|$ is the number of edges in our map. See Chapter 7 for more details on the map matching algorithm.

Chapter 6

Trajectory mapping

In this Chapter, we suggest algorithms for the problems defined in Chapter 4 relevant to the (k,m)-segment mean and (k, m)-segment map, which forms the first half of our semantic compression system after the initial preprocessing (see the Trajectory clustering and map block in Figure 2-1). First, we give an overview of the (k,m)-segment mean problem and our Expectation-Maximization (EM) algorithm. Then, we formulate the coreset for 1-segment mean (linear regression), which enables a linear-time implementation of the (k,m)-segment mean algorithm. We explain the parameter estimation algorithm, and finally, we discuss the map construction step.

6.1 (k, m)-segment mean algorithm

For constructing the (k,m)-segment mean (see Definition 4.1.7), we suggest a novel algorithm that converges to a local minimum, since it has the properties of an EM (Expectation Maximization) algorithm. Each iteration of our EM algorithm takes $O(n)$ time, an improvement over the naive dynamic programming approach in the literature that takes $O(n^3)$ time [16]. Additionally, our algorithm is parallelizable.

Expectation-Maximization: In the partitioning (expectation) step, we compute the best k-segment mean, given that is projection is a particular m clusters. In the clustering (maximization) step, we compute the best m clusters, given the k-segment and the assignments to their respective clusters.

The bottleneck of the computation occurs in the partitioning (expectation) step, where in order to find the globally optimal k-segment mean with a particular set of m clusters, the exhaustive dynamic programming solution requires $O(n^3)$ [16]. This algorithm quickly becomes impractical for even moderately sized datasets.

Our (k, m)-segment mean algorithm still takes a dynamic programming approach: we split the computation into $\lceil \frac{k}{2} \rceil$ 2-segment mean problems (see Figure 6-1), which we show we can compute in linear time using what we call the 1-segment coreset (refer to Section 6.2 for more details). We additionally show that we can compute all $\lceil \frac{k}{2} \rceil$ subproblems simultaneously using $O(1)$ operations, since all of the subproblems are independent, and in $O(n)$ time. This simultaneous computation lends itself well to parallelism. Since the $\lceil \frac{k}{2} \rceil$ subproblems only update half of the partitions (each 2-segment mean updates one partition, see Figure 6-1), we may similarly split the computation into a different set of $\lceil \frac{k}{2} \rceil$ 2-segment mean problems, which we distinquish as shifting the odd and even partitions. See Algorithm 1 for the pseudocode.

Algorithm 1: km_seg_EM($k, m, input$)

```
   /* Initial partition of input into k partitions                */
1  partition ← DOUGLAS_PEUCKER(input, k)
   /* Initial clustering of k partitions to m types               */
2  cluster ← LEAST_SQUARES(partition)
   /* Initial error compared to input                             */
3  cost ← GET_COST(partition, clusters, input)
4  lim ← Inf
   /* Expectation-Maximization                                    */
5  while cost < lim do
   |  /* Expectation step                                         */
6  |  partition ← SHIFT_EVEN_PARTS(partition, clusters)
   |  /* Expectation step                                         */
7  |  partition ← SHIFT_ODD_PARTS(partition, clusters)
8  |  clusters ← LEAST_SQUARES(partition)    /* Maximization step */
9  |  lim ← cost                                  /* Update limit */
10 |  cost ← GET_COST(partition, clusters, points)  /* Update cost */
11 return (partition, clusters, cost)
```

Theorem 6.1.1 *Let P be a signal in $\mathbb{R}^d$, and $i_{\text{end}} \geq 1$, $k \geq m \geq 1$ be integers. For every i, $1 \leq i \leq i_{\text{end}}$, let S_i denote the m segments that are computed during the ith iteration within*

the EM loop of $\text{KM_SEG_EM}(P, k, m)$, *which solves the* (k, m)*-segment mean problem. Then*

a) *For every* i, $1 \leq i \leq i_{\text{end}}$, *we have*

$$\sum_{p \in P} \text{Dist}(p, S_i) \leq \sum_{p \in P} \text{Dist}(p, S_{i-1}).$$

b) *The running time of this algorithm is* $O(nmi_{\text{end}})$ *time.*

Proof:

a) See Algorithm 1 for the EM-(k, m)-segment mean algorithm. The estimation (partition) step performs a 2-step process to adjust the partitions to reduce the cost when possible. The algorithm first shifts the $\lceil \frac{k}{2} \rceil$ even partitions, such that all shifts may occur independently; then, the algorithm does the same to the $\lceil \frac{k}{2} \rceil$ odd partitions. The maximization (cluster) step computes the spline efficiently using least squares, yielding the best m segments.

   The KM_SEG_EM algorithm converges to a local minimum in the style of an EM algorithm. At each iteration, we perform an estimation step and a maximization step. In the estimation (partitioning) step, we shift the partitions if and only if it reduces the overall cost (fitting cost). In the maximization (clustering) step, we compute new segements if and only if they reduce the overall cost. We repeat the estimation and maximization steps until convergence. The algorithm terminates when neither step produces a lower cost. Since the cost is monotonically decreasing in each iteration, we guarantee convergence to a local optimum.

b) Each iteration of the EM algorithm requires $O(nm)$ for the partitioning (expectation) (see Section 6.3), $O(nm)$ for the clustering (maximization) (see [16]), and $O(n)$ for the cost computation. Thus, the overall cost is $O(nmi_{end})$, where i_{end} is the number of iterations until termination.

□

6.2 Optimal algorithm for 2-segment mean

In this section, we present an efficient optimal algorithm for the 2-segment mean, which is a specific case of the k-segment mean problem (See Chapter 4.1 for definition). Our algorithm is linear-time and constant-space using a constant size 1-coreset (with $\varepsilon = 0$, see Chapter 4.1 for definitions). We first give an informal overview for our approach. Then, we formalize the 1-coreset, present the optimal algorithm for 2-segment mean, and prove its complexity bounds. In the next section, we present the improved partitioning algorithm (estimation step) of the trajectory clustering algorithm, using the 2-segment mean algorithm.

Note that in this Section we are careful to include d, the dimensionality of the input, in our complexity analysis. For the rest of the thesis, we assume it to be a constant (2, in this work).

6.2.1 Informal overview

First, we take advantage of the fact that we must exhaustively perform computations against all subintervals within a time interval of size n. This added structure allows us to incrementally compute costs much more efficiently than performing each computation independently, reducing the total time complexity of the n operations from $\Theta(n^2)$ to $\Theta(n)$, and the amortized cost of each operation is $O(1)$. Selecting the subinterval with the lowest cost is the same as computing the 2-segment mean of a signal (See Section 4.1 for definition).

Second, we wish to simultaneously compute the fitting cost for a particular subinterval against several segments. Here, we restructure the computation to take advantage of MATLAB's extensively fine-tuned libraries for matrix operations. The resulting computation requires a factor m fewer operations, though each operation takes $O(m)$ longer, with overall runtime $O(d^2mn)$. The real result is in significantly reducing the constant factor of performing many fitting cost computation.

6.2.2 Coreset for 1-segment mean

Definition 6.2.1 (Fitting matrix) *A fitting matrix D for a segment X and a signal P is a d-by-d matrix such that $\tau = \text{trace}(D)$ is the fitting cost between X and P. The fitting matrix may be expressed as*

$$D = (AX - P)^T(AX - P) \tag{6.1}$$

Remark 6.2.2 *D, defined above, is similar to $\|AX - P\|^2$. However, only the diagonal entries affect the fitting cost.*

This fitting cost computation is expensive in our (k,m)-segment mean algorithm because we must compute a new AX for every new time interval, which costs $O(td)$, where t is the length of the time interval and d is the dimensionality of the input stream. The subsequent subtraction and multiplication also requires $O(td)$ time.

Claim 6.2.3 *Suppose $\tau = \text{trace}(D)$ is the fitting cost between a segment X and a signal P for time range $[t_1 \ldots t_2]$ (see Section 4.1 for definitions). We may construct A such that AX is an interpolation of X with $t = (t_2 - t_1 + 1)$ parts between the two endpoints of the segment.*

For instance, for a 2-dimensional (x, y) constant-frequency signal, we have:

$$A = \begin{bmatrix} 1 & t_1 \\ 1 & t_1 + 1 \\ \vdots & \vdots \\ 1 & t_2 \end{bmatrix}, X = \begin{bmatrix} b_x & b_y \\ m_x & m_y \end{bmatrix}, \text{and } P = \begin{bmatrix} x_{t_1} & y_{t_1} \\ x_{t_1+1} & y_{t_1+1} \\ \vdots & \vdots \\ x_{t_2} & y_{t_2} \end{bmatrix}$$

where b_y represents the intercept (at t_1) of the y-dimension of the segment x and m_y represents the change in y-coordinate per change in time (slope). The same applies to the x-dimension (b_x and m_x), and this is generalizable to any d dimensions.

Observation 6.2.4 *To solve the 2-segment mean problem, we must compute the fitting cost for every possible 2-segment of some signal P (in order to find the 2-segment with the*

smallest fitting cost). We may order these computations in such a way that each 2-segment differs from the previous by a point insertion to one segment and a point deletion from the other segment.

Observe that Equation 6.1, when expanded is:

$$D = X^T A^T A X - P^T A X - (P^T A X)^T + P^T P \tag{6.2}$$

Claim 6.2.5 *The matrices $A^T A$, $P^T A$, and $P^T P$ are size $O(d)$-by-$O(d)$ and require only $\Theta(1)$, $\Theta(d)$, and $\Theta(d^2)$ (respectively) time and space to update for point insertions and deletions. d is the dimensionality of the input stream. Their respective sizes are 2-by-2, d-by-2, and d-by-d.*

For example, suppose we wished to compute these matrices for time interval $[t_1...t_2+1]$ given the matrices for time interval $[t_1...t_2]$ (point insertion). Then, all we need are the new point (x_{t_2+1}, y_{t_2+1}) and the time $t_2 + 1$ to update the matrices in $O(d^2)$ time by simple matrix operations:

$$A^T A_{[t_1...t_2+1]} = A^T A_{[t_1...t_2]} + \begin{bmatrix} 1 & t_2+1 \\ t_2+1 & (t_2+1)^2 \end{bmatrix} \tag{6.3}$$

$$P^T A_{[t_1...t_2+1]} = P^T A_{[t_1...t_2]} + \begin{bmatrix} x_{t_2+1} & x_{t_2+1} \cdot (t_2+1) \\ y_{t_2+1} & y_{t_2+1} \cdot (t_2+1) \end{bmatrix} \tag{6.4}$$

$$P^T P_{[t_1...t_2+1]} = P^T P_{[t_1...t_2]} + \begin{bmatrix} x_{t_2+1}^2 & x_{t_2+1} \cdot y_{t_2+1} \\ x_{t_2+1} \cdot y_{t_2+1} & y_{t_2+1}^2 \end{bmatrix} \tag{6.5}$$

By similar operations, a point deletion also requires only $O(d^2)$ update time.

Claim 6.2.6 *The matrix X is small and requires $\Theta(d)$ time to compute. d is the dimensionality of the input stream. X is a 2-by-d matrix.*

For $d = 2$, the computation of X is as follows. Given a segment with start point (x_1, y_1)

and end point (x_2, y_2), and time interval $[t_1...t_2]$:

$$X = \begin{bmatrix} b_x & b_y \\ m_x & m_y \end{bmatrix} = \begin{bmatrix} x_1 & y_1 \\ \frac{x_2-x_1}{t_2-t_1} & \frac{y_2-y_1}{t_2-t_1} \end{bmatrix} \tag{6.6}$$

Claim 6.2.7 *The computation of the fitting matrix D may be computed in* $O(d^2)$ *time for point insertions and deletions, where d is the dimensionality of the input stream.*

By Claim 6.2.5 and 6.2.6, $A^T A$, $P^T A$, $P^T P$, and X all require just $O(d^2)$ time to update for point insertions and deletions. The matrix multiplications shown in Equation 6.2 also requires $\Theta(d^2)$ time. All matrices are very small, and the resulting matrix is d-by-d.

Definition 6.2.8 (Dynamic 1-segment coreset) (C, E, F) *is a coreset for input* (A, P) *if for all* x*:*

$$D = x^T A^T A x - P^T A x - (P^T A x)^T + P^T P = x^T C x - E x - (E x)^T + F \tag{6.7}$$

A coreset is dynamic if it supports point insertions and deletions.

Theorem 6.2.9 *A dynamic* 1*-segment coreset* (C, E, F) *for input* (A, P) *can be computed such that construction time is* $O(d^2 n)$*, updates (point inserts take* $O(d^2)$ *time, and all operations take* $O(d^2)$ *space. C,F, and E are respectively 2-by-2, d-by-2, and d-by-d.*

Proof: Respectively, $C = A^T A$, $E = P^T A$, and $F = P^T P$. By Claim 6.2.5, point insertions and deletions take $O(d^2)$ time and space. Naturally, construction of such a coreset for n points requires n point inserts or $O(d^2 n)$ time and $O(d^2)$ space. By Claim 6.2.7, (C, E, F) computes the fitting matrix D for any segment x. □

6.2.3 Algorithm for $(2, m)$-segment mean

Definition 6.2.10 ($(2, m)$-segment mean) *For an integer* $m \geq 1$*, a* $(2, m)$-segment *is a* 2*-segment* f *in* $\mathbb{R}^d$ *whose projection* $\{f(t) \mid t \in \mathbb{R}\}$ *is a set of at most* 2 *of* m *given segments in* $\mathbb{R}^d$*. Given a sequence* P *in* $\mathbb{R}^d$*, the* $(2, m)$-segment mean f^* *of* P *minimizes* $\tau(P, f)$ *among every possible* $(2, m)$*-segment* f *in* $\mathbb{R}^d$ *for some given* m *segments.*

Theorem 6.2.11 *Using the dynamic* 1*-segment coreset, we may compute the optimal* $(2, m)$*-segment mean in* $O(d^2mn)$ *time and* $O(d^2)$ *space.*

Proof: We start by constructing a dynamic 1-segment coreset for all the points and computing the optimal fitting cost among a set of M segments. Then, one by one, we remove a point from the coreset and add it to a second coreset (initially empty). With each pop and push, we compute again the optimal fitting cost. By Claim 6.2.7, each fitting cost computation takes $O(d^2m)$ time and space (for m segments). We take the lowest fitting cost among the n computations, for an overall computation cost of $O(d^2mn)$ time and $O(d^2)$ space. See Algorithm 2 for the pseudocode and Figure 6-1 for graphical demonstration.

Without the coreset, each (independent) fitting cost computation would take $O(d^2mn)$ time for an overall complexity of $O(d^2mn^2)$. □

Algorithm 2: 2_seg_mean(P, M)

```
1  n ← length(P)
2  min_cost ← ∞
3  coreset_L ← 1_SEG_CORESET(br)                              /* Empty */
4  coreset_R ← 1_SEG_CORESET(P)                          /* All points */
5  for i ← [1...n] do
6  |   (cost_L, m_L) ← FITTING_COST(coreset_L, M)
7  |   (cost_R, m_R) ← FITTING_COST(coreset_R, M)
8  |   if cost_L + cost_R < min_cost then
9  |   |   min_cost ← cost_L + cost_R
10 |   |   min_{m_L} ← m_L
11 |   |   min_{m_R} ← m_R
12 |   coreset_L ← 1_SEG_CORESET_INSERT(coreset_L, P[i])  /* Push point */
13 |   coreset_R ← 1_SEG_CORESET_DELETE(coreset_R, P[i])   /* Pop point */
14 return (min_cost, (min_{m_L}, min_{m_R}))
```

Remark 6.2.12 *Using the dynamic 1-segment coreset and the method of quadratic minimization [23], we may similarly solve the general* 2*-segment mean problem, which gives the optimal* 2*-segment solution, instead of selecting only among* m *options. This can also be computed in* $O(d^2n)$ *time and space for point insertions and deletions. However, for the case of inspecting a subproblem of the* (k, m)*-segment mean problem, we look specifically at the case of the* 2*-segment mean problem where we select from* m *given segments.*

6.2.4 Operation reduction

For a practical implementation we wish to simultaneously perform the fitting cost computation for a signal P against several given segments. Reducing the number of matrix operations greatly decreases the constant factor of our running time, although it does not affect the asymptotic running time. In this Section, we take advantage of MATLAB's extensively fine-tuned libraries for matrix operations.

Observation 6.2.13 *Each computation of fitting matrix $D = (AX - P)^T(AX - P)$ may be simplified and restructured as a small dot product, which can then be incorporated into a larger matrix multiplication. When computing fitting costs for a particular subinterval against m different segments, this allows MATLAB to perform a single (larger) operation instead of m (smaller) operations, which is much more efficient.*

Claim 6.2.14 *The computation of fitting matrix $D = (AX - P)^T(AX - P)$ may be simplified as a small dot product of 9 values for $d = 2$.*

We already know that A^TA, P^TA, and P^TP are small. For $d = 2$:

$$A^TA = \begin{bmatrix} a_{11} & a_{21} \\ a_{21} & a_{22} \end{bmatrix} \tag{6.8}$$

$$P^TA = \begin{bmatrix} pa_{11} & pa_{12} \\ pa_{21} & pa_{22} \end{bmatrix} \tag{6.9}$$

$$P^TP = \begin{bmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{bmatrix} \tag{6.10}$$

In Equation 6.11 and 6.12, we see that by simple matrix operations, we can re-write X^TA^TAX and P^TAX (and we ignore the non-diagonal entries).

$$X^TA^TAX = \begin{bmatrix} b_x^2a_{11} + 2b_xm_xa_{21} + m_x^2a_{22} & - \\ - & b_y^2a_{11} + 2b_ym_ya_{21} + m_y^2a_{22} \end{bmatrix} \tag{6.11}$$

$$P^TAX = \begin{bmatrix} pa_{11}b_x + pa_{12}m_x & - \\ - & pa_{21}b_y pa_{22}m_y \end{bmatrix} \tag{6.12}$$

In Equation 6.13, we see that the resulting fitting cost $\tau = \text{trace}(D)$ computation can be written as a dot product. $\hat{p}$ is the vector with coefficients relevant for the signal P. $\hat{s}$ is the vector with coefficients relevant for the segment X. This computation requires only 1 matrix operation.

$$\begin{aligned}
\tau &= \text{trace}\big(X^TA^TAX - P^TAX - (P^TAX)^T + P^TP\big) \\
&= b_x^2 a_{11} + 2b_x m_x a_{21} + m_x^2 a_{22} + b_y^2 a_{11} + 2b_y m_y a_{21} + m_y^2 a_{22} \\
&\quad - 2(pa_{11}b_x + pa_{12}m_x) - 2(pa_{21}b_y + pa_{22}m_y) + p_{11} + p_{22} \\
&= \begin{bmatrix} a_{11} & a_{21} & a_{22} & pa_{11} & pa_{12} & pa_{21} & pa_{22} & p_{11} & p_{22} \end{bmatrix} \cdot \\
&\quad \begin{bmatrix} b_x^2 + b_y^2 & 2(b_x m_x + b_y m_y) & m_x^2 + m_y^2 & -2b_x & -2m_x & -2b_y & -2m_y & 1 & 1 \end{bmatrix} \\
&= \hat{p} \cdot \hat{s}
\end{aligned} \tag{6.13}$$

Claim 6.2.15 *Computing the fitting costs $\hat{\tau}$ against m different segments requires only* 1 *matrix operation.*

Following from Theorem 6.2.11 and the matrix formulation given by Claim 6.2.14, to compute the fitting cost of a signal P against m different segments:

$$\hat{\tau} = \hat{p} \cdot \begin{bmatrix} \hat{s}_1 & \hat{s}_2 & \dots & \hat{s}_m \end{bmatrix} \tag{6.14}$$

The runtime is still $\Theta(d^2m)$, but with a much smaller constant factor.

Theorem 6.2.16 *Using the dynamic* 1*-segment coreset, we may compute the optimal* $(2, m)$*-segment mean in* $O(n)$ *matrix operations.*

Proof: Algorithm 2 requires n fitting cost computations, each for m segments. By Claim 6.2.15, we may perform each of the n fitting cost computations in 1 matrix operation, so

n total matrix operations are needed. We additionally need $O(n)$ matrix operations for the pushes and pops and the initial coreset construction, for an overall $O(n)$ operations.

The runtime is still $\Theta(d^2mn)$, but with a much smaller constant factor. □

6.3 Partitioning (expectation) step algorithm

In this section, we describe different ways of implementing the partitioning stage of the (k, m)-segment mean algorithm, having broken the problem into $\lceil \frac{k}{2} \rceil$ 2-segment mean problems. The objective of the partitioning stage is to find the best partition of the input into k parts given m possible segments to use. This is the (k, m)-segment mean problem with the m segments held constant.

Remark 6.3.1 *Finding the optimal global partitioning requires $O(n^3)$ time, using an exhaustive dynamic-programming approach [16]. This becomes quickly impractical even for medium-sized datasets. We instead propose algorithms for a locally optimal solution.*

First, we describe the naive algorithm, which computes fitting costs independently for each 2-segment mean subproblem (does not use the 1-segment coreset). Second, we describe an improved version that makes use of Theorem 6.2.15 to reduce the number of required operations and also its asymptotic runtime. Finally, we present a parallelizable version of the partitioning algorithm, which further reduces the required operations. The algorithms require $O(n^2m)$, $O(nm)$, $O(nm)$ time, respectively, and they require $O(n^2m)$, $O(n)$, $O(m)$ operations, respectively. The first two candidate algorithms produce the exact same result. The third candidate algorithm splits the input into subproblems somewhat differently. Our system uses the parallel implementation, as shown in Figure 6-2.

6.3.1 Naive algorithm

The simplest implementation performs a linear walk through the $k - 1$ 2-segment mean problems. For each subproblem, it checks whether the fitting cost would be lower if the partition were instead anywhere else between its bounds, given that it can select from any of m segments. This algorithm computes the fitting cost for each of the $O(n)$ subintervals

against each of the m segments, and *shifts* the partition according to the lowest cost. Continuing in this manner, this algorithm takes $O(mn^2)$ time, since each cost computation is $O(n)$.

6.3.2 Using 2-segment mean algorithm

This algorithm also performs a linear walk through the $k-1$ 2-segment mean problems. Making use of Theorem 6.2.11, it computes costs for each 2-segment mean subproblem incrementally to reduce the overall runtime to $O(nm)$. Making use of Theorem 6.2.16, it uses matrix operations to reduce the constant factor as well as the number of operations from $O(nm)$ to $O(n)$. Please see Figure 6-1 for graphical steps of the algorithm.

6.3.3 Extension for parallelism

We observe that shifting partition lines locally at most affects its 2 neighbors. Due to this conditional independence, note that we may shift all odd numbered partition lines without affecting the costs of one another; the same applies to even numbered partition lines. Thus, instead of performing a linear walk through the $\lceil \frac{k}{2} \rceil$ 2-segment mean problems, we split the computation into $\lceil \frac{k}{2} \rceil$ non-overlapping 2-segment mean problems, such that the moving partitions are all odd or all even numbered. We perform all even shifts in a single operation, then all the odd (see Figure 6-2). For each operation, we must consider each of m segments, so this algorithm requires $O(m)$ operations. The runtime remains at $O(nm)$, but may be parallelized to $O(n)$. Please see Figure 6-2 for graphical steps of the algorithm.

6.4 Parameter estimator

We implemented a trinary search algorithm for searching through the k and m space. As shown in Figure 6-3, the cost falls off exponentially as m increases. The objective of the parameter estimation is to find the elbow of the curve. We achieve this by searching for the point along the curve that is farthest away from a reference line defined by extreme m (small and large). Figure 6-4 shows the results of maps generated from different m.

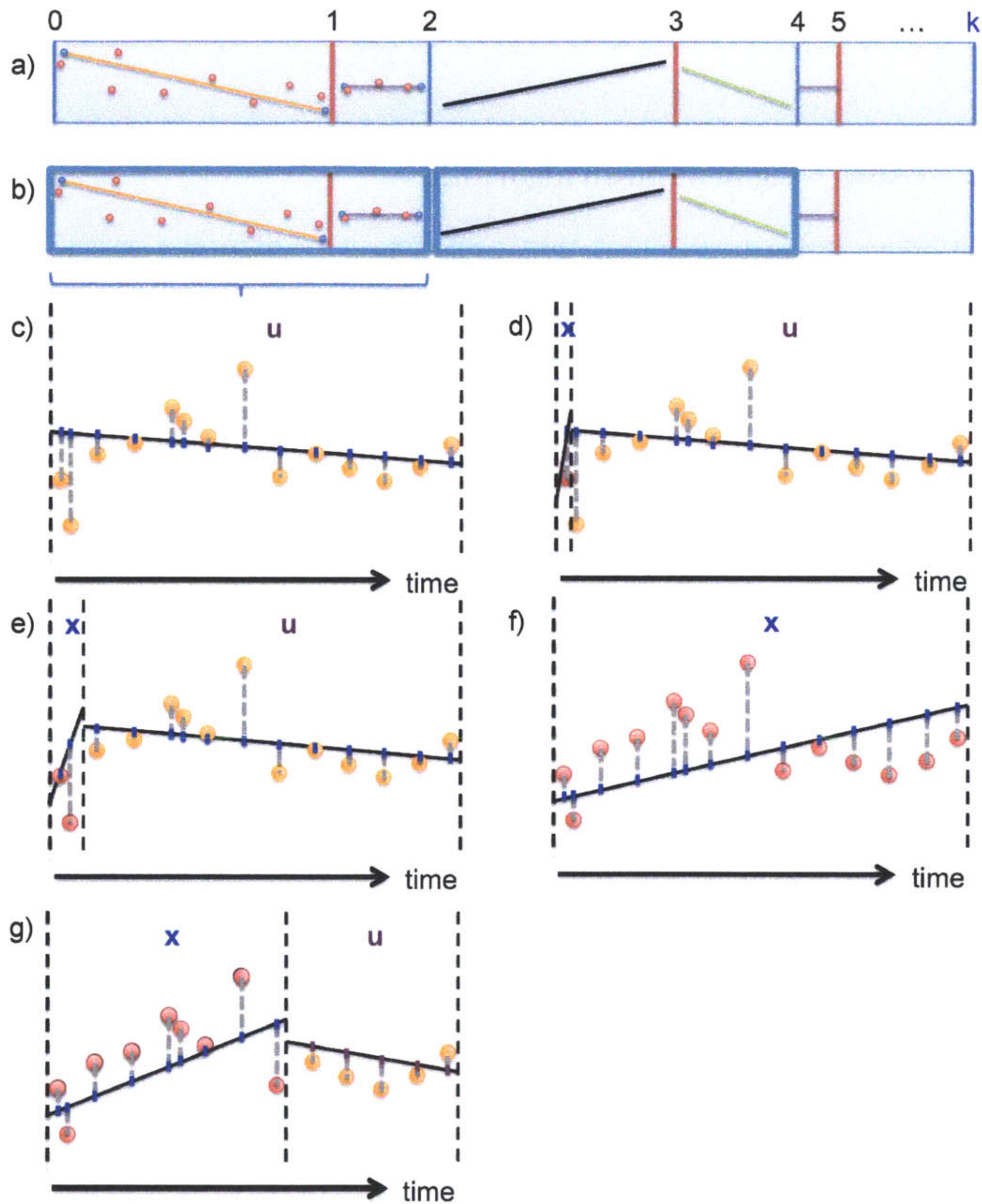


Figure 6-1: *(a)* Shows the input, which consists of n points, partitioned into k parts, each with an initial assignment of to one of m clusters. *(b)* We split the input into $\lceil \frac{k}{2} \rceil$ 2-segment mean problems and we show the steps of computation of one of the subproblems in *(c)-(g)*. *(c)* We initialize our right 1-segment coreset in $O(n)$ time to compute the cost for all of the n points. *(d)* We update our right coreset by removing the leftmost point and we update/initialize our left coreset by adding the leftmost point, which takes O(1) time. We compute the cost using the coreset, which costs O(1). *(e)* We update our right coreset by removing its leftmost point and we update our left coreset by adding the next point to its right, which takes O(1) time. We compute the cost using the coreset, which costs O(1). *(f)* We continue in this manner until the right coreset is empty. Now we have n costs. *(g)* Finally, we compute the partition that yields the minimum cost.

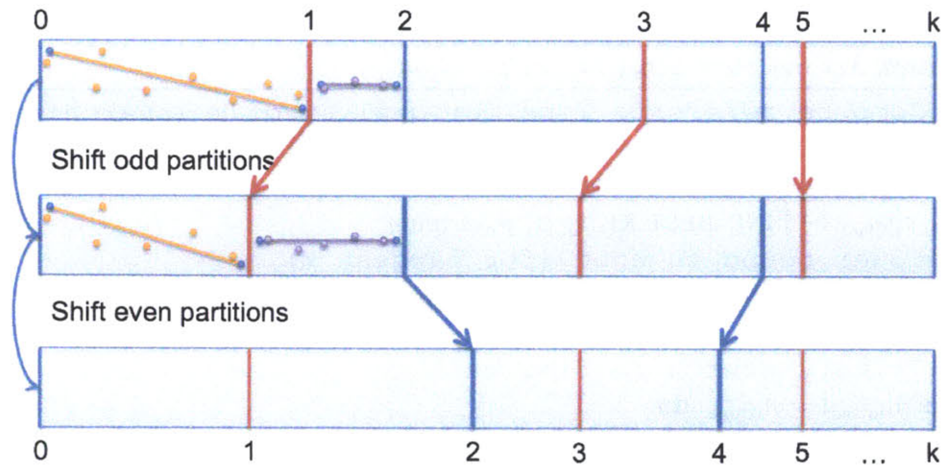


Figure 6-2: The partitioning stage of the (k,m)-segment mean algorithm consists of finding the best partition of the input into k parts given m possible segments to use. Instead of a computationally expensive full search, we shift the partition bars locally. We **parallelize** this process by shifting all the odd numbered partitions at once (red), then the even partitions (blue). Here, we show the k parts of the input; the first few parts contain points for demonstrative purposes (orange and purple). Note that the points stay in place between iterations, but the selected segments are scaled accordingly (to the length of the partition)

Algorithm: The algorithm takes in a range of k and m values to search through and outputs the optimal k and m. We perform a nested binary search, first for the parameter m, then for the parameter k. For each m, there is an optimal k [16], so although the parameter search is nested, the two parameters are independent. The best parameter is determined by a density-based function [5], where we return the farthest point from a reference line defined by the parameter extremes. The parameter search process for k and m are identical. See Algorithm 3 and its helper Algorithm 4 for more details.

Analysis: The time complexity of the parameter estimator is $\mathrm{O}\big(nmi_{end} \cdot \log^2(n)\big)$. The (k, m)-segment mean algorithm requires $\mathrm{O}\big(nmi_{end}\big)$ and each nested parameter search adds a $\mathrm{O}\big(\log(n)\big)$ factor.

Memoizer: We have implemented a data structure for saving learned (k,m)-segment mean and the associated cost for specific (k,m) parameters. This memoizer structure allows for quick retrieval of (k, m)-segment results if the same configuration had run previously, which is extremely useful and practical during parameter search.

Algorithm 3: Find_best_m($m_0, m_n, k_0, k_n, input$)

```
   /* Compute reference line for quality of m (searching
   for the elbow in curve)                                          */
   /* Upper bound on cost (for smallest m)                          */
 1 (k_m0, cost_0) ← FIND_BEST_K(m_0, k_0, k_n, input)
   /* Lower bound on cost (for largest m)                           */
 2 (k_mn, cost_n) ← FIND_BEST_K(m_n, k_0, k_n, input)
 3 m_lower ← m_0, cost_lower ← cost_0, m_upper ← m_n, cost_upper ← cost_n
   /* Trinary search to find best m                                 */
 4 while m_lower != m_upper do
       /* Split search space into 3 parts                           */
 5     (m_a, m_b) ← TRISECTION(m_lower, m_upper)
       /* Cost for 2 intermediate m values                          */
 6     (k_a, cost_a) ← FIND_BEST_K(m_a, k_0, k_n, input)
 7     (k_b, cost_b) ← FIND_BEST_K(m_b, k_0, k_n, input)
 8     dist_a ← NORMALIZED_DISTANCE(m_a, cost_a, m_0, cost_0, m_n, cost_n)
       /* Evaluate vs reference line */
 9     dist_b ← NORMALIZED_DISTANCE(m_b, cost_b, m_0, cost_0, m_n, cost_n)
       /* Remove third of search space farther from elbow           */
10     if dist_a > dist_b then
11         m_upper ← m_b, cost_upper ← cost_b
12     else
13         m_lower ← m_a, cost_lower ← cost_a
14 return (m_upper, cost_upper)
```

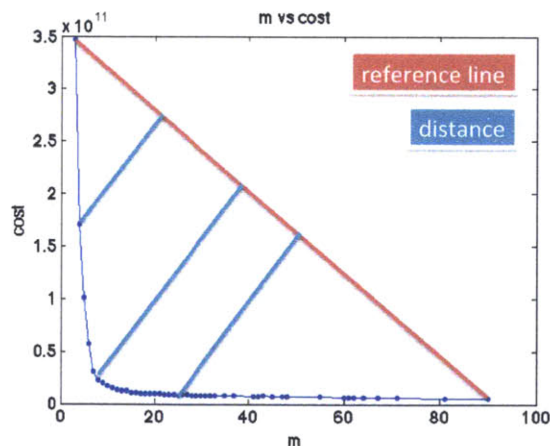


Figure 6-3: Shown is a sample m vs cost curve. The parameter estimator searches for the elbow of the curve by using a trinary search to look for the point farthest away from the reference line (red). Distances are shown in light blue. In this example, $m = 7$ is located at the elbow.

Algorithm 4: `Find_best_k`$(m_0, k_0, k_n, input)$

```
   /* Compute reference line for quality of k (searching
   for the elbow in curve)                                        */
   /* Upper bound on cost (for smallest k)                        */
1  cost_0 ← KM_SEG_EM((m_0, k_0, input)
   /* Upper bound on cost (for smallest k)                        */
2  cost_n ← KM_SEG_EM(m_n, k_n, input)
3  k_lower ← k_0, cost_lower ← cost_0, k_upper ← k_n, cost_upper ← cost_n
   /* Trinary search to find best m                               */
4  while k_lower! = k_upper do
       /* Split search space into 3 parts                         */
5      (k_a, k_b) ← TRISECTION(k_lower, k_upper)
       /* Cost for 2 intermediate k values                        */
6      (k_a, cost_a) ← KM_SEG_EM(m_0, k_a, input)
7      (k_b, cost_b) ← KM_SEG_EM(m_0, k_b, input)
8      dist_a ← NORMALIZED_DISTANCE(k_a, cost_a, k_0, cost_0, k_n, cost_n)
       /* Evaluate vs reference line */
9      dist_b ← NORMALIZED_DISTANCE(k_b, cost_b, k_0, cost_0, k_n, cost_n)
       /* Remove third of search space farther from elbow         */
10     if dist_a > dist_b then
11         k_upper ← k_b, cost_upper ← cost_b
12     else
13         k_lower ← k_a, cost_lower ← cost_a
14 return (k_upper, cost_upper)
```

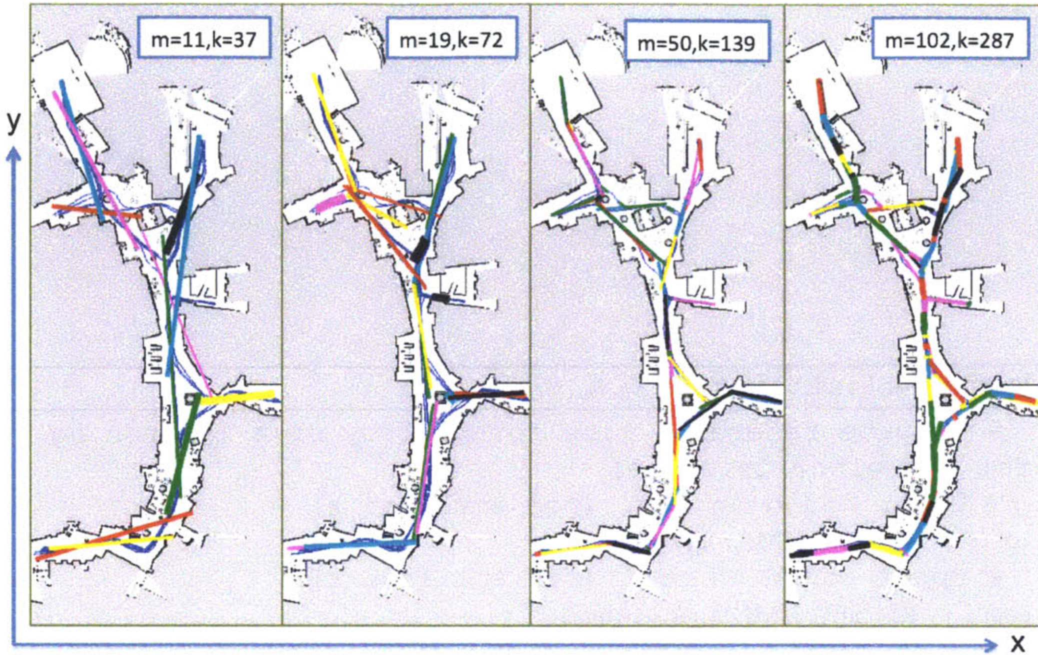


Figure 6-4: Shown here is the full original indoors trajectory (blue) projected onto x-y, as well as the m segments that form the map (colored segments) for several different m. Notice the trade-off between compression size and compression quality. Smaller m yields larger errors. Our parameter estimator selected $m = 19$.

6.5 Map construction

Algorithm: To construct a connected map from the m states and $k-1$ transitions: We add $2m$ nodes $V_m = \{v_1, v_2, ..., v_{2m}\}$ for each endpoint of the m segments. Then, we add m edges that link the $2m$ endpoints. Now, to connect the graph, we add an edge to the map for each unique transition (t_i, t_i') such that $t_i, t_i' \in V_m$ for $1 \le i \le k-1$. Adding auxiliary edges represents traversals as legal transitions in the map. Repeated auxiliary edges are added only once. Note that the set of consecutive edges may contain redundant pairs, especially if the agent is performing repeated tasks. See Algorithm 5 for the psuedocode and Figure 6-5 for an example.

Analysis: The map requires $\mathrm{O}(m+k)$ time to construct. The number of edges in the resultant map is bounded by the smaller of $m+k$ or $\frac{m(m-1)}{2}$ (fully connected). The number of nodes is $2m$. Since $m \le k$, the size complexity of the map is $\mathrm{O}(k)$ for edges and $\mathrm{O}(m)$ for nodes.

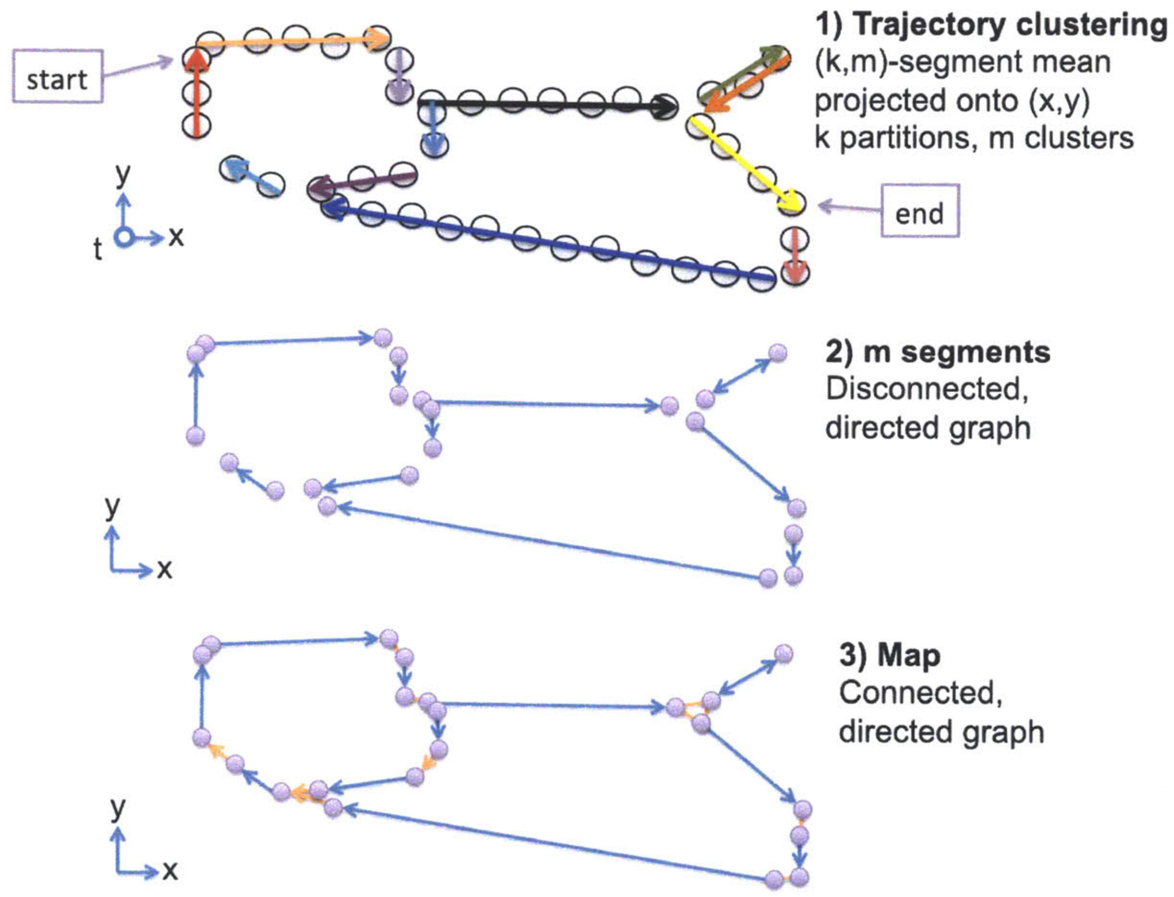


Figure 6-5: Shown here is an example of map construction, which is the last step of the trajectory mapping stage. The resulting information-rich map is then used to further compress the input stream. *Top*: The (k, m)-segment mean produced by the trajectory clustering algorithm, when projected onto x-y, consists of m directed segments. *Middle*: The m directed segments (blue edges) from the (k, m)-segment mean form part of the map, but they lie disconnected. The endpoints of the m clusters form the nodes of the graph (purple). *Bottom*: By following the order of the k segments (each of which is one of the m clusters), the transitions between one cluster to the next yields additional auxiliary edges (orange) to the graph that connect the m clusters. The resulting approximation of the real map is a graph with nodes (end points of the segments (purple)) and edges (the segments (blue) and any transitions between the segments that may have occured during the trajectory (orange)).

Algorithm 5: `Make_map`$(input, k, m)$

```
/* Endpoints of segments form the set nodes                */
```
1 $nodes \leftarrow clusters[:, 1] + clusters[:, 2]$

```
/* Cluster segments are definitely part of the map         */
```
2 $edges \leftarrow clusters$

```
/* Add missing edges for each unique transition between
clusters                                                    */
```
3 **for** $i \leftarrow [1...k-1]$ **do**

4   **if** ***not*** $(partition[i], partition(i+1))$ ***in*** $edges$ **then**

5         $edges \leftarrow edges + (partition[i], partition(i+1))$

6 **return** $(nodes, edges)$

Chapter 7

Trajectory compression

In this Chapter, we propose algorithms for the problems defined in Chapter 4 relevant to the trajectory compression, which forms the second half of our semantic compression system (see the Trajectory compression block in Figure 2-1). For compressing the input against the map, we suggest an algorithm based on binary search and the algorithm in [25]. The algorithm has worst case of polynomial time, but in practice (average case) usually runs in linear time given its A* heuristics and the sparseness of the map.

7.1 Trajectory compression using map matching

Algorithm: Given a connected map generated from the (k, m)-segment mean, we apply a HMM-based map matching algorithm [25] on the input sequence of k segments ($2k$ points) to compute the trajectory that most likely describes the moving object in time in terms of nodes in the map. Then, using a recursive binary search, we prune out intermediate points within the trajectory are redundant information, given the map. See Algorithm 6 for the psuedocode.

Analysis: The time complexity of the trajectory compression algorithm is $O\big(n \cdot \|E\| \cdot \log(n)\big)$. The map matching algorithm itself requires $O\big(n \cdot \|E\|\big)$, since A* requires $O\big(\|E\|\big)$, where n is the length of the input and $\|E\|$ is the number of edges in our map. The additional $\log(n)$ factor comes from the recursive binary search for compressing the trajectory. Additionally, since we only prune out intermediate nodes in the trajectory that are encoded

Algorithm 6: `Traj_compress`(seq, map)

```
1 traj ← MAP_MATCH(seq, map)   /* traj is a sequence of edges */
  /* seq is updated with non-noisy coordinates */
2 seq ← GET_COORDINATES(traj)
3 if MAP_MATCH(seq, map) = MAP_MATCH([seq.first, seq.last], map) then
  |  /* End points capture all information (given the map)
  |     */
4 |  return (seq.first, seq.last)
5 else
  |  /* End points do not capture enough information, so
  |     recurse */
6 |  return
  |  MAP_MATCH(seq.first_half, map) + MAP_MATCH(seq.last_half, map)
```

in the map (recoverable by map matching), our trajectory compression algorithm is lossless.

Our algorithm is always applied to the input of k segments rather than the original input of size n, and our map is size $O(k)$, so the complexity of this system component is $O(k^2 \cdot \log(k))$.

Map matching algorithm: The trajectory compressor algorithm invokes a black box map matching algorithm. For our system, we used a HMM-based map matching implementation, detailed in [25]. Our choice of map matching algorithm additionally allows our system to compress trajectories that were *not* used to create the map. The HMM-based map matching algorithm we selected is essentially A* search married with the Viterbi algorithm, with the heuristic that shorter paths are the ones that agents realististically prefer. In our semantic representation and compression system, we only use the A* and the heuristic aspects. However, the embedded Viterbi algorithm allows us to handle noise in the inputs, i.e. points that do not directly correspond to nodes in the graph, by applying dynamic programming to find the most likely sequence of hidden states. This is useful, for example, in long term robot operation, where the initial sensor readings can be used to build up a map, and the sensor streams collected thereafter may bypass the trajectory mapping stage and pass directly to the trajectory compression stage. The Viterbi feature adds an additional factor of $O(\|V\|^2)$ to the runtime, for an overall map matching runtime of $O(n \cdot \|V\|^2 \cdot \|E\|)$, where $\|V\|$ is the number of nodes in our map. Note that this use of our system is *not* lossless.

7.2 Trajectory decompression

Algorithm: The trajectory decompression algorithm is simply the map matching algorithm, the same algorithm used in the trajectory compression algorithm. The map matching algorithm finds the shortest paths between points according to our map and the full trajectory is the decompressed output. Since we only pruned out intermediate nodes in the trajectory that are recoverable by map matching, our trajectory compression algorithm is lossless.

Analysis: The map matching algorithm requires $\mathrm{O}(n \cdot \|E\|)$, since A* requires $\mathrm{O}(\|E\|)$, where n is the length of the input and $\|E\|$ is the number of edges in our map.

Chapter 8

Experiments

8.1 Overview

We performed a series of experiments to demonstrate the flexibility and scalability of our compression. We start with robot trajectories, both indoors and outdoors, which may be useful for helping pinpoint incidents and decision making in real time. Next, we examine how our system can be used for personal mobility applications, for example summarizing the mobility patterns of people. We perform semantic compression on a personal GPS stream collected over several months. Finally, we examine how our system can be used for map generation applications, for example where maps may not yet exist or where there may be ongoing changes to existing maps. Here, we perform semantic compression on GPS trajectories collected from taxicabs over several days in Singapore. We explain most of our results in terms of the ground robot (indoors) experiment.

We evaluate the (k,m)-segment mean results on compression quality, since the output may be used for real-time decision making or other processing. We discuss the implications of compression loss on compression quality.

8.2 Data sets

We collected and selected datasets of varying sizes, settings, and scopes. We give a summary of the data sets, followed by more details on data collection and data quality. Our

main experiment is on the ground robot (indoors), and most of our results are explained in terms of this dataset. The quadrotor robot experiments shows that our system performs well outdoors. The personal smartphone experiment shows the ability of our system to effective summarize a person's life in terms of their movement. The taxicab GPS experiment shows the ability of our system to generate a sketch of a city map.

- **Ground robot**: 72,273 points, localized from SLAM, first floor of an academic building, indoors only, 4 hours.
  Using the Robot Operating System [28], we employed the GMapping simultaneous localization and mapping package [4] on a custom built omnidirectional ground robot equipped with a 30 m capable Hokuyo scanning laser rangefinder (see Figure 8-1a). The SLAM algorithm was used to create the baseline for the location data for the indoor experiment. We remotely teleoperated the robot to concurrently map and patrol the first floor of an academic building. Over the course of the 90 minute run, the robot travelled over 1.2 km and recorded a two-dimensional path composed of over $70,000$ points. As illustrated in Figure 8-1c, the path contains localization noise, repeated traversals, and various loops.

- **Quadrotor robot**: 12,000 points, onboard GPS unit, courtyard of an academic building, outdoors only, 10 minutes.
  We equipped an Ascending Technologies Pelican quadrotor flying robot with a single board computer (SBC) and a Samsung Galaxy SIII smartphone; see Fig. 8-1b. We then manually piloted the robot for approximately 10 minutes above an outdoors courtyard. During the flight, the onboard SBC running the Robot Operating System (ROS) [28] recorded filtered GPS readings at 20 Hz from the robot's high level processor. Also recorded was the smartphone's downward facing video stream, which we used to visually verified the resulting GPS path of the robot. Overall, a sensor stream of $12,000$ GPS points was acquired from the hardware experiment for a medium-size run.

- **Personal smartphone**: 26,249 points, GPS from smartphone, Greater Boston area, indoors and outdoors, 8 months.

For the personal GPS experiment, we loaded a smartphone application called Travvler that performs data logging to a local server. We allowed an individual to collect their GPS location at a frequency of approximately 30Hz over 8 months. Due to frequent data drops (and thus, gaps in the signal), we additionally employed NAVTEQ's map of Massachusetts for data patching as a preprocessing step.

- **Taxicab GPS**: 4,281 points, onboard GPS unit, Singapore metropolitan area, outdoors only, 5 days.
  We additionally tested our system on data collected from Singapore taxicabs, which was collected at approximately 1Hz over the period of a month. Due to the sparseness of the resulting data (from the low collection frequency), we additionally employed NAVTEQ's map of Singapore for data patching as a preprocessing step.

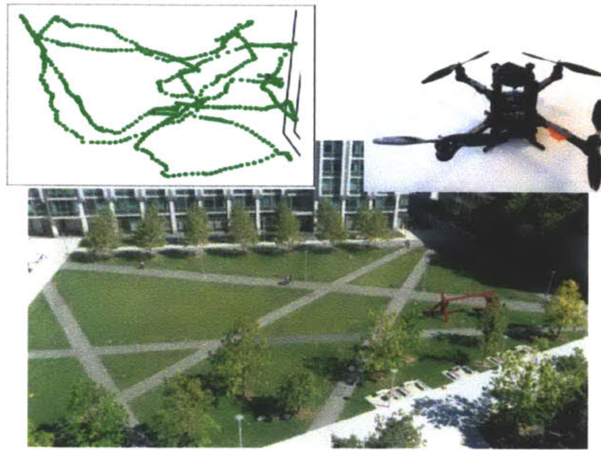
8.3 Processing

All the data processing and algorithms are run on commodity machines with 64-bit Linux or Mac OS X and maximum computing power of a 2.3GHz Intel®Core™i7 processor and 8GB RAM. The main development tools are MATLAB (R2012b) and Python (2.7.3).

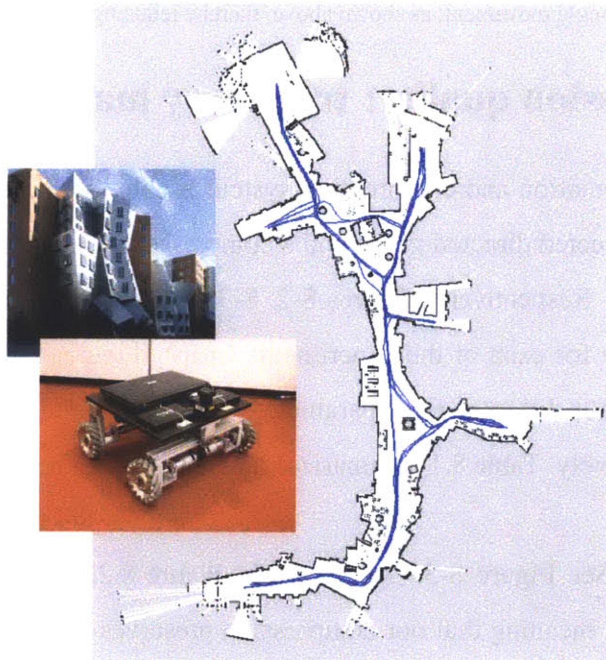
System: We implemented the components of the semantic representation and compression system in this thesis: the coreset construction (MATLAB), data patching (Python), (k, m)-segment mean algorithm (MATLAB), parameter estimation (MATLAB), the map construction (MATLAB), and the map compression (Python).

Data collection: We have also implemented modules for data collection and management in MATLAB. The location information is aggregated from a GPS data stream for the outdoor experiment, and from a SLAM implementation for the indoor experiment.

We run the algorithms as described in Section 2, extracting the "important" information and generating a map from each input sequence (see Figure 8-3 and 8-2). We verify that we recover the semantic trajectory in the decompression step. With tens of thousands of input points, the runtime of the entire flow is several hours.



(a)



(b)

Figure 8-1: *Top:* Our quadrotor flying robot equipped with an onboard computer and smartphone collects GPS data in a courtyard. *Bottom:* An omnidirectional ground robot equipped with laptop and scanning laser rangefinder performs SLAM in an academic building. The resulting occupancy grid map and recorded path of the ground robot.

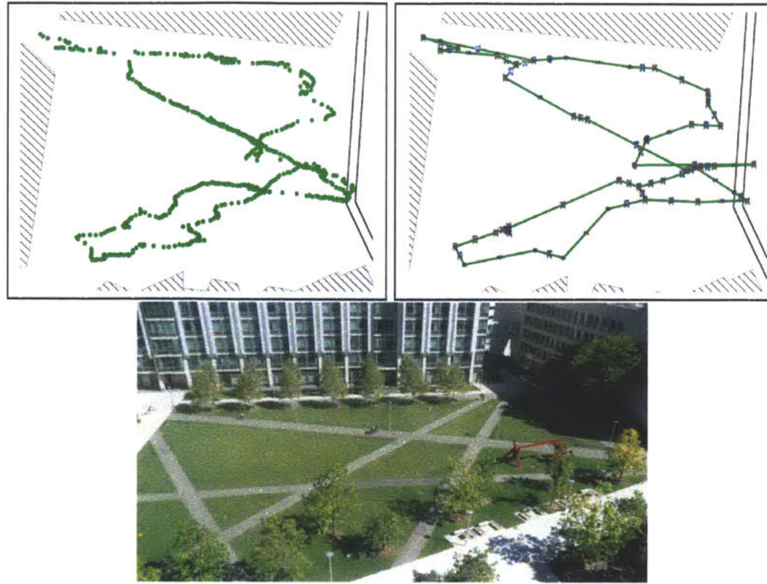


Figure 8-2: Raw input location stream from the quadrotor in courtyard area (left), the corresponding generated map (center), and a view of the courtyard where the data was collected (right). Repeated movements are captured in the map as a single movement, as shown above, thereby reducing the noise from the original data.

8.4 Compression quality: trajectory mapping results

Our semantic representation and compression system is able to efficiently summarize the four data sets as connected directed graph and a compressed trajectory of points.

All experiments: Respectively, Figures 8-2, 8-3, 8-4, 8-5 show the visuals of the semantic representation for each of the experiments (map). Figures 6-4 and 8-5 also show different summaries for different error parameters for the indoors building and city map experiments, respectively. Table 8.1 summarizes the compression results with error parameters.

Ground robot: See Figure 8-3. As shown in Figure 8.2, we targeted an RMSE of approximately $20cm$, meaning that our compression preserves the original trajectory with an average error of $20cm$. For many robotic applications such as patrolling a building or courtyard, a $20cm$ tolerance is more than sufficient to allow for real-time decision making from reported statuses and incidents. Since we encode information as a directed graphical map instead of an occupancy grid map, it is computationally simpler to process. In fact, it is sometimes also more intuitive to report the general location than a specific (x, y) coordinate, e.g. using road name, mile marker, and direction to report traffic incidents.

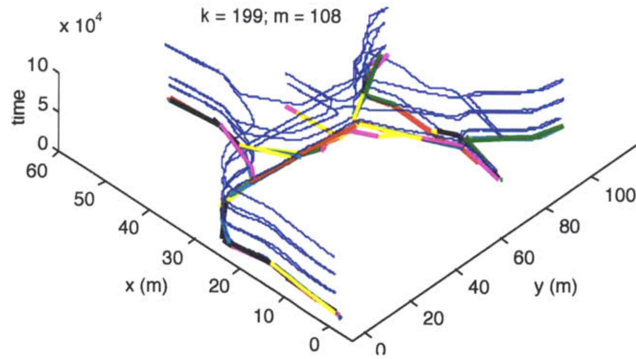


Figure 8-3: Shown here is the full original trajectory in time (blue), as well as the m segments that form the map (not blue). Paths that are traversed several times are automatically summarized with a single path that represents all the traversals (with bounded error).

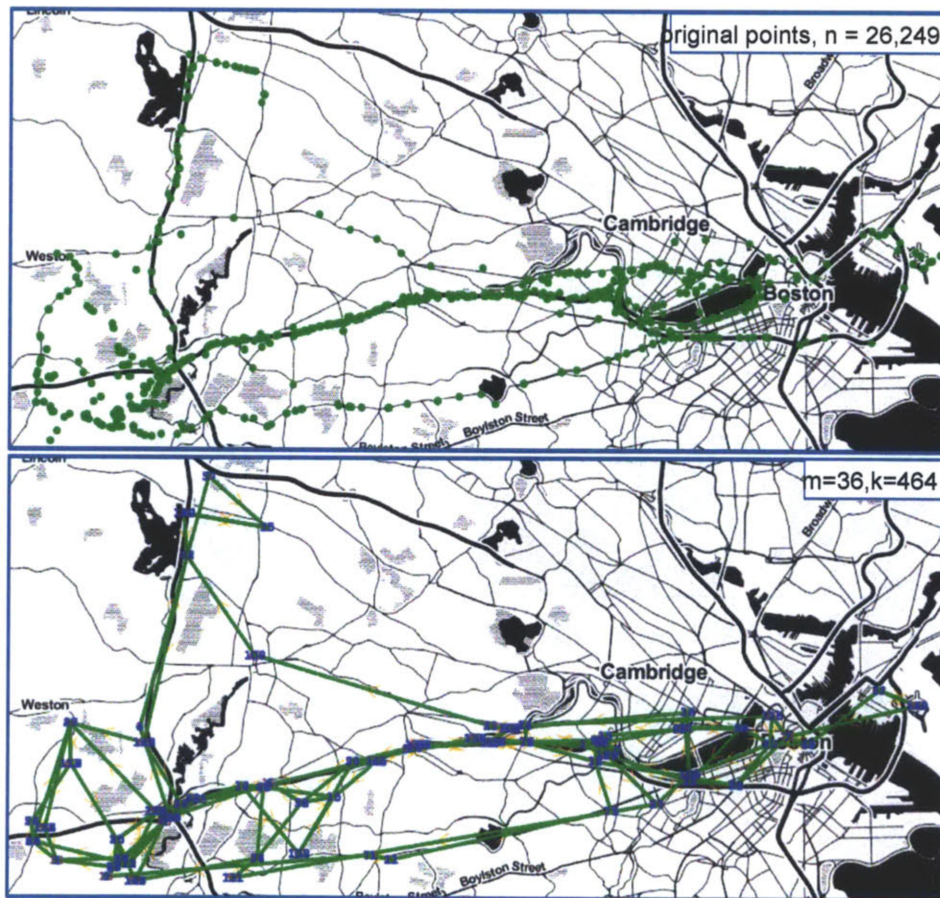


Figure 8-4: Semantic compression of personal GPS trajectory.

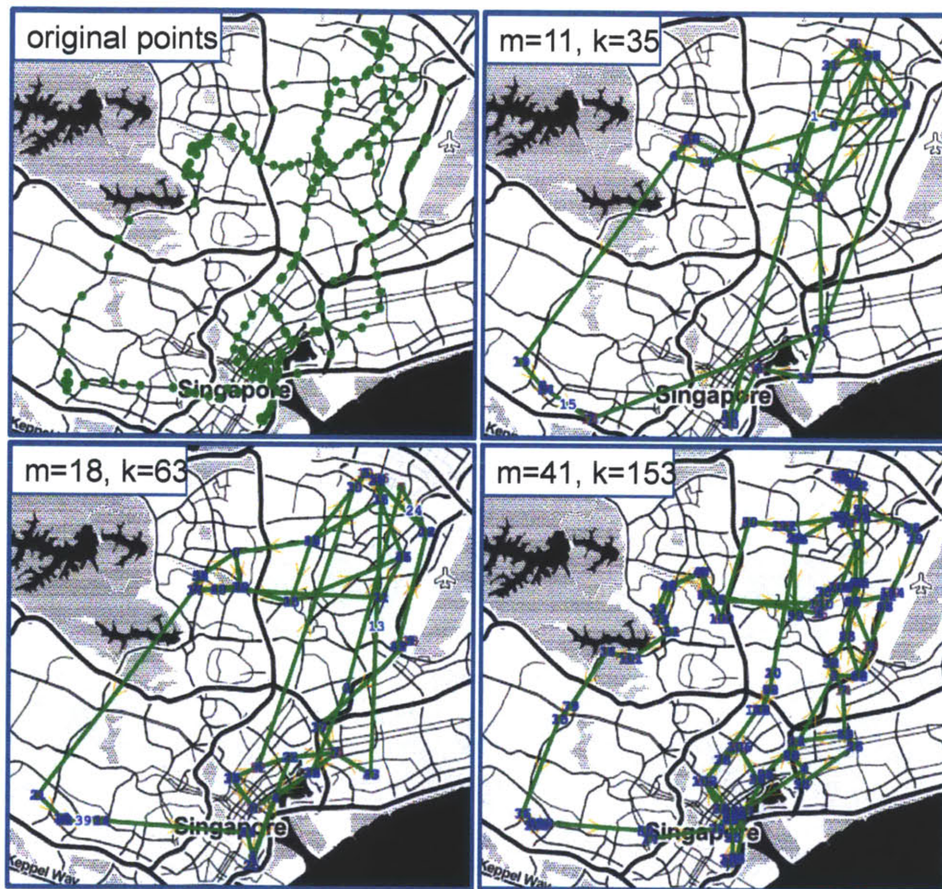


Figure 8-5: Semantic compression of a Singapore taxicab trajectory.

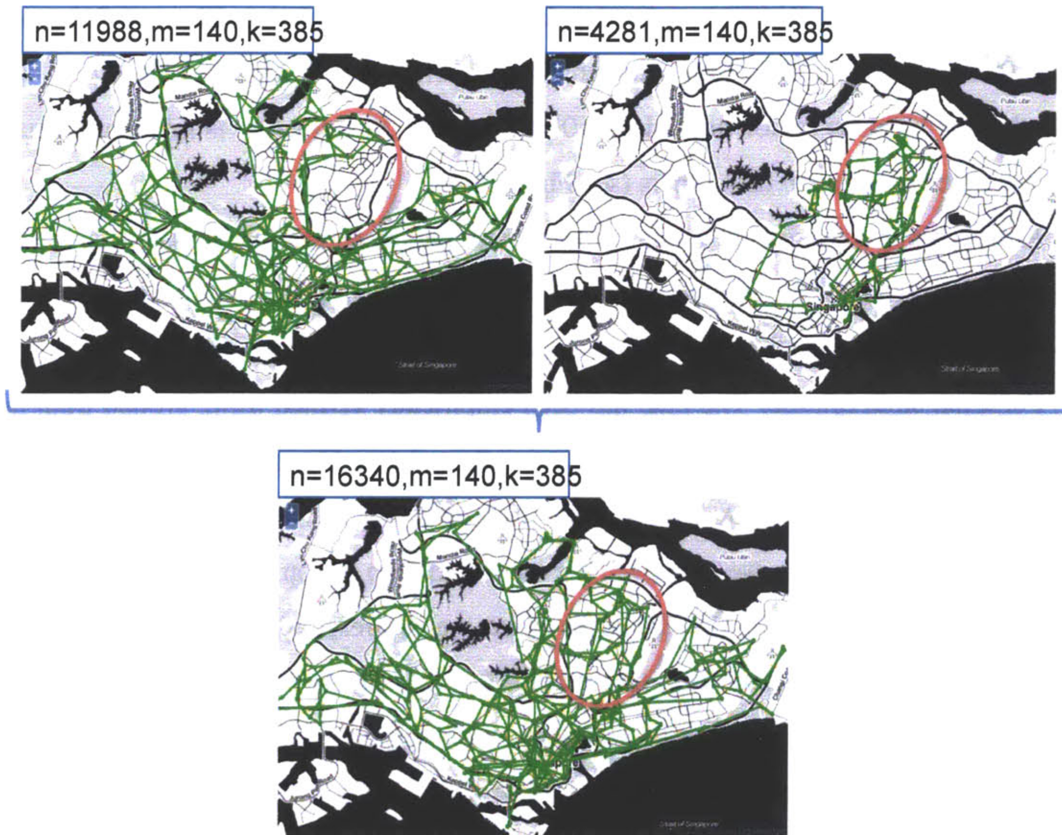


Figure 8-6: Merging capability of our semantic representation and compression system. The trajectories of two taxis (top) are merged into a single more complete map (bottom). Note the circled area, which is lacking in segments in the top left taxi's trajectory but abundant in the top right taxi's trajectory. Together (bottom), their map is more complete.

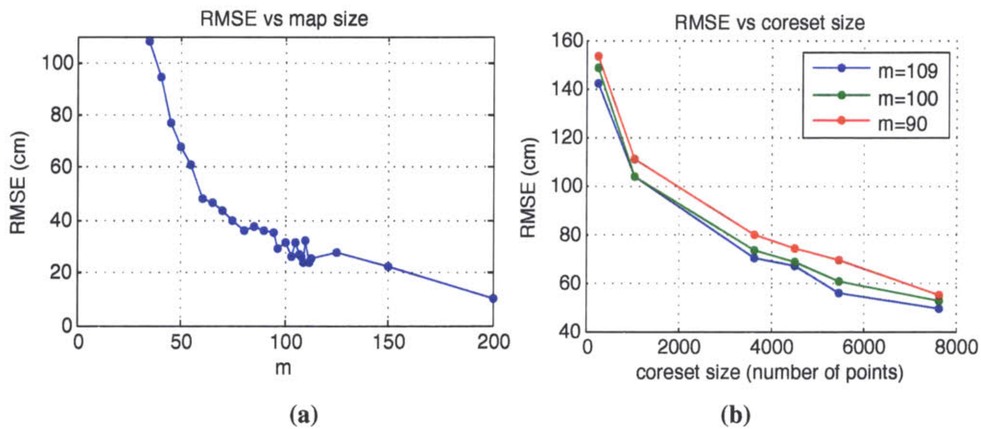


Figure 8-7: *Left:* The average error drops off exponentially as the size of the map increases (for k=200). *Right:* The average error drops off exponentially as the size of the coreset increases. The trend is independent of the choice of m, as shown for k=200.

We can control the RMSE by adjusting k, m, and the coreset size. Figure 8-7 demonstrate that adjusting m and coreset size predictably affects the RMSE. In Figure 8-7a, we note that the unstable region around m=100 shows that the trajectory is inherently better approximated by some m than others (see Figure 6-4).

Although our compression is not lossless, these parameters can be adjusted according to the requirements for different applications. Increasing k, m, and coreset size reduces the RMSE, but will increase the runtime and the size of the compression.

Personal smartphone: The map has been empirically verified using ground truth information from the individual whose data we collected (see Figure 8-4). Locations such as home, work, friend's house, fitness center, as well as specific routes between locations were verified. In addition, we overlayed the computed map on top of an actual map of the Greater Boston area to verify that the edges of our map correspondes to streets.

Taxicab GPS: As ground truth information is not available for this dataset, we overlayed the computed map on top of an actual map of Singapore using OpenStreetMap [1] to verify that the edges of our map corresponds to streets (see Figure 8-5). With multiple taxis, we may actually merge their trajectories together for a more complete map (see Figure 8-6) by using the (k, m)-segment mean result from each individual taxi.

8.5 Compression size: trajectory compression results

We additionally perform trajectory compression on the data sets, following trajectory mapping (See Sections 8.2 and 8.3 for data set and processing details). We evaluate the trajectory compression results on compression size, since the output is primarily useful for data logging and offline storage. We measure the effectiveness of our system in addition to bzip2 versus compression using only bzip2, which is an industry standard for lossless text compression. This gives us a measure of the semantic redundancy in the data, rather than the textual redundancy.

Experiment	Number of Points			Map size		Compression ratio			**Improve-ment**	RMSE	Running time
	original	coreset	comp	nodes	edges	bzip2	semantic	sem+bzip2			
Ground robot	72,273	7,651	52	216	258	3.8	237.0	646.2	**167.4**	24 cm	7.4 hr
Quadcoptor robot	12,000	12,000	100	40	21	11.9	328.5	709.0	**59.7**	10 m	2.1 hr
Personal smartphone	26,249	6816	907	336	622	3.0	10.5	27.2	**9.1**	0.3 km	9.7 hr
			291	60	119		54.81	132.21	**44.2**	0.8 km	9.8 hr
Taxicab GPS	4,281	4,281	103	270	292	7.36	12.1	33.1	**4.5**	41 m	2.3 hr
			47	40	50		79.1	171.0	**23.2**	0.2 km	2.4 hr
	16,340	16,340	677	358	551	6.4	26.8	67.8	**10.6**	0.2 km	14.5 hr

Table 8.1: Experimental results. Our semantic compression performs consistently better than bzip2 (in size) and may be layered with bzip2 for further compression. The **coreset** column indicates the size of the k-coreset computed in the preprocessing stage, and **comp** indicates the length of the resulting compressed trajectory. The **improvement** metric is the measure of how well semantic+bzip2 compresses as compared to just bzip2, or in other words, how much using our compression on top of bzip2 improves the baseline compression. For smaller error parameters, the improvement will be smaller. For applications that tolerate larger error, the improvement may be drastic. Except for the quadcoptor experiment (trajectory clustering, map construction, and trajectory compression only), the listed **running time** is for the entire semantic representation and compression system, including preprocessing, trajectory mapping (and parameter search), and trajectory compression.

First % of points	Number of Points original	compressed	Map size	Compression ratio bzip2	semantic	sem+bzip2	**Improvement**	RMSE (cm)
10%	7,227	2	16 nodes, 15 edges	4.1	380.4	651.7	**158.1**	19.05
20%	14,454	3	48 nodes, 47 edges	3.8	254.9	584.7	**152.3**	21.30
30%	21,681	4	70 nodes, 69 edges	3.7	263.0	638.4	**170.0**	21.90
40%	28,909	5	100 nodes, 105 edges	3.8	243.4	602.4	**158.1**	21.71
50%	36,136	13	136 nodes, 141 edges	3.8	213.8	547.8	**143.5**	21.55
60%	43,363	10	178 nodes, 194 edges	3.8	195.8	521.2	**135.9**	19.23
70%	50,591	24	194 nodes, 215 edges	3.8	200.5	536.9	**137.9**	20.33
80%	57,818	28	206 nodes, 238 edges	3.8	211.6	567.9	**147.0**	22.35
90%	65,045	23	216 nodes, 255 edges	3.8	229.6	619.2	**160.4**	20.89
100%	72,273	52	216 nodes, 258 edges	3.8	237.0	646.2	**167.4**	23.60

Table 8.2: Ground robot: As more of the robot's positions and movement patterns are captured, the compression ratio improves (60%-100%). Our compression also works well for simple paths (first 30%). No k-coreset was computed for this experiment.

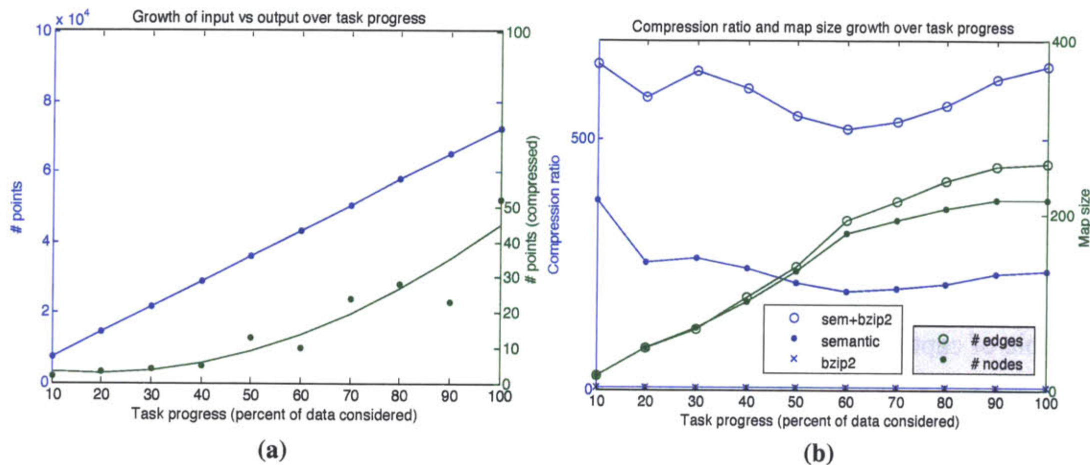


Figure 8-8: We compress subsets of the ground robot data set to demonstrate how our system scales over time and to simulate the online operation of our system. *Left*: Growth of input size vs output size as the task progresses. *Right*: Compression ratio trend and growth of map size as task progresses for the ground robot experiment.

All experiments: The results are summarized in Table 8.1. Our semantic compression far outperforms bzip2 in every test with a small RMS error. In the next section, we present and explain our compression ratio in the context of a patrolling application.

Ground robot: Our compression result (semantic compression cascaded with bzip2) is 646.2 times smaller than the original trajectory and 167.4 than performing bzip2 on the original trajectory. We factor in the size of both the sequence of coordinates and the map. Starting with a trajectory of over 72K points, our compression is a sequence of 52 points along with a small map. Our system continues to perform well as more data is collected, as shown in Figure 8-8.

8.6 Intuition for trajectory compression

To give better intuition of our compression algorithm, we perform compression on increasing subsets of the ground robot experiment, as detailed in Table 8.2. The data exhibits 3 main stages: initial exploration (increase in map size), initial task (increase in map size and compressed points), and task repetition (increase in compressed points).

Initial exploration: The first 10% of the data is simply a path with no loops or crosses, so we compress this as the first and last points (2 points) and a few straight segments that

approximate the curve in between (the map). Since there is very little semantic information in a simple path, our algorithm far outperforms bzip2, which does not take semantics into account, by a factor of 158.1. Our algorithm continues to take advantage of the relatively little semantic information in the trajectory until 30% of the run, compressing 22K datapoints to 4 points and a small map, a 170.0 times improvement over bzip2. During this stage, we primarily add nodes and edges to the map; a very small number of points is capable of capturing the semantics of the trajectory.

Initial task: As the robot patrols its environment, the trajectory becomes more complicated (adding loops and crosses), which increases both the size of the map and the number of points in our compression. WIth a more complicated trajectory, the compression must retain more points in order to differentiate between multiple possible routes through the map. Thus between 30% and 60%, the improvement over bzip2 dips from 170.0 to 135.9.

Task exploration: As the environment becomes increasingly explored over time, the trajectory continues to add points to the compressed result but adds little information to the map. By 90% of the run, the environment has mostly been explored, as seen by the relatively small change in map size between 90% and 100%. From 60% to 100% of the run, we see a steady improvement ratio over bzip2, from 135.9 to 167.4. We expect that this ratio would continue to increase as the robot patrols its already explored environment.

Chapter 9

Conclusion

This thesis describes a scalable map making algorithm using semantic compression as a way of coping with raw data size and representation size for long-duration robot operations. We show how coresets and (k,m)-segment algorithms cope with data error and uncertainty by compressing sensor streams with bounded error to critical points. Our compression algorithm first builds a map from the (k, m)-segment mean generated from the raw data (trajectory mapping). It then uses the map topology to further compress the data stream (trajectory compression).

We improve upon previous work in compression size of sensor streams and argue that the compression quality may be easily adjusted to be "good enough." We conducted several experiments to demonstrate the scalability of our systems and to introduce some possible applications. In our main ground robot experiment, our compressed representation has a 167.4-fold size reduction when compared against the popular compression tool bzip2. Our result is 646.2 times smaller than the original datastream. We additionally demonstrate the capability to automatically summarize a personal GPS stream, generate a sketch of a city map, and merge trajectories from multiple taxicabs for a more complete map.

By carefully selecting points according to the generated map, we are additionally able to guarantee a connected and semantically correct trajectory. Our algorithm is most effective on data streams with repetitions or data streams with very little semantic information. It is least effective on data streams where the states in the state model are fully connected because in such cases there is little redundant information not already captured by the map.

Bibliography

[1] Open StreetMap.

[2] Current map-matching algorithms for transport applications: State-of-the art and future research directions. *Transportation Research Part C: Emerging Technologies*, 15(5):312 – 328, 2007.

[3] rosbag Commandline Documentation, March 2012.

[4] ROS: gmapping package, January 2013.

[5] Pankaj K Agarwal and Nabil H Mustafa. k-means projective clustering. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 155–165. ACM, 2004.

[6] T. Asano and N. Katoh. Number theory helps line detection in digital images. In *Proc. 4th Ann. Intl. Symposium on Algorithms and Computing*, pages 313–322, 1993.

[7] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, SODA '97, pages 360–369, 1997.

[8] James Biagioni and Jakob Eriksson. Inferring road maps from gps traces: Survey and comparative evaluation. In *Transportation Research Board 91st Annual Meeting*, number 12-3438, 2012.

[9] J. Borenstein and Liqiang Feng. Measurement and correction of systematic odometry errors in mobile robots. *Robotics and Automation, IEEE Transactions on*, 12(6):869 –880, dec 1996.

[10] Sotiris Brakatsoulas, Dieter Pfoser, Randall Salas, and Carola Wenk. On map-matching vehicle tracking data. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 853–864, 2005.

[11] M. Burrows, D. J. Wheeler, M. Burrows, and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[12] Lili Cao and John Krumm. From gps traces to a routable road map. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 3–12. ACM, 2009.

[13] Daniel Chen, Leonidas J Guibas, John Hershberger, and Jian Sun. Road network reconstruction for organizing paths. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1309–1320. Society for Industrial and Applied Mathematics, 2010.

[14] G. Chen, B. Chen, and Y. Yu. Mining Frequent Trajectory Patterns from GPS Tracks. In *2010 Intl. Conf. on Comp. Intel. and Soft. Eng.*, 2010.

[15] D. Sacharidis, et al. On-line discovery of hot motion paths. In *Proc. the 11th Intl. Conf. on Extending DB Technology*, pages 392–403, 2008.

[16] Cynthia Sung Dan Feldman and Daniela Rus. The single pixel gps: Learning big data signals from tiny coresets. In *Advances in Geographic Information Systems*.

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

[18] Peter Fenwick. Block sorting text compression - final report, 1996.

[19] Daniel S. Hirschberg and Debra A. Lelewer. Efficient decoding of prefix codes. *Commun. ACM*, 33(4):449–459, April 1990.

[20] Georgios Kellaris, Nikos Pelekis, and Yannis Theodoridis. Trajectory compression under network constraints. In *Advances in Spatial and Temporal Databases*, pages 392–398. Springer, 2009.

[21] Sinn Kim and Jong-Hwan Kim. Adaptive fuzzy-network-based c-measure map-matching algorithm for car navigation system. *Industrial Electronics, IEEE Transactions on*, 48(2):432 –441, apr 2001.

[22] J. G. Lee, J. Han, and K. Y. Whang. Trajectory clustering: a partition-and-group framework. In *Proc. of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 593–604, 2007.

[23] Tom Lyche. The Singular Value Decomposition and Least Squares Problems. 2008.

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

[25] Paul Newson and John Krumm. Hidden Markov map matching through noise and sparseness. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '09, pages 336–343, 2009.

[26] Brian Niehoefer, Ralf Burda, Christian Wietfeld, Franziskus Bauer, and Oliver Lueert. Gps community map generation for enhanced routing methods based on trace-collection by mobile phones. In *Advances in Satellite and Space Communications, 2009. SPACOMM 2009. First International Conference on*, pages 156–161. IEEE, 2009.

[27] W. Y. Ochieng, M. A. Quddus, and R. B. Noland. Map-matching in complex urban road networks. *Brazilian Journal of Cartography*, 55(2):1–18, 2004.

[28] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A.Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, volume 3, 2009.

[29] Falko Schmid, Kai-Florian Richter, and Patrick Laube. Semantic trajectory compression. In *Advances in Spatial and Temporal Databases*, pages 411–416. Springer, 2009.

[30] Cyrill Stachniss, Udo Frese, and Giorgio Grisetti. What is SLAM?, January 2007.

[31] Sebastian Thrun, Wolfram Burgard, Dieter Fox, et al. *Probabilistic robotics*, volume 1. MIT press Cambridge, 2005.

[32] Stephen Tully, George Kantor, Howie Choset, and Felix Werner. A multi-hypothesis topological slam approach for loop closing on edge-ordered graphs. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 4943–4948. IEEE, 2009.

[33] F. Van Diggelen. Innovation: Gps accuracy-lies, damn lies, and statistics. *GPS WORLD*, 9:41–45, 1998.

[34] William Welch. More than 50 crashes on Bay Bridge curve, November 2009.

[35] C. White. Some map matching algorithms for personal navigation assistants. *Transportation Research Part C: Emerging Technologies*, (1-6):91–108, December.

[36] J.J.C. Ying, W.C. Lee, T.C. Weng, and V.S. Tseng. Semantic trajectory mining for location prediction. In *Proc. 19th ACM SIGSPATIAL Intl. Conf. on Advances in Geographic Information Systems*, pages 34–43, 2011.

[37] Lijuan Zhang, Frank Thiemann, and Monika Sester. Integration of gps traces with road map. In *Proceedings of the Second International Workshop on Computational Transportation Science*, pages 17–22. ACM, 2010.