Université Libre de Bruxelles

Line Simplification Algorithm

Author:
Soufiane AJOUAOU
000459811

Supervisor:
Esteban Zimanyi

May 7, 2024

Abstract

BLABLABLA

Keywords— AAA, BBB, DDD, CCCC, TTTT

Acknowledgements

BLABLABLA

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In cartography, the need to simplify complex geographical data for clear and efficient representation is becoming increasingly important. The rise of automated cartography driven by computer technologies has heightened the demand for efficient algorithms, especially in the areas of feature extraction and simplification.

This master thesis aims to contribute to the study of cartographic representation techniques, with a central emphasis on the utilization of POSTGIS and MobilityDB [29], by exploring solutions for real-time processing of trajectories. Indeed dynamic adaptability is crucial in modern applications such as GPS navigation, real-time geospatial data analysis, and the constant monitoring of geospatial data streams, all of which are integral to the POSTGIS and MobilityDB ecosystems.

In this study, our primary aim will be to simplify line representations of mobility data within the environment of POSTGIS and MobilityDB. This involves the processing of an ordered set of n+1 points in a plane, delineating a polygonal path composed of line segments, and deriving a simplified path with fewer segments while preserving the fundamental attributes of the initial path. Our study stands out for its focus on the real-time nature of the data, which aligns well with the principles and capabilities of POSTGIS and MobilityDB.

An underlying assumption for this work is the simplicity of the provided path, with an absence of self-intersections. The presence of self-intersections in cartographic data often indicates issues related to digitization errors. While our goal is to keep the resulting approximation simple, the issue of computational feasibility within the POSTGIS and MobilityDB environment is a key consideration.

The idea of effectively representing data involves several aspects, such as keeping the original and simplified paths close, minimizing the area between them, incorporating critical points from the original path into the simplified one, and optimizing various measures of curve discrepancy. These aspects represent challenges to overcome for an effective real-time trajectory compression algorithm because of the limited information available about the path being processed.

Chapter 2

State of the Art

This section will present a review of the literature on line simplification algorithms, stream processing algorithms and introduce several notions related to the subject of this master thesis.

2.1 Error Metrics

In this section we will explain how to compare a trajectory with its simple counterpart in detail. In this study, a trajectory is represented as a polyline P, which is made up of a series of points $\{p_1, ..., p_n\}$, where the points pi are made up of the longitude, latitude, and sample time-stamp, Xi, Yi, and ti, respectively. Approximation A, which is a subset of P, the polyline of the original trajectory, is the name given to the simplified version of a trajectory [26]. Moreover, the original trajectory's p1 and pn must be included in Approximation A. In the thesis we will focus on a spatial metrics in order to compare the trajectory and its simple version.

2.1.1 Hausdorff Distance

One simple similarity measure for trajectories is the Hausdorff distance. It is a very general similarity measure that can be used for any two point sets. If we have two sets of points P and Q, such as two trajectories if we treat them as polygonal curves and disregard the timestamps, the directed Hausdorff distance from P to Q is equal to the Euclidean distance between the point in P that is furthest from any point in Q, and the point in Q that is closest to that point [15]. Written formally, we get:

$$\overrightarrow{\mathrm{H}}(P, Q) = \max_{p \in P} \min_{q \in Q} \|p - q\|$$

The undirected Hausdorff distance, also just called the Hausdorff distance, is then the maximum of the directed distances in both directions.

$$\mathrm{H}(P, Q) = \max\{\overrightarrow{\mathrm{H}}(P, Q), \overrightarrow{\mathrm{H}}(Q, P)\}$$

2.1.2 Frechet Distance

Another trajectory similarity measure that we use often is the Fréchet distance. It is based on the principle that similar polygonal curves should not just be close in space, but there should also exist some parametrization of the curves such that if we traverse both simultaneously we should remain close at all times [15]. Closeness here is defined as having small Euclidean distance. The trajectories are treated as polygonal curves and the timestamps are not taken into account. For polygonal curves/trajectories P and Q, with P and Q probes respectively, the Fréchet distance is defined as:

$$\mathrm{F}(P,Q) = \inf_{(\sigma,\theta)} \max_{j:[0,1]} \|P(\sigma(j)) - Q(\theta(j))\|$$

where σ and θ are continuous non-decreasing functions from [0, 1] to the real intervals [1, n] and [1,m], respectively. This is often explained with the following analogy: Suppose someone is walking their dog. P is the path the owner takes, and Q is the path of the dog. The owner and dog can change their speed at will, but they cannot go backwards on the path. The Fréchet distance is then the shortest possible length the dog's leash can have for this walk to be possible. The weak Fréchet distance is similar, but the constraint is dropped that σ and θ are non-decreasing. Going back to the man-walking-dog analogy this means the man and dog can freely move backwards and forwards over their path if this results in a shorter leash being needed. The weak Fréchet distance between curves thus gives a lower bound for the (strong) Fréchet distance. The discrete Fréchet distance is a variant where σ and θ are discrete functions from 0,..., k to 1,..., n and 1,...,m with the property that $0 \leq \sigma(i+1) - \sigma(i) \leq 1$ and $0 \leq \theta(i+1) - \theta(i) \leq 1$. This has been explained as someone walking a pet frog, where instead of walking along edges of the polygonal curve, the owner and frog can only hop from vertex to vertex. The discrete Fréchet distance between two curves gives an upper bound on their (continuous) Fréchet distance. As you might expect, computing the exact Fréchet distance between two curves is not completely straightforward. Alt and Godau [3] described an approach for computing this distance. They de σ ne an algorithm for solving a decision variant of the problem of computing the distance. This algorithm can answer if the Fréchet distance between two curves is at most some value δ . Then they use a technique called parametric search to σ nd the minimum value of δ . Their decision algorithm works by constructing what is called a free space diagram for a value for δ . It consists of a grid of $(n-1) \times (m-1)$ cells, where each cell corresponds to a pair of line segments, one from P and one fromQ. Each column of cells corresponds to an edge of P and each row corresponds to an edge of Q. For example, the point (2.6, 3.5) in a free space diagram corresponds to the probe gotten by linearly interpolating between P's second and third probes with a λ of 0.6, and the probe gotten by linearly interpolating between Q's third and fourth probes with a λ of 0.5. Each cell is divided into free space and forbidden space. If the Euclidean distance between the curves at a point is less than or equal to δ , the point is in the free space. If the distance is greater than δ , the point lies in the forbidden space. See Figure 2.1.

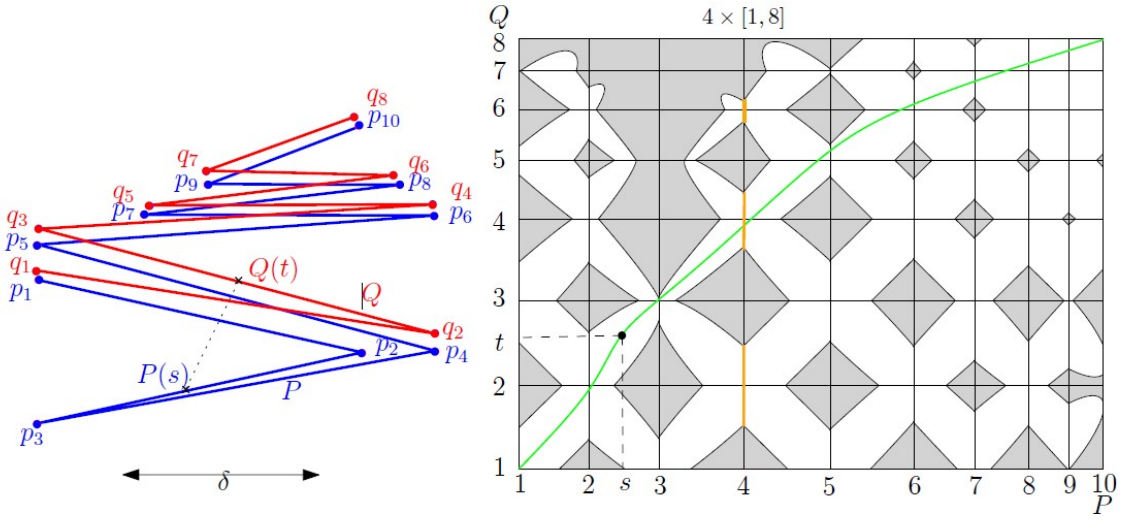

Figure 2.1: Enter Two polygonal curves P (in blue) and Q (in red), and their free space diagram for the chosen value of δ. The free space is shown in white and the forbidden space is shown in gray. Each cell of the FSD corresponds to the combination of one edge of P and one edge of Q. (s, t) is a free point in the diagram, lying on a reachable path in the free space. One green spot is marked in both the FSD and on the associated spots on P and Q. A x- and y- monotone path contained in the free space from (1, 1) to (8, 10), shown in green, corresponds to parametrizations of P and Q realizing a Fréchet distance of at most δ [15].

Now, an x- and y-monotone path from the point (1, 1) to (n,m) entirely through the free space corresponds to parametrizations of P and Q such that the distance between the two is at most δ at any time, i.e. the Fréchet distance is at most δ . For additional details we refer to the paper by Alt and Godau [3].

2.1.3 Dynamic Time Warping Distance

Dynamic Time Warping (DTW) is a techniques to find the optimal alignment between two sequences in same time. The DTW distance take into account the time dimension. This distance can be expressed as the minimum cost of a warping path p between X and Y where X and Y are two sequences (time-dependant). [18]

2.2 Line Simplification Algorithm

Line simplification deals with the simplification of arbitrary lines, which can be straight or curved. The goal is to reduce the complexity of lines while still ensuring that the simplified representation captures the main features of the original lines. This algorithm address the problem of Line simplification this problem who is relevant for GPU computing and Spatial Data processing

2.2.1 Douglas-Peucker Algorithm

The Douglas-Peucker algorithm [7] [11] takes a polyline P a sequence of points $\{p_1, ..., p_n\}$, and a user defined allowed spatial error, $\varepsilon > 0$. The algorithm builds an approximation polyline P' , initially consisting of p_1 and p_n. It continues adding the point pi out of the original polyline that has the largest shortest-euclideandistance to P' until that distance is smaller than ε as demonstrated in Figure 2.2.

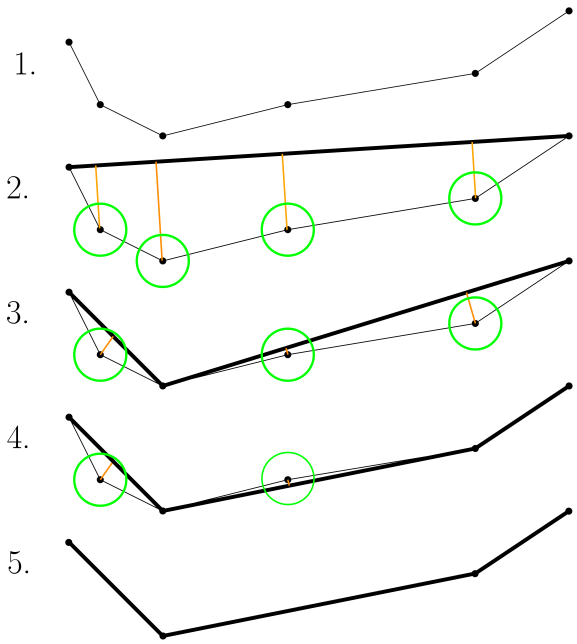

Figure 2.2: Illustration of how the Douglas-Peucker algorithm iteratively simplifies a line. The allowed spatial error ε is depicted with green circles [26].

2.2.2 Visvalingam-Whyatt Algorithm

The Visvalingam-Whyatt algorithm [27] uses the concept of 'effective area', which is the area of the triangle formed by a point and its two neighbors. The algorithm takes a polyline P a sequence of points $\{p_1, ..., p_n\}$, and a user defined allowed spatial displacement error, $\varepsilon > 0$. For every set of three consecutive points $\{p_{i-1}, p_i, p_{i+1}\}$ a triangle is formed with its surface being the 'effective area'. Iteratively point p_i is dropped that results in the least areal displacement to form an approximation as illustrated in Figure 2.3. This process halts when the 'effective area' is larger than ε.

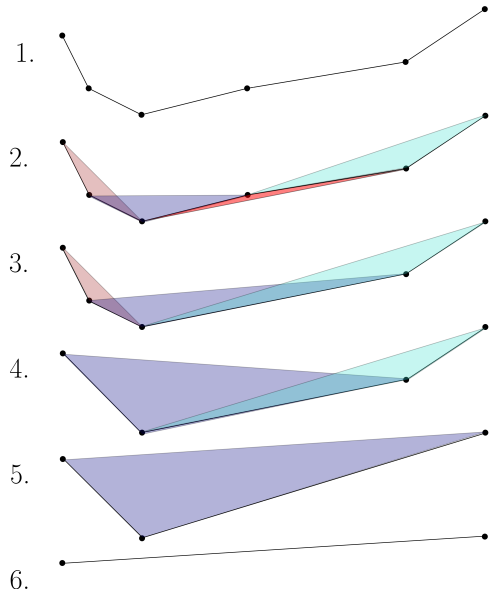

Figure 2.3: Illustration of how the Visvalingam-Whyatt algorithm iteratively simplifies the line [26].

2.2.3 Imai-Iri

The basis of the Imai-Iri algorithm [13] lies in the construction of an unweighted directed acyclic graph G. This graph is constructed by connecting all combinations of two points that would create an allowed shortcut. A breadth-first search is done on this graph to compute the shortest path connecting the first and last point, resulting in the approximation. This algorithm takes a polyline P a sequence of points $\{p_1, ..., p_n\}$, and a user defined allowed spatial error, $\varepsilon > 0$. For each combination of two points (p_i and p_j) it checks if a line between them intersects all circles with radius ε that center on the points that lie between them $\{p_x > p_i, p_x < p_j\}$. When this is the case, the line $p_i p_j$ is an allowed shortcut and is added to the graph G, see Figure 2.4. After all allowed shortcuts are added to graph G, breadth-first search is done to find the shortest path through the graph from p_1 to p_n.

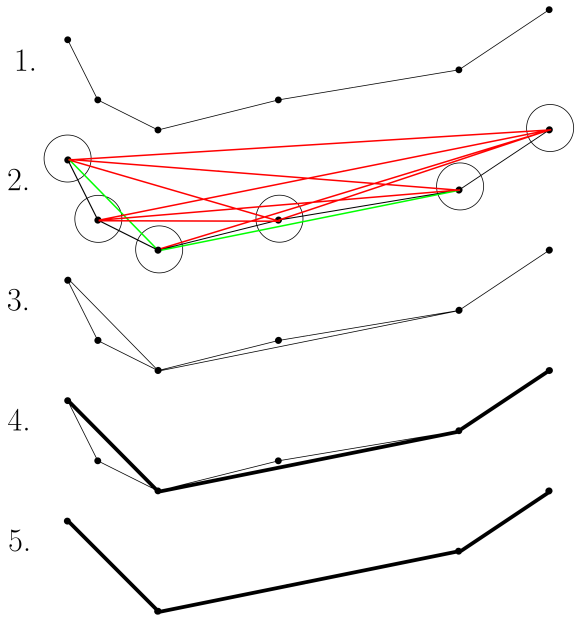

Figure 2.4: Illustration of how the Imai-Iri algorithm generates shortcuts. Green lines are allowed shortcuts, red lines are not allowed [26].

2.3 Streaming Process Algorithm

Traditional data-management systems software is built on the concept of persistent data sets that are stored reliably in stable storage and queried/updated several times throughout their lifetime. For several emerging application domains, however, data arrives and needs to be processed on a continuous (24×7) basis, without the benefit of several passes over a static, persistent data image [8].

It is therefore important to have algorithm to process those stream data.

2.3.1 Data-Stream

Perhaps the most basic synopsis of a data stream is a sample of elements from the stream. A key benefit of such a sample is its flexibility: the sample can serve as input to a wide variety of analytical procedures and can be reduced further to provide many additional data synopses [8]. If, in particular, the sample is collected using random sampling techniques, then the sample can form a basis for statistical inference about the contents of the stream. Data-stream sampling problems require the application of many ideas and techniques from traditional database sampling, but also need significant new innovations, especially to handle queries over infinite-length streams. Indeed, the unbounded nature of streaming data represents a major departure from the traditional setting. We draw an important distinction between a stationary window, whose endpoints are specified times or specified positions in the stream sequence, and a sliding window whose endpoints move forward as time progresses. Examples of the latter type of window include “the most recent n elements in the stream” and “elements that have arrived within the past hour.” Sampling from a finite stream is a special case of sampling from a stationary window in which the window

boundaries correspond to the first and last stream elements. When dealing with a stationary window, many traditional tools and techniques for database sampling can be directly brought to bear. In general, sampling from a sliding window is a much harder problem than sampling from a stationary window: in the former case, elements must be removed from the sample as they expire, and maintaining a sample of adequate size can be difficult.We also consider "generalized" windows in which the stream consists of a sequence of transactions that insert and delete items into the window; a sliding window corresponds to the special case in which items are deleted in the same order that they are inserted.

In the context of this thesis we will investigate the technology of POSTGIS on data stream and adapt the algorithm based on the techniques used by POSTGIS for data stream.

2.3.2 Stream Processing

Data streams can be generated in various scenarios, including a network of sensor nodes, a stock market or a network monitoring system and so on [19]. It exist multiple techniques that could be done on a data stream such as continuous queries,clustering,classification,frequent items mining,outlier and anomaly detection.A stream processing solution has to solve different challenges [24] :

- Processing massive amounts of streaming events (filter, aggregate, rule, automate, predict, act, monitor, alert)
- Real-time responsiveness to changing market conditions
- Performance and scalability as data volumes increase in size and complexity
- Rapid integration with existing infrastructure and data sources: Input (e.g. market data, user inputs, files, history data from a DWH) and output (e.g. trades, email alerts, dashboards, automated reactions)
- Fast time-to-market for application development and deployment due to quickly changing landscape and requirements
- Developer productivity throughout all stages of the application development lifecycle by offering good tool support and agile development
- Analytics: Live data discovery and monitoring, continuous query processing, automated alerts and reactions
- Community (component / connector exchange, education / discussion, training / certification)
- End-user|ad-hoc continuous query access
- Alerting
- Push-based visualization

2.3.3 Technologies

In this section, we discuss some technological solutions for data streams processing. Apache Storm is a distributed real-time computation system for processing large volumes of high-velocity data [14]. Is a distributed real-time computation system for processing fast, large streams of data. Storm is an architecture based on master-workers paradigm. So a Storm cluster mainly consists of a master and worker nodes, with coordination done by Zookeeper.

Spark Streaming [23] is an extension of the core Spark API [22] that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Flume, Twitter, ZeroMQ, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window, see Figure 2.5.

Figure 2.5: Spark Streaming model representation as we can see it can receive multiple data sources and output to multiple data sources [23].

Finally, processed data can be pushed out to files systems, databases, and live dashboards. In fact, you can apply Spark's machine learning and graph processing algorithms on data streams, see Figure 2.6).

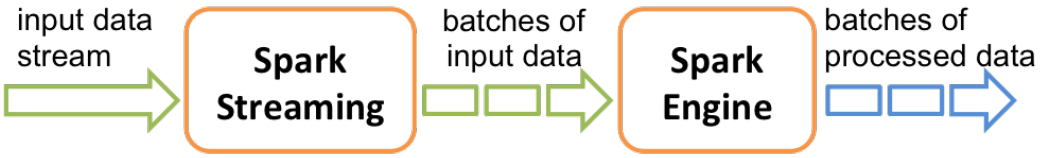

Figure 2.6: Spark Processing [23].

Apache Samza [21] is a distributed stream processing framework. It uses Apache Kafka for messaging, and Apache Hadoop YARN to provide fault tolerance, processor isolation, security, and resource management see Figure 2.7.

Apache Flume [28] is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data. It has a simple and

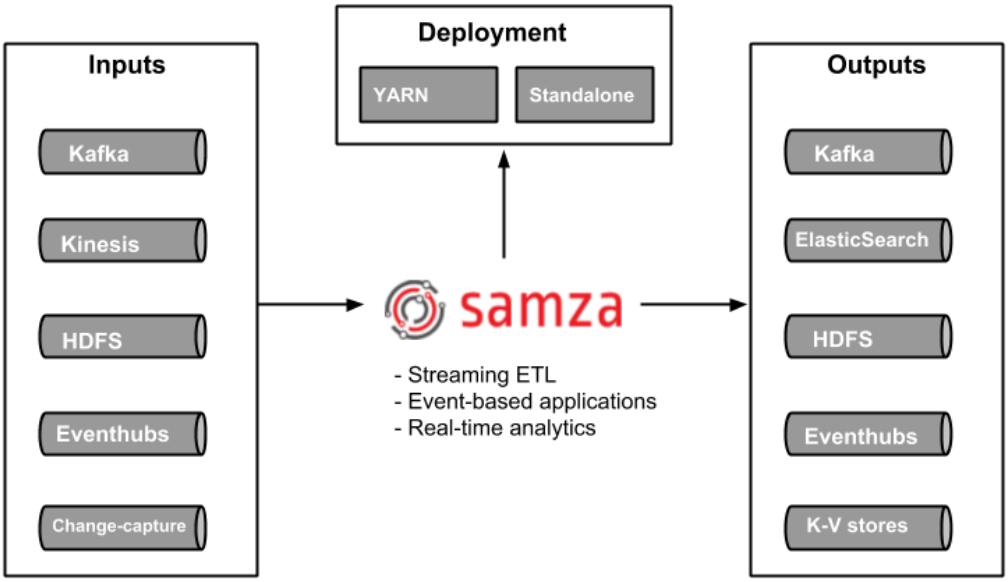

Figure 2.7: Apache Samza [21].

flexible architecture based on streaming data flows. It is robust and fault tolerant with tunable reliability mechanisms and many failovers and recovery mechanisms. It uses a simple extensible data model that supports online analytic applications see Figure 2.8.

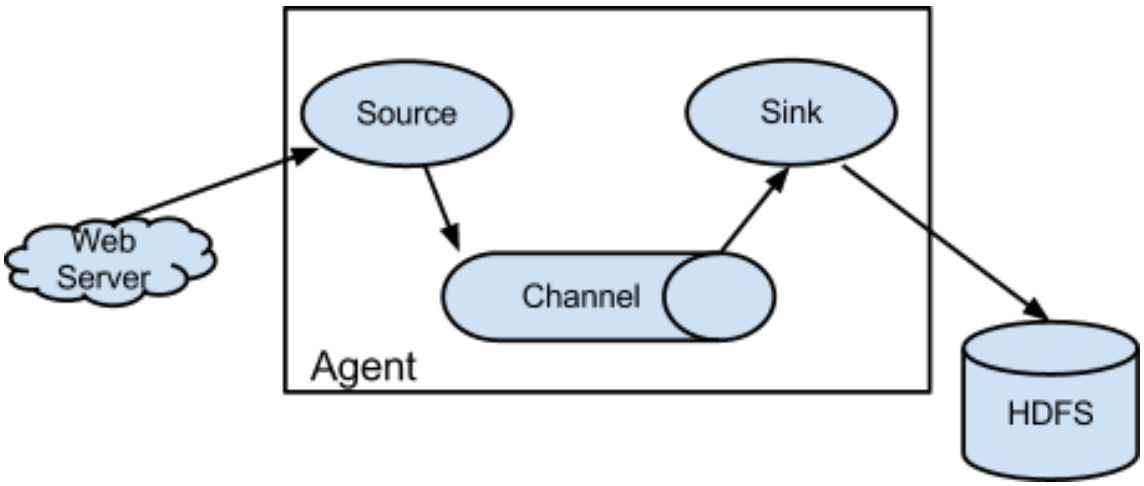

Figure 2.8: Apache Flume [28].

Apache Kafka itself is often used as a kernel for data stream architecture. Originally, Apache Kafka is publish-subscribe messaging rethought as a distributed commit log [4]. Apache Kafka is a distributed system designed for streams. It is built to be fault-tolerant, high-throughput, horizontally scalable, and allows geographically

distributing data streams and processing. See Figure 2.9 illustrates stream-centric architecture in Confluent blog [25].

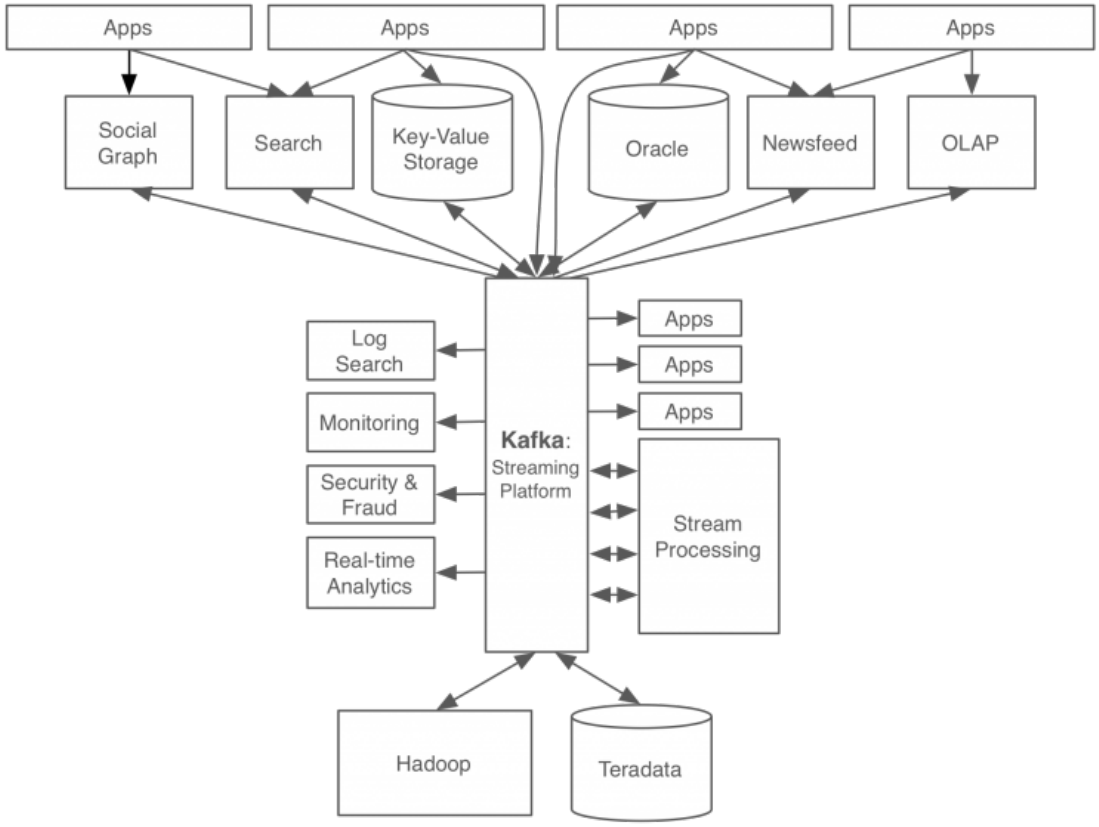

Figure 2.9: Stream-centric architecture on Apache Kafka [25].

Amazon Kinesis [6] is a fully managed, cloud-based service for real-time data processing over large, distributed data streams. Amazon Kinesis can continuously capture and store terabytes of data per hour from hundreds of thousands of sources such as website clickstreams, financial transactions, social media feeds, IT logs, and locationtracking events.

IBM InfoSphere Streams [5, 12] is an advanced analytic platform that enables the development and execution of applications that process information in data streams. InfoSphere Streams enables continuous and fast analysis of massive volumes of moving data to help improve the speed of business insight and decision making see Figure 2.10.

2.3.4 Streaming for Line Simplification

In this section we will discuss the current state of line simplification algorithm/problem in the context of stream processing. The main problematic of this thesis is related to this problem because of that this subject will have a more in depth analysis.

As mentionned in "Streaming algorithms for line simplification" [1], suppose we are tracking one, or maybe many, moving objects. Each object is equipped with

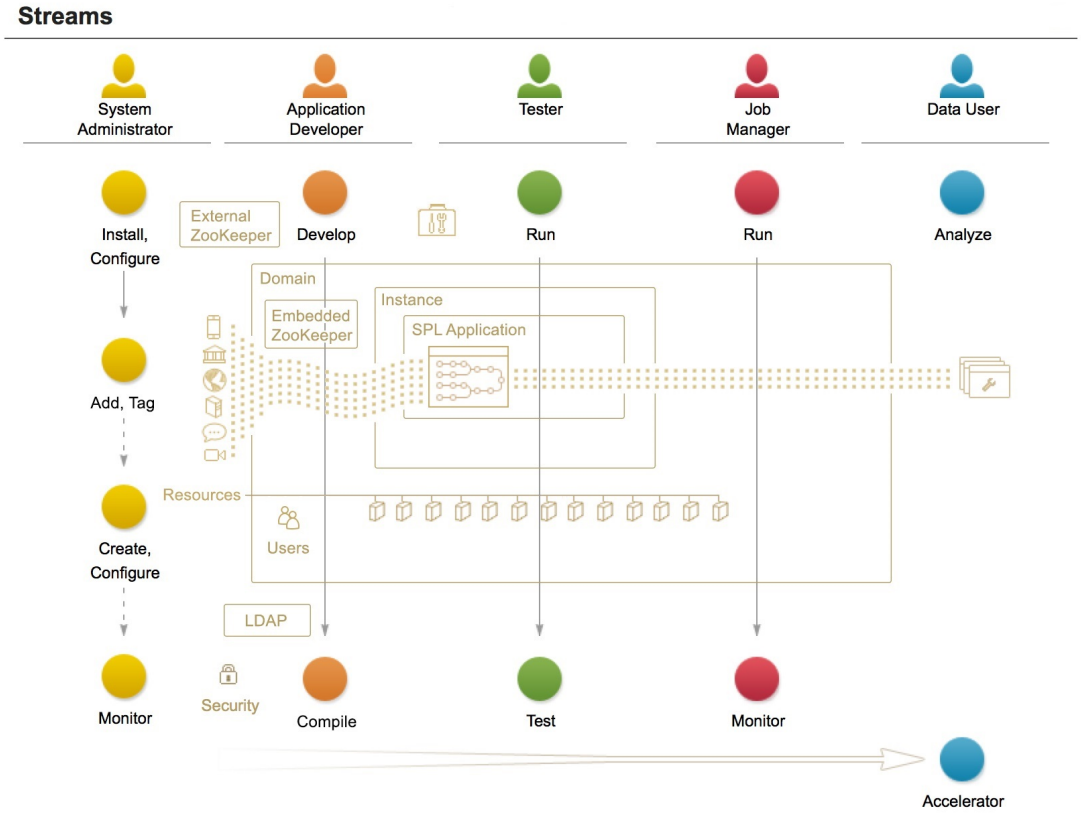

Figure 2.10: IBM InfoSphere Streams [12].

a device that is continuously transmitting its position. Thus we are re- ceiving a stream of data points that describes the path along which the object moves. The goal is to maintain this path for each object. We are interested in the scenario where we are tracking the objects over a very long period of time, as happens for instance when studying the migratory patterns of animals. In this situation it may be undesirable or even impossible to store the complete stream of data points. Instead we have to maintain an approximation of the input path. This leads us to the following problem: we are receiving a (possibly infinite) stream $p0, p1, p2, ...$ of points in the plane, and we wish to maintain a simplification (of the part of the path seen so far) that is as close to the original path as possible, while using not more than a given (fixed) amount of available storage.

Formalization

In this subsection we will formalize the problem of line simplification in a streaming mode based on the article [1]. To be able to state the problem we wish to solve and the results we obtain more precisely, we first introduce some terminology and

definitions. Let $p_0, p_1, \ldots$ be the given stream of input points. We use $P(n)$ to denote the path defined by the points $p_0, p_1, \ldots, p_n$ - that is, the path connecting those points in order - and for any two points p, q on the path we use $P(p, q)$ to denote the subpath from p to q. For two vertices p_i, p_j we use $P(i, j)$ as a shorthand for $P(p_i, p_j)$. A segment $p_i p_j$ with $i < j$ is called a link or sometimes a shortcut. Thus $P(n)$ consists of the links $p_{i-1}p_i$ for $0 < i \leqslant n$. We assume a function error is given that assigns a non-negative error to each link $p_i p_j$. A ℓ-simplification of $P(n)$ is a polygonal path $Q := q_0, q_1, \ldots, q_k, q_{k+1}$ where $k \leqslant \ell$ and $q_0 = p_0$ and $q_{k+1} = p_n$, and $q_1, \ldots, q_k$ is a subsequence of $p_1, \ldots, p_{n-1}$. The error of a simplification Q for a given function error, denoted error(Q), is defined as the maximum error of any of its links.

Evaluation

Now consider an algorithm $\mathcal{A} := \mathcal{A}(\ell)$ that maintains an ℓ-simplification for the input stream $p_0, p_1, \ldots$, for some given ℓ. Let $Q_{\mathcal{A}}(n)$ denote the simplification that $\mathcal{A}$ produces for the path $P(n)$. Let Opt(ℓ) denote an optimal off-line algorithm that produces an ℓ-simplification. Thus error $\big(Q_{Opt(\ell)}(n)\big)$ is the minimum possible error of any $\ell-$ simplification of $P(n)$. We define the quality of $\mathcal{A}$ using the competitive ratio, as is standard for on-line algorithms. We also allow resource augmentation. More precisely, we allow $\mathcal{A}$ to use a $2k$-simplification, but we compare the error of this simplification to $Q_{Opt(k)}(n)$. (This is similar to Agarwal et al. [2] who compare the quality of their solution to the min- k problem for a given maximum error δ to the optimal value for maximum error $\delta/2$.) Thus we define the competitive ratio of an algorithm $\mathcal{A}(2k)$ as

$$\text{competitive ratio of } \mathcal{A}(2k) := \max_{n \geqslant 0} \frac{\text{error}\left(Q_{\mathcal{A}(2k)}(n)\right)}{\text{error}\left(Q_{Opt(k)}(n)\right)},$$

where $\frac{\text{error}\left(Q_{\mathcal{A}(2k)}(n)\right)}{\text{error}\left(Q_{Opt(k)}(n)\right)}$ is defined as 1 if error $\left(Q_{\mathcal{A}(2k)}(n)\right) =$ error $\left(Q_{Opt(k)}(n)\right) = 0$. We say that an algorithm is c competitive if its competitive ratio is at most c.

Algorithm

We will discuss the existing algorithm for the line simplification in a streaming model. In the article [1] they propose the following algorithm. Our algorithm is quite simple. Suppose we have already handled the points $p_0, \ldots, p_n$. (We assume $n > \ell + 1$; until that moment we can simply use all points and have zero error.) Let $Q := q_0, q_1, \ldots, q_\ell, q_{\ell+1}$ be the current simplification. Our algorithm will maintain a priority queue $\mathcal{Q}$ that stores the points q_i with $1 \leqslant i \leqslant \ell$, where the priority of a point is the error (as computed by the oracle) of the link $q_{i-1}q_{i+1}$. In other words, the priority of q_i is (an approximation of) the error that is incurred when q_i is removed from the simplification. Now the next point p_{n+1} is handled as follows:

1. Set $q_{\ell+2} := p_{n+1}$, thus obtaining an $(\ell + 1)$-simplification of $P(n + 1)$.
2. Compute error* $(q_\ell q_{\ell+2})$ and insert $q_{\ell+1}$ into $\mathcal{Q}$ with this error as priority.

3. Extract the point q_s with minimum priority from $\mathcal{Q}$; remove q_s from the simplification.

4. Update the priorities of q_{s-1} and q_{s+1} in $\mathcal{Q}$.

As we can see the following algorithm is using the error function in order to correct the result in real-time. In the context of this thesis we will investigate a solution that is based on the existing algorithm douglas-peucker to see if it is possible to maintain a simplified version of the trip using this algorithm.

2.3.5 Moving Object Database

In order to store the position of our tracking objects it is important to search on moving object database as data output for this thesis. Moving objects are objects that change their value or location with time. These can be vehicles, persons, animals, aircarft, the air temperature of a city, the fuel price in a certain gas station, etc. The ubiquity of tracking devices and IoT technologies has resulted in collecting massive amounts of data that describe the temporal evolution of such objects and values. This creates opportunities to build applications on this data, which in turn calls for building [29].

As mentioned above the need for a specific database in order to have an efficient data storage. In the context of this thesis we will focus on mobilityDB which is a Moving Object database that extends PostgreSQL

PostgreSQL

PostgreSQL is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2, developed at the University of California at Berkeley Computer Science Department [20]. POSTGRES pioneered many concepts that only became available in some commercial database systems much later.

PostgreSQL is an open-source descendant of this original Berkeley code. It supports a large part of the SQL standard and offers many modern features:

- complex queries
- foreign keys
- triggers
- updatable views
- transactional integrity
- multiversion concurrency control

Also, PostgreSQL can be extended by the user in many ways, for example by adding new

- data types

- functions
- operators
- aggregate functions
- index methods
- procedural languages

And because of the liberal license, PostgreSQL can be used, modified, and distributed by anyone free of charge for any purpose, be it private, commercial, or academic.

PostGIS

PostGIS represents a potent open-source instrument facilitating the creation of resilient spatial databases. Serving as the geographic extension of the PostgreSQL database management system, PostGIS enables the incorporation of geographic objects within data tables [16]. These geographic objects are specialized data types designed for the storage of geographic positions or sets thereof, integrated seamlessly into lines or polygons. In essence, PostGIS emerges as a formidable tool, empowering users to manage intricate geographical data proficiently and to visually interrogate such data when employed in conjunction with graphical tools, exemplified by QGIS.

MobilityDB

MobilityDB uses the abstract data type approach of MOD implementations [10]. In an extensible relational database system this amounts to adding user-defined types that can be used as attribute types inside relations. MobilityDB defines in PostgreSQL and PostGIS the temporal types: `tgeompoint`, `tgeogpoint`, `tfloat`, `tint`, `ttext`, and `tbool`. These types encode functions from the time domain to their corresponding base type domains. MobilityDB extends the existential spatial database by adding a new dimension. It becomes more dynamics and respond to the need of processing moving object data.

Chapter 3

Design

3.1 SQUISH-E

In this section we introduce the SQUISH-E algorithm, present a pseudo-code and analyze its complexity.

The SQUISH-E algorithm compresses a trajectory T by utilizing two parameters, λ and μ, to strategically minimize the Synchronized Euclidean Distance (SED) error while achieving a specified compression ratio (λ). It operates by compressing T until further compression would result in an increase in SED error above μ, with a notable case being SQUISH-E(λ) where setting μ to 0 focuses on minimizing SED error to achieve the compression ratio of λ. In contrast, SQUISH-E(μ) highlights a scenario where λ is set to 1, aiming to maximize the compression ratio without exceeding the SED error threshold defined by μ. A pivotal aspect of SQUISH-E is its use of a priority queue Q, in which the priority of each point is determined by an upper bound on the SED error that could be introduced by its removal. This mechanism allows SQUISH-E to efficiently identify and remove the point with the lowest priority, i.e., the point whose removal would result in the least increase in SED error, in $O(\log |Q|)$ time. This process effectively controls the growth of SED error, ensuring the algorithm's efficiency in compressing trajectory data while maintaining the integrity of the spatial information [17]. In order to have a fully online algorithm SQUISH-E(μ) is not taken into account and we only take into account SQUISH-E(λ). The error function is the same but in our implementation it could be a nice idea to compare different error function to find the best ones for some situation.

3.1.1 Algorithm Pseudo-Code

In this section we will present the pseudo-code of SQUISH-E algorithm used in our work.

Algorithm 1: SQUISH-E(T, λ)

Input: trajectory T, lower bound λ on compression ratio as a percentage value
Output: trajectory T'

```
β ← 4                                          // the initial capacity of Q is 4
foreach point P_i ∈ T do
    if i * lambda ≥ β then
        β ← β + 1                              // increase the capacity of Q
    set_priority(P_i, ∞, Q)                    // enqueue P_i with the priority of P_i being ∞
    π[P_i] ← 0
    if i > 1 then
        succ[P_{i-1}] ← P_i                    // register P_i as P_{i-1}'s closest successor
        pred[P_i] ← P_{i-1}                    // register P_{i-1} as P_i's closest predecessor
        adjust_priority(P_{i-1}, Q, pred, succ, π)        // Algorithm 3
    if |Q| = β then
        reduce(Q, pred, succ, π)                          // Algorithm 2
return trajectory T' comprising the points in Q in the order reflected in the succ map
```

In the algorithm 1 we can analyze the time complexity. The algorithm works iteratively, receiving each point one by one, which is convenient for our streaming case. Complexity is therefore at least $O(n*max(O(loop)))$ .The lambda variable has also been modified. Next, the following adjust priority 3 and reduce 2 algorithms are presented.

Algorithm 2: reduce(Q, pred, succ, π)

Input: priority queue Q, maps pred, succ and π (refer to Table 2)

```
P_j ← remove_min(Q);    // the lowest priority point is removed from Q
π[succ[P_j]] ← max(priority(P_j), π[succ[P_j]]);    // update neighbor priority
π[pred[P_j]] ← max(priority(P_j), π[pred[P_j]]);    // update neighbor priority
succ[pred[P_j]] ← succ[P_j];    // update neighbor's neighbor of P_j
pred[succ[P_j]] ← pred[P_j];    // update neighbor's neighbor of P_j
adjust_priority(pred[P_j], pred, succ, π);    // Algorithm 3
adjust_priority(succ[P_j], pred, succ, π);    // Algorithm 3
remove the entry for P_j from pred, succ, and π;    // garbage collection
```

Algorithm 3: adjust_priority(P_j, Q, pred, succ, π)

Input: point P_j, priority queue Q, maps pred, succ and π (refer to Table 2)

```
if pred[P_j] ≠ null and succ[P_j] ≠ null then
    p ← π[SED(P_j, pred[P_j], succ[P_j])];
    set_priority(P_j, p, Q);
```

To achieve the optimum complexity of this algorithm, it is imperative that the most expensive operation in the loop is $O(log(n))$. Next, the focus will be on variables and their methods, in order to determine the required implementation

elements.

3.1.2 Variables

Variable	Description
Q	Priority queue
p_π	Point with the lowest priority
pred	Predecessor map
succ	Successor map
π	Priority map
P_i	A point in the priority queue
SED	Some distance function

Table 3.1: List of Variables Used in Algorithms

Map

As mentioned before we have 3 map two that has as pair key-value point-point and another with point-priority .

Method The method that is necessary to implement for this map is the following

- set(key,value,map)
- remove(key,map)
- get(key,map) -> value

PriorityQueue

Another structure is a priority queue, but with quite different methods. The priority queue stores elements that are point-priority pairs, and has a sorted structure so that the minimum can be removed. This structure poses a real challenge, due to its nature and the methods we'll define below.

Method The methods required to implement priorityqueue are as follows:

- set_priority(key,value,Q) this function can be expressed as the two following methods
  - remove(key,Q) or replace(key,Q) // depends on implementation
  - push(key,value,Q)
- remove_min(Q)

As the set function is closer to the characteristics of an array than a priorityqueue, this poses a challenge if existing resources don't allow these methods to be implemented in $O(log(n))$ at most.

Chapter 4

Implementation

This chapter focuses on the implementation of the elements described in the previous chapter.

4.1 SQUISH-E Implementation

4.1.1 Algorithm 1

```
1
2  void
3  iteration_simplification_sqe(void *p_i , void *p_j ,
4  size_t *beta,const double lambda,int i,
5  Dict *succ,Dict *pred,
6  PDict  *p,struct PriorityQueue *Q,
7  bool syncdist,interpType interp ,bool hasz ,uint32_t minpts)
8  {
9  if( i * lambda >= *beta)
10 {
11 *beta += 1;
12 }
13 set_priority_queue(p_i,INF,Q);
14 set_priority_dict(p_i,0,p);
15 if(i >= 1)
16 {
17 set_point_dict(p_i,p_j,pred);
18 set_point_dict(p_j,p_i,succ);
19 adjust_priority(p_j,Q,pred,succ,p, syncdist, interp , hasz );
20 }
21 size_t size = size_queue(Q);
22 if(size - *beta == 0 ){
23 reduce(Q,pred,succ,p,syncdist, interp , hasz );
24 }
25 }
```

Listing 4.1: SQUISH-E

This algorithm follow the pseudo code of SQUISH-E it called the set function of the

priority queue and the set function of the point map and priority map that will be describe in the section about implementation of variable. Then the reduce function and adjust priority is called in order to reduce if the size meet the treshold and the adjustment for each point that has a predecessor and a successor in order to know the error that it will produce by removing the corresponding point.

4.1.2 Algorithm 2

```
1
2  void
3  reduce(struct PriorityQueue *Q,Dict *pred,Dict *succ,PDict  *p,
4  bool syncdist,interpType interp ,bool hasz )
5  {
6  struct PriorityQueueElem *entry = remove_min(Q);
7  size_t size_before = size_queue(Q);
8
9  void * p_j = entry->point;
10 double priority = entry->priority;
11
12 void * p_i = get_point_dict(p_j,pred);
13 void * p_k = get_point_dict(p_j,succ);
14
15 double pr_i = get_priority_dict(p_i,p); if(priority > pr_i){ pr_i =
   ↪ priority; }
16 double pr_k = get_priority_dict(p_k,p); if(priority > pr_k){ pr_k =
   ↪ priority; }
17
18 set_priority_dict(p_k, pr_k ,p);
19 set_priority_dict(p_i, pr_i ,p);
20
21 set_point_dict(p_i,p_k,succ);
22 set_point_dict(p_k,p_i,pred);
23 set_point_dict(p_k,p_i,pred);
24
25 adjust_priority(p_k ,Q,pred,succ,p,syncdist, interp ,hasz );
26 adjust_priority(p_i ,Q,pred,succ,p,syncdist, interp ,hasz );
27
28
29 //Delete pointer
30 free(entry);
31 destroy_elem_PriorityDict(p_j,p);
32 destroy_elem_PointDict(p_j,succ);
33 destroy_elem_PointDict(p_j,pred);
34
35 }
```

Listing 4.2: reduce

This algorithm describe the reduction method as mentioned above remove the lowest priority points from the queue and it update the neighbors. Because we use C langage we free memory of the useless data.

4.1.3 Algorithm 3

```
1
2  void
3  adjust_priority(void *p_i,struct PriorityQueue *Q, Dict *pred,Dict *
       ↪ succ,PDict  *p,
4  bool syncdist,interpType interp ,bool hasz )
5  {
6
7  void * p_h = get_point_dict(p_i,pred);
8  void * p_k = get_point_dict(p_i,succ);
9  if( p_h != NULL &&  p_k != NULL )
10 {
11 if(syncdist)
12 {
13 double priority = get_priority_dict(p_i,p) + SED(p_h,p_i,p_k, interp ,
       ↪ hasz );
14 set_priority_queue(p_i,priority,Q);
15 }
16 }
17 }
```

Listing 4.3: adjust_priority

4.2 Variables Implementation

In this section we'll discuss how the variables have been implemented and how effective they are, followed by a discussion of the performance and complexity achieved.

4.2.1 Map

In this section we will focus on the implementation of the map

```
1
2  #include <search.h>
3
4
5  typedef struct PriorityDict
6  {
7  void * key;
8  double priority;
9  } PriorityDict;
10
11
12 typedef struct PointDict
13 {
14 void *key;
15 void *value;
16 } PointDict;
17
18
19 typedef void* Dict;
```

```
20 typedef void* PDict;
```

Listing 4.4: Map C implementation

This structure is a Map that link a point to a corresponding value like priority or another point. We just need to define a struct with two values the key and the value. The typedef void* is using for the GNU Library to avoid using void** and add more clarity to the code.

Method

```
1
2  void *
3  get_point_dict(void *p_i,Dict *dict)
4  {
5  PointDict find;
6  find.key = p_i;
7  void * result = tfind(&find, dict, compar);
8  if(result){
9  result = (*(PointDict**)result)->value;
10 }
11 return result;
12 }
13
14
15 void
16 set_point_dict(void * p_i,void * p_j,Dict *dict)
17 {
18 PointDict *find = malloc(sizeof(PointDict));
19 find->key = p_i;
20 find->value = p_j;
21 void * result = tfind(find, dict, compar);
22 if(result){
23 (*(PointDict**)result)->value = p_j;
24 }
25 else{
26 tsearch(find, dict, compar); /* insert */
27 }
28 }
29
30
31 void destroy_elem_PointDict(void * p_i,Dict *dict)
32 {
33 PointDict find;
34 find.key = p_i;
35 void * r = tfind(&find, dict, compar);
36 if(r)
37 {
38 PointDict * to_free = (*(PointDict**)r);
39 tdelete(&find,dict,compar);
40 free(to_free);
41 }
```

```
42 }
```

Listing 4.5: Point Map Methods

Here we use function of GNU C Library that implement a binary tree. It has the property to have for each operation a maximal complexity of $O(log(n))$ The two mapping use the same implementation in reality it would be a good idea to resolve this duplication case using a template or thing that can resolve it but the C language limit the capability.

Discussion

As mentioned in the previous sub section the map use GNU C Library binary tree structure to perform search and set. Those algorithm have a complexity of $O(log(n))$ as mentioned before and it has the property to be have a dynamic allocation of memory.

4.2.2 PriorityQueue

```
1
2  typedef struct PriorityQueueElem
3  {
4  void * point;
5  double priority;
6  int index; //reference his own index
7  } PriorityQueueElem;
8
9  typedef void* IDict;
10
11 typedef struct PriorityQueue
12 {
13 PriorityQueueElem **arr;
14 IDict dict;
15 size_t size;
16 size_t capacity;
17 } PriorityQueue;
```

Listing 4.6: PriorityQueue

This structure implements the function of a priority queue using the structure of a min heap. It has as value PriorityQueueElem which have the point and the corresponding priority and a reference to this index in the heap. The main structure is using an array of pointer of PriorityQueueElem as long as an index map, a int that represent the number of points in the queue and lastly the numbers points allocated.

Method

```
1
2  PriorityQueueElem *remove_min(PriorityQueue *Q)
3  {
4  PriorityQueueElem *result = NULL;
5  if (Q->size != 0) {
6  result = Q->arr[0];
7  // Replace the deleted node with the last node
8  Q->arr[0] = Q->arr[Q->size - 1];
9  Q->arr[Q->size - 1] = NULL;
10 if(Q->arr[0]){
11 Q->arr[0]->index = 0;
12 }
13
14 // Decrement the size of heap
15 Q->size--;
16
17 tdelete(result,&Q->dict,compar_index);
18 // Call minheapify_top_down for 0th index
19 // to maintain the heap property
20 minHeapify(Q, 0);
21
22 }
23 return result;
24 }
```

Listing 4.7: PriorityQueue Remove Min

```
1
2  void
3  set_priority_queue(void *p_i,double priority ,PriorityQueue *Q)
4  {
5  struct PriorityQueueElem * insert = replace_elem(p_i,priority,Q);
6  if(insert == NULL){
7      insert = malloc(sizeof(struct PriorityQueueElem));
8      insert->point = p_i;
9      insert->priority = priority;
10     insert->index = -1;
11     tsearch(insert, &Q->dict, compar_index); /* insert */
12 }
13 if(insert->index == -1){
14     push(insert,Q);
15 }
16 }
17
18
19 struct PriorityQueueElem *
20 replace_elem(void *p_i,double priority ,PriorityQueue *Q)
21 {
22 PriorityQueueElem *elem = get_elem(p_i,&Q->dict);
23 if(elem ){
24 if(Q->size > elem->index && elem->index != -1){
25 Q->arr[elem->index]->priority = priority;
26 insertHelper(Q, elem->index);
27 minHeapify(Q, elem->index);
28 }
```

```
29  }
30  return elem;
31  }
32
33  void
34  push(PriorityQueueElem * insert,PriorityQueue *Q)
35  {
36  Q->size++;
37  if(Q->size > Q->capacity){
38  Q->arr = realloc(Q->arr, Q->size * sizeof(PriorityQueueElem*));
39  }
40  Q->arr[Q->size-1] = insert;
41  Q->arr[Q->size-1]->index = Q->size-1;
42  insertHelper(Q, Q->size-1);
43  }
```

Listing 4.8: PriorityQueue Set Elem

From other implementation this one has the property to have at most a complexity of $O(log(n))$ the choice of a min heap to implement the priorityqueue fullfill the complexity limitation that we see in the design phase. The min heap is structure has a structure of a tree like we see in Figure 4.1 and has the property that each tree and sub tree has the property that the parent nodes is always lesser than the child nodes. The min heap implementation is inspired from this website [9].

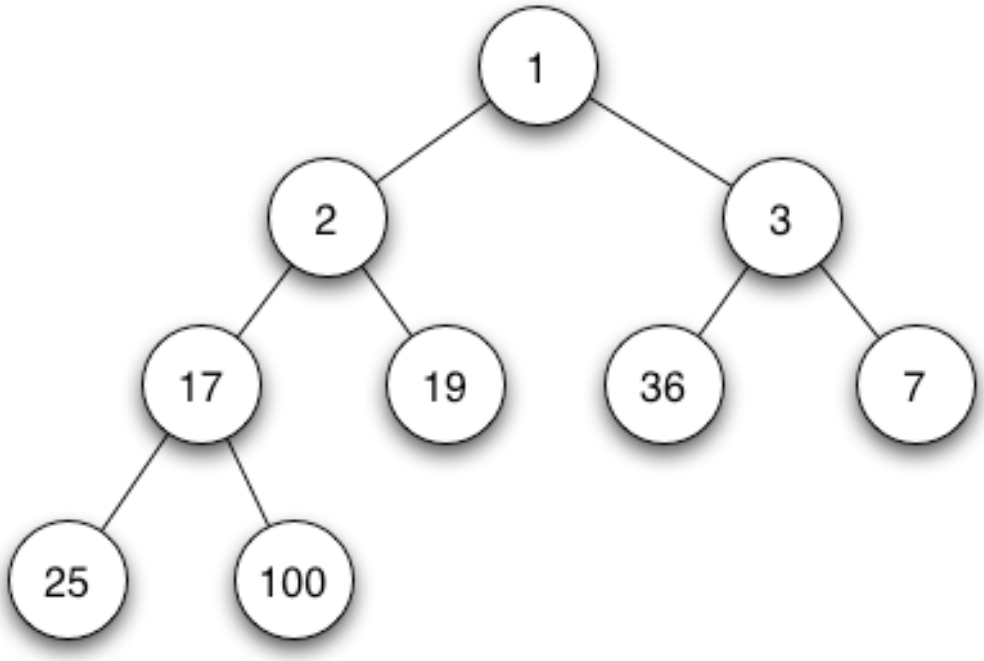

Figure 4.1: Min Heap Structure

Discussion

The idea here is to use the min heap structure to obtain in a complexity of maximum $O(log(n))$ the point with the lowest priority. In the side we use a index map in order to replace point in the heap in an efficient way. Due to the fact that the design include the set of an element when it is already in the array. This structure has the advantage to benefit from static memory. The set works using the mapping to see if the point is already present in the map. If it is we replace it in the heap by changing his priority and apply a check with his ancestor then his descendant. If the point is not present we add a value at the end of the heap and check and update their ancestor to satisfy the condition of a min heap. So at the end the two operations needed for the algorithm reach a complexity of $O(log(n))$

4.3 Postgres Implementation

In this section we describe how the algorithm is implemented into the mobilityDB sql functions.

4.3.1 SQL Code

```
CREATE FUNCTION SquishESimplify(tfloat, float, boolean DEFAULT
    ↪ TRUE)
RETURNS tfloat
AS 'MODULE_PATHNAME', 'Temporal_simplify_sqe'
LANGUAGE C IMMUTABLE STRICT PARALLEL SAFE;
CREATE FUNCTION SquishESimplify(tgeompoint, float, boolean
    ↪ DEFAULT TRUE)
RETURNS tgeompoint
AS 'MODULE_PATHNAME', 'Temporal_simplify_sqe'
LANGUAGE C IMMUTABLE STRICT PARALLEL SAFE;
```

Listing 4.9: SQUISHE SQL Code

4.3.2 Example

This section show some examples of usages of the pgsql function describe in 4.9. This allowed to use the algorithm in an offline settings using request.

```

SELECT SquishESimplify(tfloat '[1@2000-01-01, 2@2000-01-02, 3
    ↪ @2000-01-04,
4@2000-01-05]', '1 day');
-- [1@2000-01-01, 3@2000-01-04]

SELECT asText(SquishESimplify(tgeompoint '[Point(1 1 1)@2000
    ↪ -01-01,
```

```
Point(2 2 2)@2000-01-02, Point(3 3 3)@2000-01-04, Point(5 5 5)
    ↪ @2000-01-05)', 0.5));
-- [POINT Z (1 1 1)@2000-01-01, POINT Z (3 3 3)@2000-01-04,
    ↪ POINT Z (5 5 5)@2000-01-05)
```

Listing 4.10: Example SQL Code

4.3.3 Result

Here we will talk about the result of the function and see the precision and the performance of the requests. We will also compare it with the C implementation in real time using the same data to compare the offline implementation with the online one.

4.3.4 Performance

This section will analyze and discuss the performance of the sql function by varying the lambda parameter and the number of points.

Number of Points	Lambda				
	1	0.75	0.5	0.25	0.01
100	00.001137	00.001965	00.002775	00.003328	00.004529
1000	00.01498	00.014966	00.022183	00.024588	00.023035
10000	00.164196	00.215692	00.214011	00.247379	00.218447
100000	01.77365	02.171435	02.844324	02.617976	02.437037
1000000	04.618455	05.371317	06.461961	06.021071	04.510934

Table 4.1: Average Execution Time by Number of Points and Lambda

In order to get those results we benchmark the request 10 times for each requests and retrieve the average of those executions. As standalone results it has no real meaning outside but we can notice that the speed of the algorithm increase around 0.5 and decrease outside those values of lambda. We can also state that maybe there is another value as maximum execution time between 0.75 and 0.25 . We can state that this is maybe because the number of instruction that mix setting in a priority queue and the instructions of reduction is around 0.5. Because when lambda is equal to 1 there is no reduction and only setting in the maximal size of a priority queue and adjust priority operation and when lambda go towards 0 there is a lot of reduction operation and setting in the minimal size of a priority queue. That is an explanation that could befit those data. When the size of the input is multiplying by 10 the execution time is around 2 times longer.

Comparison with C

In order to have meaning to those result we will compare it with the C implementation using the real time approach and to see between the offline and the online approach the difference that happen.

Number of Points	Lambda				
	1	0.75	0.5	0.25	0.01
100	00.000094	00.000092	00.00009	00.000098	00.000096
1000	00.00081	00.000822	00.000943	00.000876	00.000788
10000	00.008889	00.008852	00.008663	00.008263	00.00818
100000	00.090018	00.088485	00.089533	00.089064	00.088728
1000000	00.892172	00.864116	00.870661	00.937186	00.920517

Table 4.2: Average Execution Time (C) by Number of Points and Lambda

As we can see, executing C code is much faster than executing sql queries, which makes execution and simplification using steam processing techniques in C both viable and possible. The speed is such that even for 1 million points the average execution time is less than 1 second.

4.3.5 Precision

This section will focus on the precision of the simplification using the same parameter as before in order to have an idea of the quality of the simplification in a function of lambda and types of path. We will discuss on trajectory based on AIS dataset. As stated in [1], there is different type of path as convex and concave. As a reminder this work focus on the simplification of lines and do not take into account land or see restriction during the simplification process. In this section we will analyze different trajectory defines them and see with different lambda and metrics evaluation the quality of the simplification. At the end we will run in a big datasets and see the results as a graph and discuss them.

Trajectory 1

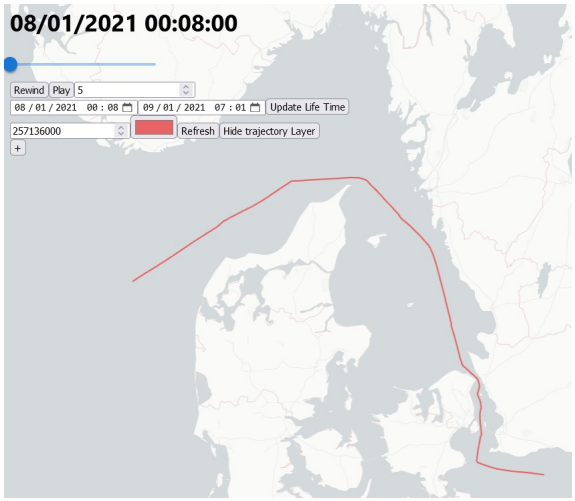

Figure 4.2: Trajectory 1

This trajectory is composed of 20865 points and begin at 1am the 8/1/2021 with a duration of 1 day. This trajectory have convex and concave path in order to analyze the effect of the simplification in those specific moment. The precision of the trajectory will be computed based on the current metrics proposed in the state of the art such as frechet distance and hausdorff distances. We will not discuss the precision of lambda 1 because it does not reduce any points from the original trajectory.

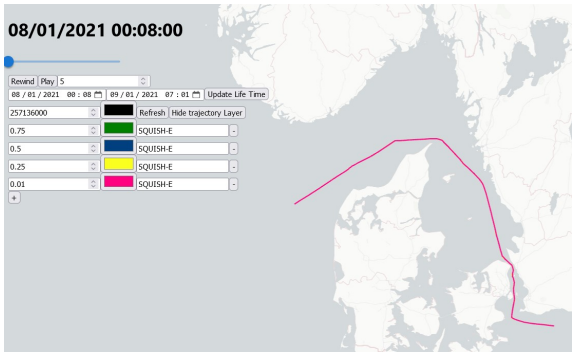

Figure 4.3: Trajectory 1 - SQUISH-E

The path looks really similar and is close to the original path in 4.3. In order to see the differences we can zoom and move the slider in order to see the moving objects in relation with the current time.

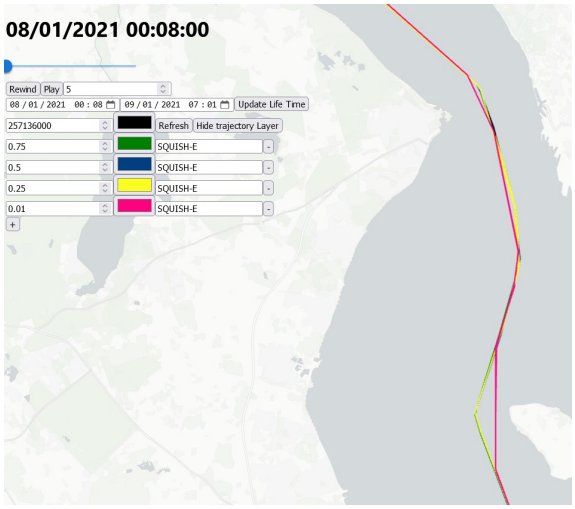

Figure 4.4: Trajectory 1 - ZOOM

In this we can see the path being simplified where there is curves and having less precision as lambda decreases.

This table gives a view of the precision of squish-e on the first trajectory. The data shows that it is consistent and precise and that errors is increasing when lambda decreases. Dynamic Time Warping Distance gives also an overview of the loss of precision when we remove more points.

	Lambda			
	0.75	0.5	0.25	0.01
Number of points	15648	10432	5215	208
Frechet Distance	818.006	818.006	818.006	6902.445
Hausdorff Distance	818.006	818.006	818.006	6902.445
DynTimeWarp Distance	80452.262	236389.238	634626.475	22557588.744
Temporal Distance	0.909	1.712	2.635	49.748

Table 4.3: Precision metrics per Lambda for Trajectory 1

Comparison With C

This section will outline the differences between the SQL and C execution since the SQL represents an offline execution and C represents the online the differences here is to underline the possible loss of accuracy in the process. In order to keep this part concise a table with all variables above will be given and choose the differences of the distances between the offline and the online trajectories.

Trajectory 1

	Lambda			
	0.75	0.5	0.25	0.01
Difference of points	208	254	154	7
Difference of Frechet Distance	0	0	0	-393.393
Difference of Hausdorff Distance	0	0	0	-393.393
Difference of DynTimeWarp Distance	555.155	6945.389	22660.099	830547.714
Difference of Temporal Distance	0.0284	0.0239	0.0175	1.842

Table 4.4: Comparison Precision metrics per Lambda for Trajectory 1

Chapter 5

Comparison

This chapter focus on the comparison of our implementation with existing other implementation in MobilityDB.

5.1 Douglas Peucker

5.1.1 Performance

Number of Points	Lambda				
	5	2	0.02	0.002	0.0002
100	00.000075	00.000107	00.000691	00.000581	00.000606
1000	00.00019	00.000194	00.056759	00.055591	00.056859
10000	00.001531	00.001675	05.721065	05.798434	05.600994
100000	00.017149	00.013302	EOF	EOF	EOF
1000000	00.169507	00.147934	EOF	EOF	EOF

Table 5.1: Average Execution Time by Number of Points and Distance

5.1.2 Similarity

5.2 MinDist

5.2.1 Performance

5.2.2 Similarity

	Douglas		
SquishE	0.02	0.002	0.0002
1x	18	67	768
Frechet	-3015	-3215	-617.274
Hausdorff	-3015	-3215	-617.274
DTW	-90073593	-15597514	-5325534
Temporal	1899	129.712	1.397
2x	36	134	1536
Frechet	-28363	-6034	-617.274
Hausdorff	-28363	-6034	-617.274
DTW	-262452925	-47034601	-8163405
Temporal	102	19.191	-1.590
3x	54	201	2304
Frechet	-34532	-8073	-617.274
Hausdorff	-34532	-8073	-617.274
DTW	-299629282	-60327008	-9036166
Temporal	-149	-12.72	-1.933
4x	72	268	3072
Frechet	-36615	-10146	-617.274
Hausdorff	-36615	-10146	-617.274
DTW	-325049247	-65978902	-9504295
Temporal	-278	-22.036	-2.315

Table 5.2: Comparison of differences between Douglas and SquishE (Frechet, Hausdorff, DTW)

Number of Points	Lambda				
	5	2	0.02	0.002	0.0002
100	00.000021	00.000006	00.000023	00.000014	00.000016
1000	00.000041	00.000036	00.00012	00.00012	00.000123
10000	00.000291	00.000293	00.001307	00.00134	00.001449
100000	00.003088	00.003044	00.015219	00.015609	00.016354
1000000	00.034338	00.036202	00.152152	00.163061	00.178996

Table 5.3: Average Execution Time by Number of Points and Distance of MinDist Algorithm

	Mindist		
SquishE	0.02	0.002	0.0002
1x	418	3652	16919
Frechet	2842.859	582.269	33.645
Hausdorff	2842.859	582.269	33.645
DTW	1834655	95109	21682
Temporal	1.54	0.158	0.091
2x	836	134	ALL
Frechet	598.393	582.269	-66.409
Hausdorff	598.393	582.269	-66.409
DTW	-3452102	-463775	-36834
Temporal	-3.687	-0.563	-0.647
3x	1254	201	//
Frechet	598.393	582.269	//
Hausdorff	598.393	582.269	//
DTW	-5014557	-655462	//
Temporal	-4.926	-1.192	//
4x	1672	268	//
Frechet	598.393	582.269	//
Hausdorff	598.393	582.269	//
DTW	-5960279	-768674	//
Temporal	-5.538	-1.764	//

Table 5.4: Comparison of differences between MinDist and SquishE (Frechet, Hausdorff, DTW)

Chapter 6

Conclusion

Bibliography

[1] Mohammad Ali Abam et al. "Streaming algorithms for line simplification". In: *Proceedings of the twenty-third annual symposium on Computational geometry*. 2007, pp. 175–183.

[2] Pankaj K Agarwal et al. "Near-linear time approximation algorithms for curve simplification". In: *Algorithmica* 42 (2005), pp. 203–219.

[3] Helmut Alt and MICHAEL GODAU. "Computing the Fréchet Distance between Two Polygonal Curves". In: *Int. J. Comput. Geometry Appl.* 5 (Mar. 1995), pp. 75–91. DOI: 10.1142/S0218195995000064.

[4] "Apache Kafka A high-throughput distributed messaging system . Kafka 0 . 9 . 0 Documentation 1". In.

[5] Chuck Ballard et al. *Ibm Infosphere Streams: Accelerating Deployments with Analytic Accelerators*. IBM Redbooks, 2014.

[6] *Data Stream Processing - Amazon Kinesis - AWS*. URL: http://aws.amazon.com/kinesis/ (visited on 11/11/2023).

[7] David H Douglas and Thomas K Peucker. "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature". In: *Cartographica: the international journal for geographic information and geovisualization* 10.2 (1973), pp. 112–122.

[8] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. *Data stream management: processing high-speed data streams*. Springer, 2016.

[9] GfG. *C program to implement min heap*. Apr. 2023.

[10] Ralf Hartmut Güting et al. "A foundation for representing and querying moving objects". In: *ACM Transactions on Database Systems (TODS)* 25.1 (2000), pp. 1–42.

[11] John Hershberger and Jack Snoeyink. "An O (n log n) implementation of the Douglas-Peucker algorithm for line simplification". In: *Proceedings of the tenth annual symposium on Computational geometry*. 1994, pp. 383–384.

[12] *IBM Documentation*. URL: https://www.ibm.com/docs/en/streams/4.1.0?topic=welcome-introduction-infosphere-streams (visited on 11/11/2023).

[13] Hiroshi Imai and Masao Iri. "Computational-geometric methods for polygonal approximations of a curve". In: *Computer Vision, Graphics, and Image Processing* 36.1 (1986), pp. 31–41. ISSN: 0734-189X. DOI: https://doi.org/10.1016/S0734-189X(86)80027-5.

[14] Ankit Jain and Anand Nalya. *Learning storm*. Packt Publishing, 2014.

[15] Mees van de Kerkhof. "Algorithmic and Experimental Results on Trajectory Data Processing". PhD thesis. Utrecht University, 2022.

[16] Angel Marquez. *PostGIS essentials*. Packt Publishing Birmingham, 2015.

[17] Jonathan Muckell et al. "Compression of trajectory data: a comprehensive evaluation and new approach". In: *GeoInformatica* 18 (2014), pp. 435–460.

[18] Meinard Müller. "Dynamic time warping". In: *Information retrieval for music and motion* (2007), pp. 69–84.

[19] Dmitry Namiot. "On big data stream processing". In: *International Journal of Open Information Technologies* 3.8 (2015), pp. 48–51.

[20] *S1. What Is PostgreSQL?*$_2$023. Nov. 2023.

[21] *Samza*. URL: https://samza.apache.org/ (visited on 11/11/2023).

[22] Abdul Ghaffar Shoro and Tariq Rahim Soomro. "Big data analysis: Ap spark perspective". In: *Global Journal of Computer Science and Technology: C Software & Data Engineering* 15.1 (2015), pp. 7–14.

[23] *Spark Streaming - Spark 3.5.0 Documentation*. URL: https://spark.apache.org/docs/latest/streaming-programming-guide.html (visited on 11/11/2023).

[24] Kai Wähner. *Real-time stream processing as game changer in a big data world with Hadoop and Data Warehouse*. Sept. 2014.

[25] Jay Kreps. *Putting apache kafka to use: A practical guide to building an event streaming platform (part 1)*. URL: https://www.confluent.io/blog/event-streaming-platform-1/ (visited on 11/11/2023).

[26] RR Van Hunnik et al. "Extensive comparison of trajectory simplification algorithms". MA thesis. 2017.

[27] M. Visvalingam and J. D. Whyatt. "Line generalisation by repeated elimination of points". In: *The Cartographic Journal* 30.1 (1993), pp. 46–51. DOI: 10.1179/000870493786962263.

[28] *Welcome to Apache Flume — Apache Flume*. URL: https://flume.apache.org/ (visited on 11/11/2023).

[29] Esteban Zimányi et al. "MobilityDB: A mainstream moving object database system". In: *Proceedings of the 16th International Symposium on Spatial and Temporal Databases*. 2019, pp. 206–209.