# SEMANTICALLY ENRICHED MODELLING, ANALYSIS, AND VISUALIZATION OF SIMPLIFIED LINEAR FEATURES AND TRAJECTORIES

RAJESH TAMILMANI

January 2018

GEODESY AND GEOMATICS ENGINEERING
UNB

TECHNICAL REPORT
NO. 311

# SEMANTICALLY ENRICHED MODELLING, ANALYSIS, AND VISUALIZATION OF SIMPLIFIED LINEAR FEATURES AND TRAJECTORIES

Rajesh Tamilmani

Department of Geodesy and Geomatics Engineering
University of New Brunswick
P.O. Box 4400
Fredericton, N.B.
Canada
E3B 5A3

January 2018

# PREFACE

This technical report is a reproduction of a thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering in the Department of Geodesy and Geomatics Engineering, January 2018. The research was supervised by Dr. Emmanuel Stefanakis, and funding was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC), Discovery Grants Program.

As with any copyrighted material, permission to reprint or quote extensively from this report must be received from the author. The citation to this work should appear as follows:

Tamilmani, Rajesh (2018). *Semantically Enriched Modelling, Analysis, and Visualization of Simplified Linear Features and Trajectories.* M.Sc.E. thesis, Department of Geodesy and Geomatics Engineering Technical Report No. 311, University of New Brunswick, Fredericton, New Brunswick, Canada, 139 pp.

# ABSTRACT

Multi-Scale maps provide a method of abstracting geographic features at different granularities. Polyline geometries are used to represent linear features, such as roads, rivers, and pipelines on maps. Map generalization processes are in use to represent these features either at different scales. Specifically, original geometries representing linear features at a large scale can be abstracted using a line simplification process. However, the simplification process may result in losing semantic attributes associated with the original geometries. This occurs as line simplification eliminates a series of points from the original geometries that contain attributes or characteristics relevant to the application domain. For example, points on the road network can contain information about accumulated length of the road, positional velocity, speed limit or accumulated gas consumption. This study adopts the SELF (Semantically Enriched Line simpliFication) data structure to preserve the length and other semantic attributes associated with individual points on linear geographic features at different granularities. SELF data structure has been implemented in PostgreSQL 9.4 with PostGIS extension and tested for both synthetic and real linear features such as rivers and pipelines. Further, Synchronous Euclidean Distance (SED) based simplification has been implemented to consider the temporal dimension of trajectories. The SELF data structure is built to preserve semantic attributes associated to individual points on original trajectories. Subsequently, a graph data model has been proposed to combine the simplified geometry of trajectory and the semantics lost during the simplification process. Original trajectories are simplified based on Synchronous Euclidean Distance (SED) and the Semantically Enriched Line simpliFication (SELF) data structure is built to preserve the semantics along with the simplified trajectories. These are modelled in terms of nodes and edges into Neo4j graph database for performing trajectory data analysis.  Finally, a visualization tool has been developed on top of Neo4j graph database to support the semantic retrieval of trajectories at different granularities. Historical vessel trajectories were used to test the SELF structure at various levels of simplification.  The simplified versions of these trajectories along with their semantics were modelled, analyzed and visualized in Neo4j using Cypher query language and Neo4j spatial procedures.

# DEDICATION

I dedicate my thesis work to my family. A special feeling of gratitude to my solicitous parents, Tamilmani and Ramani who has always motivated and supported from India. My brother Raguram have always supported me in difficulties.

I would like to express my indebtedness to my supervisor for his guidance, support and mentorship throughout this research.

# ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to Dr. Stefanakis for all his support and mentorship throughout my research. His support and encouragement motivated me with many opportunities for conferences, networking and research. His collaborative input and feedback was invaluable.

I would like to thank Natural Sciences and Engineering Research Council of Canada for providing funding to make this research successful.

I would also like to thank Sylvia Whitaker for providing administrative support throughout my Master’s degree.

I would like to thank Maduako Ikechukwu for his useful recommendations during the graph model implementation mentioned in this thesis.

I am thankful to my family and friends for their support and motivation.

## Table of Contents

## 3. SEMANTICALLY ENRICHED SIMPLIFICATION OF TRAJECTORIES..…44

## 4. MODELLING AND ANALYSIS OF SEMANTICALLY ENRICHED SIMPLIFIED TRAJECTORIES USING GRAPH DATABASES…….…………86

# LIST OF FIGURES

# LIST OF TABLES

# 1. Introduction

Multi-Scale maps provide a method of abstracting geographic features at different granularities. Multi-Scale representation in geographical information system (GIS) applications solely depends on the cartographic generalization methods. Generalization is a collection of processes for abstracting the level of graphical details which can be presented at a particular map scale. The most common and fundamental generalization process is simplification, which removes the high-density vertices from the linear features (e.g. rivers, pipelines and roads) based on a given criterion. The process of simplification results in reducing the complexity and redundancy of linear features. In addition, the simplification process may result in losing the geometric properties associated with the original linear geometries, as a set of intermediate points will be eliminated. These intermediate points can contain attributes or characteristics depending on the application domain. For example, points on a road network can contain information about the accumulated length of the road, positional velocity, speed limit or accumulated gas consumption.

The advent of satellite technologies has enabled the usage of GPS devices on moving objects. GPS devices mounted on moving objects generate streams of geo-location data, which describe the path travelled by the object during a period of time. This path is called trajectory.  Common application domains using trajectory data are city planning, transportation manage-ment systems, and other location-aware applications [*K. Buchin et al.* 2008]. In the era of big data, graph databases address the major challenges in management and analysis of voluminous data. The concept of storing and representing data in terms of nodes, edges and properties makes graph databases different from relational databases and well suited for trajectory data management systems [*Stefanakis* 2017]. Spatial analysis capabilities have already been added to graph database systems. For instance, Neo4j, one of the most prevalent graph database systems, provides of a spatial plugin called Neo4j Spatial to facilitate spatial operations on geo-spatial data modelled using graphs [*Neo4j Spatial Plugin* 2017].

This thesis focuses on retaining the semantics lost during the process of simplification of linear geographic features including trajectories. This research commenced with implementing SELF (Semantically Enriched Line simpliFication) data structure for static linear features [*Stefanakis* 2015]. Further, an implementation of the SELF structure for dynamic linear features has been carried out. Finally, a graph model for representing simplified trajectories along with their semantics has been proposed. This model can facilitate the analysis and visualization of simplified trajectories using graph databases. This is an article-based thesis, which is presented and supported through the following three papers:

Paper 1 (Peer Reviewed)
**Tamilmani R,** Stefanakis E, 2017. Enriched geometric simplification of linear features. *Geomatica Vol. 71, No.1, 2017, pp. 3 to 19*. doi: dx.doi.org/10.5623/cig2017-101

Paper 2 (Under Review)
**Tamilmani R,** Stefanakis E, 2017. Semantically enriched simplification of trajectories.

Paper 3 (Under Review)
**Tamilmani R,** Stefanakis E, 2017. Modelling and Analysis of Semantically Enriched Simplified Trajectories using Graph Databases.

### 1.1 Thesis Structure

This research is presented as a five-chapter, article-based thesis, Figure 1.1. Chapter 1 introduces the motivation for this research. The next three chapters (Chapter 2 to Chapter

4) present peer reviewed or under review articles, at the moment of drafting the thesis. Chapter 5 provides a summary and conclusion of the presented research as well as future opportunities. In Chapters 2 through 4, the primary research was conducted by the first author while the co-author provided supplementary advice on content.

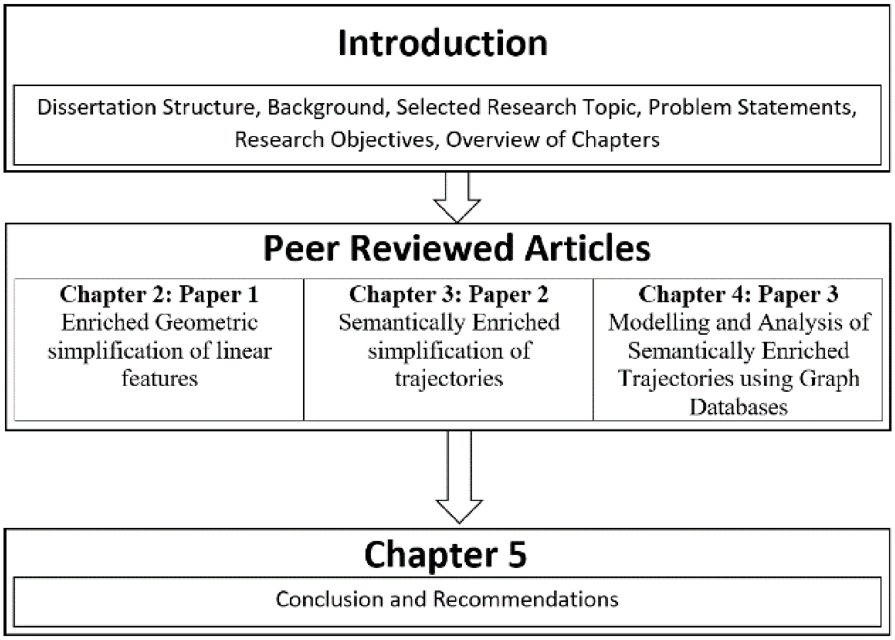

Introduction
Dissertation Structure, Background, Selected Research Topic, Problem Statements, Research Objectives, Overview of Chapters
Peer Reviewed Articles
Chapter 2: Paper 1
Enriched Geometric simplification of linear features
Chapter 3: Paper 2
Semantically Enriched simplification of trajectories
Chapter 4: Paper 3
Modelling and Analysis of Semantically Enriched Trajectories using Graph Databases
Chapter 5
Conclusion and Recommendations

**Figure 1.1** Thesis structure

## 1.2 Background

### 1.2.1 Cartographic Generalization

Map generalization is an important concept in cartography that aims at abstracting (or reducing) the level of details on a map at different scales [*Weibel* 1996]. While there are considerable processes in map generalization, such as selection, combination, smoothing, enhancement and simplification, the fundamental and common generalization process in cartography is simplification. Simplification is the process of removing high-density vertices from the linear map features (e.g. rivers, pipelines and roads) based on a given criterion. The Douglas-Peucker algorithm is extensively used for simplifying lines and provides a simplified version of an original line by controlling the offset while minimizing the distortion. The simplified version is formed only by retaining a subset of the vertices and this results in a considerable reduction in length of the line based on a threshold parameter provided by the user [*Douglas et al.* 1973]. As a result, the accumulated length at each point on the original line is not preserved in the simplified line.

### 1.2.2 Trajectory Simplification

Over the years the usage of GPS devices in mobility vehicles has increased exponentially and massive amounts of data are being generated by these devices. This data is used in various public and business applications such as urban transportation planning, fleet management and traffic modelling [*K. Buchin et al.* 2008]. The enormous volume of data does not allow for analytical methods to be applied. For example, a trip duration of 30 minutes, with the location being recorded every 5 seconds, will result in a total of 360 points. In a single day, the dataset will grow to 17,280 points. It necessitates the identification of the methods for reducing the complexity of the dataset while retaining its main characteristics. The process of reducing the volume of a trajectory dataset is called trajectory reduction or simplification. The process of trajectory reduction has evolved from cartographic generalization.

The Douglas-Peucker (DP) algorithm is a recursive approach for simplifying linear features. It takes as input the original linear geometry and a threshold distance. The simplified version of the linear geometry is generated by controlling the offset while minimizing the distortion. At the end of a recursive process, only a subset of the vertices is retained to form the simplified geometry. The resultant geometry ends up in reduction in length [*Douglas et al.* 1973]. Douglas-Peucker algorithm has limited scope to be utilized in trajectory simplification. DP simplification algorithm does not consider the temporal dimension (time) associated with the vertices of trajectories. Furthermore, the semantics (e.g. speed, heading and distance travelled) of those points of the original line (trajectory) eliminated by the simplification are not preserved in the simplified version. As DP algorithm has the limitation of not being able to consider the temporal dimension of a trajectory, the notion of the Synchronous Euclidean Distance (SED) was introduced by Meratnia and de By. The basic idea of SED is to retain certain points which are more significant in forming the trajectory than other points as they better convey the trajectory characteristics for a particular application domain.

Over the years, researchers have focused on modelling and analyzing trajectories using graphs. However, the tremendous amount of data points contained in trajectories turns the handling of graphs inefficient.

### 1.2.3 SELF (Semantically Enriched Line simpliFication)

SELF data structure has been introduced by Stefanakis [2015] to enrich the simplified line with semantics associated to the original version while achieving efficient generalization of trajectories by any of the simplification algorithms. The author has defined two variants in SELF structure based on how detailed the semantics attached to the simplified geometry are to be.

The basic variant of SELF attaches the original line length (e.g., kilometric travel distance) to the simplified line. In this variant, a line with end points *1* (start), *n* (end), and total length (dn) will be represented by a simplified line defined as follows [*Stefanakis* 2015]:

$$[x_1, y_1, x_n, y_n, d_n] \text{ (SELF variant: basic)}$$

An advanced variant for function lines will also tag the accumulated length per vertex along the line. Hence, each vertex *K* of the original line will orthogonally be projected on the simplified line and the footprint point *K′* will be assigned the accumulated length $d_k$ from point *1* (start) to vertex *K* along the original line. If $d_k'$ is the Euclidean distance of point *K′* from end point *1,* the simplified line will be represented as follows [*Stefanakis* 2015]:

$$[x_1, y_1, x_n, y_n, d_n, \text{ARRAY } \{(d_k', d_k);\ k=2, \ldots, n-1\}] \text{ (SELF variant: advanced-function)}$$

For supporting the trajectory data enrichment, the advanced variant of SELF structure has been extended to tag trajectory semantics: speed, heading, time and distance travelled. DP-SED algorithm works well for trajectory simplification, as it retains the spatiotemporal characteristics of the trajectory. Each point on the original trajectory is projected on the generalized version based on SED. The footprint of each point will be assigned with speed, heading, time and distance travelled at that point.

$$[x_1, y_1, x_n, y_n, d_n, \text{ARRAY } \{(d_k', \text{speed, heading, time}, d_k);\ k=1, \ldots, n\}] \text{ (SELF variant: trajectory)}$$

### 1.3 Research Topic

The primary purpose of this research is to retain the semantic and geometric attributes associated with individual locations of original linear features and trajectories in their simplified versions. This has been accomplished by enriching the representation of the simplified lines with an array of values corresponding to multiple locations along the original lines. To this end, a graph model to represent the simplified geometry of trajectories along with their semantics has been proposed. Then, trajectories can be analyzed using Cypher query language and Neo4j spatial procedures. A visualization tool on top of Neo4j is developed for semantic interpolation at different of trajectory simplification.

## 1.4 Problem Statement

The massive trajectory dataset becomes difficult to handle as the millions of raw data points make the processing complex. Thus, trajectory simplification techniques should be utilized to reduce the number of points in a trajectory. While the traditional simplification algorithms use the distance offset as a criterion to eliminate the redundant points, temporal dimension in trajectories should also be considered in retaining the points which convey both the spatial and temporal characteristics of the trajectory. At the same time, the simplification process results in losing the geometric and semantic attributes associated with the intermediate points on the original geometries.

## 1.5 Research Objectives

The primary purpose of this research is to retain the geometric (length) and semantic attributes associated with individual locations of original linear features by associating the semantic values to the simplified geometry as an array of values corresponding to multiple locations along the simplified geometry [*Stefanakis* 2015].

The specific research objectives are as follows:

- Implement SED based trajectory simplification technique to consider spatio-temporal data in trajectory generalization
- Implement the SELF structure to support static and dynamic polylines and to test with both synthetic and real world features.
- Propose a graph model for combining simplified geometry of trajectory and SELF structure and perform trajectory data analysis on the modelled data using Cypher query language and Neo4j spatial procedures.
- Develop a visualization tool on top of Neo4j for semantic interpolation at different levels of trajectory simplification

### 1.6 Data

The datasets used for demonstrating the effectiveness of the SELF structure in semantic interpolation include both public domain data and open source data. Table 1.1 lists out the datasets used in each chapter. These datasets are freely available for everyone to use. The study area for the chapter 2 includes linear features from New Brunswick province while the chapters 3 and 4 involve historical trajectory data of moving vessels collected over the Aegean Sea.

**Table 1.1** Data sources and description

| Data Type | Source | Location | Description | Chapter |
|---|---|---|---|---|
| ESRI Shapefiles (Linear geometries) | GeoNB Data Catalogue [2014] | New Brunswick, Canada | Three river streams, which are part of the "North Tay River," the "Waasis Stream" | **Chapter 2**<br>Tamilmani, R., Stefanakis, E., [2017] |

| | | | | |
|---|---|---|---|---|
| | | | and the “South Branch Rusagonis Stream,” as well as two pipelines, which are part of the “MNP Moncton Lateral” and the “MNP Utopia lateral”. | |
| Comma-Separated Value file containing individual locations and semantics of the moving vessel | MarineTraffic Automatic Identification System [2017] | Aegean Sea, Greece | Individual locations and the sematic attributes of the moving vessels in the Aegean Sea | **Chapter 3 and 4**<br>Tamilmani, R., Stefanakis, E., [2017] |

### 1.7 Chapter Summaries

In Chapter 1, the background information, motivation, and structure of the thesis have been presented. In addition to that, the overall concepts of cartographic generalization and trajectory simplification have been introduced and the limitations of the existing methods were described.

Chapter 2 describes the steps followed in implementing the SELF structure for managing static linear features. The literature review part of this chapter discusses about cartographic generalization and introduces the SELF data structure for linear features. The implemented algorithm applies two kinds of compression on the SELF structure known as Point level and Segment level. The effectiveness of SELF data structure in semantic interpolation at different levels of simplification is tested with both synthetic and real world features.

Chapter 3 provides a literature review about trajectory simplification and briefly describes the SED based simplification and SELF structure for dynamic linear features. Further, the steps followed to implement SED simplification technique and build the SELF structure are described. Explanations of the experiments with various real-world trajectories have been presented.

Chapter 4 introduces a graph model for transforming the semantically enriched simplified trajectory to a graph. Nodes and edges can then be analyzed using graph query languages and ad-hoc geospatial procedures in a graph database. This chapter also discusses the functionality of a visualization tool developed for helping the user in performing semantic interpolation at different levels of simplification.

Chapter 5 concludes this research. This chapter also discusses the future potential of this research.

**REFERENCES**

Douglas, D.H. and Peucker, T.K., 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer,* 10 (2), 112–122. doi:10.3138/FM57-6770-U75U-7727

K. Buchin, M. Buchin, and J. Gudmundsson. Detecting single file movement. In *GIS '08: Proceedings of the 16thACM SIGSPATIAL international conference on Advances in geographic information systems,* pages 1-10, New York, NY, USA, 2008. ACM.

GeoNB Data Catalogue. March 11, 2014. New Brunswick, Canada [cited February 6, 2017]. Retrieved from http://www.snb.ca/geonb1/e/DC/catalogue-E.asp

MarineTraffic, 2017. Live-ships map: AIS [online]. Available from: http://www.marinetraffic.com/ais/ [Last visited, June 27, 2017]

Meratnia, N. and de By, R.A., 2004. Spatiotemporal compression techniques for moving point objects. In: *Proceedings of the international conference on extending database technology* (EDBT). Berlin: Springer, 765–782. LNCS 2992.

Neo4j Spatial Plugin, http://neo4j-contrib.github.io/spatial/0.24-neo4j-3.1/index.html, Accessed on: 17th August 2017

Stefanakis, E., 2015. SELF: Semantically enriched Line simpliFication. In: *International Journal of Geographical Information Science*, Vol. 29, Iss. 10, 2015, Pages 1826-1844 doi: 10.1080/13658816.2015.1053092

Stefanakis, E., 2017. Graph Databases – Recent development in Neo4j may help accommodate the Geospatial Community. *GoGeomatics. Magazine of GoGeomatics Canada*. January 2017.

Tamilmani R, Stefanakis E, 2017. Enriched geometric simplification of linear features. *Geomatica Vol. 71, No.1, 2017, pp. 3 to 19*. doi: dx.doi.org/10.5623/cig2017-101

Weibel, R., 1996. A typology of constraints to line simplification. In: *Proceedings of 7th international symposium on spatial data handling*, 12–16 August, Delft. IGU, 533–546.

## 2. Enriched geometric simplification of linear features

### Abstract

Polyline geometries are used to represent linear features such as roads, rivers and pipelines on maps. The generalization process ends up with a polyline that represents the feature at either a different resolution or different scale. In addition, the simplification process may result in losing the geometric properties associated with the intermediate points on the original geometries. These intermediate points can contain attributes or characteristics depending on the application domain. For example, points on the road network can contain information about accumulated length of the road, positional velocity, speed limit or accumulated gas consumption. This paper involves implementing the SELF (Semantically Enriched Line simpliFication) data structure to preserve the length attributes associated to individual points on actual linear features [*Stefanakis* 2015]. The number of points to be stored in the SELF structure is optimized by applying alternative compression techniques. The data structure has been implemented in *PostgreSQL 9.4* [2014] with *PostGIS* [2016] extension using PL/pgSQL to support static and non-functional polylines. Extended experimental work has been carried out to better understand the impact of simplification to both synthetic and real (natural and artificial) linear features such as rivers and pipelines. The efficiency of SELF structure in regard to geometric property preservation was tested at various levels of simplification.

---

Reprinted from Geomatica Vol. 71, Tamilmani R, Stefanakis E, 2017. Enriched geometric simplification of linear features. No.1, 2017, pp. 3 to 19. doi: dx.doi.org/10.5623/cig2017-101

## 2.1 Introduction

Multi-Scale maps provide a method of abstracting the Earth's geographic features using different levels of detail at multiple scales. While this concept has existed for hundreds of years, multi-scale representation in geographical information system (GIS) applications solely depend on the cartographic generalization methods. Generalization is the process of simplifying the level of graphical details which can be presented at a particular map scale. The most common and fundamental generalization process is simplification, which removes the high-density vertices from the linear features (e.g., rivers, pipelines and roads) based on a given criterion. The process of simplifying aides in reducing the complexity and redundancy in a dataset.

The Douglas-Peucker algorithm is extensively used for simplifying lines and provides a simplified version of an original line by controlling the offset while minimizing the distortion. The simplified version is formed only by retaining a subset of the vertices and this results in a considerable reduction in length of the line based on threshold parameter provided by the user [*Douglas et al.* 1973]. As a result, the accumulated length at each point on the original line is not preserved in the simplified line. The detailed study on Douglas-Peucker algorithm demonstrates that it is the most visually effective line simplification algorithm [*Wu et al.* 2003]. In Fig. 2.1, only the first and last points of original line are retained in the simplified line. The Douglas-Peucker algorithm also retains some intermediate points depending on the threshold distance.

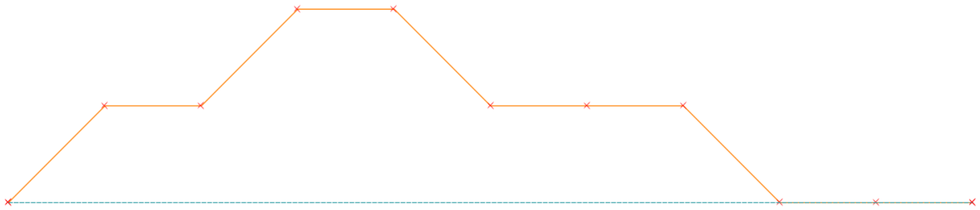


FIGURE 2.1 : COMPARISON OF ORIGINAL (SOLID ORANGE) AND THE SIMPLIFIED (BLUE DASHED) VERSION

This paper presents an implementation of the SELF (Semantically Enriched Line simpliFication) data structure for static linear features that preserves the length attribute associated with individual locations of original lines, especially for the accumulated length on original lines. This attribute is associated to the simplified line as an array of values corresponding to multiple locations along the simplified line. The SELF structure can store any semantic annotations associated with individual locations or segments of the original line [*Stefanakis* 2015].

The purpose of this research is to retain the geometric (length) attribute associated with individual locations of original lines. This has been accomplished by associating the accumulated length values to the simplified segment as an array of values corresponding to multiple locations along the simplified segment [*Stefanakis* 2015]. The research objectives are:

1. To implement the SELF structure to support static and non-functional polylines and to test with both synthetic and real world features.
2. To compare the interpolated distance values using SELF structure at different levels of simplification.

This paper is organized as follows. Section 2.2 provides a literature review about cartographic generalization and introduces the SELF data structure for linear features.

Section 2.3 describes the steps followed to implement the SELF structure. Section 2.4 presents PostGIS functionalities and explanations of the experiment with various real world features. Section 2.5 summarizes the contribution of this paper and introduces future developments for the SELF data structure with respect to dynamic lines and testing with various application domains.

## 2.2 Literture Review

### *2.2.1 Cartographic Generalization*

Generalization is an important concept in cartography that aims at simplifying the level of details on a map at different scales [*Weibel* 1996]. While there are considerable techniques in generalization such as selection, combination, smoothing, enhancement and simplification, the fundamental and common generalization process in cartography is simplification. Simplification of linear features has acquired a continuous growth of research over the years by cartographers [*Cromley* 1991, *Weibel* 1997, Robinson *et al.* 2005].

The Douglas-Peucker line simplification algorithm, an improved classis, was introduced to address the problem of topological inconsistency between original and simplified 2D polylines. The algorithm avoids the self-intersections on the simplified version [*Wu et al.* 2003]. Line simplification algorithms have been experimented in a streaming environment where the amount of storage is limited, so that all the points cannot be stored [*Abam* 2010].

*Richter et al.* [2012] introduced the concept of semantic trajectory compression which allows for compression of trajectory data while permitting minimal and acceptable loss in the information associated with the individual points on the trajectory. This information depends on the application domain and the nature of the trajectory. The algorithm they proposed enables a user to determine the reference point and all possible movement change descriptions from that point.

Various techniques have been proposed to enforce the topological constraints while simplifying a polyline [ *Shahriari and Tao* 2002, *Titus et al.* 2015, *QiuLei et al.* 2016]. According to Shahriari and Tao there is no simplifying algorithm that calculates the threshold values based on the desired accuracy level. The authors propose adaptive tolerance line simplification in which the user supplies the target level for desired accuracy and the simplification tolerance value is calculated accordingly. Recently, a series of attempts have been made to enrich the content of linear features to address the problem of annotating trajectories with semantic data. [*Alvares et al.* 2007, *Yan et al.* 2011, *Richter et al.* 2012, *Parent et al.* 2013].

SELF data structure has been introduced by *Stefanakis* [2015] to enrich the simplified line to convey some semantics associated with the original version. He categorized the lines as functional or non-functional based on their relation between the originals and simplified lines (Fig. 2.2). Further, the author provided an algorithm for decomposing non-functional lines into a finite number of functions.

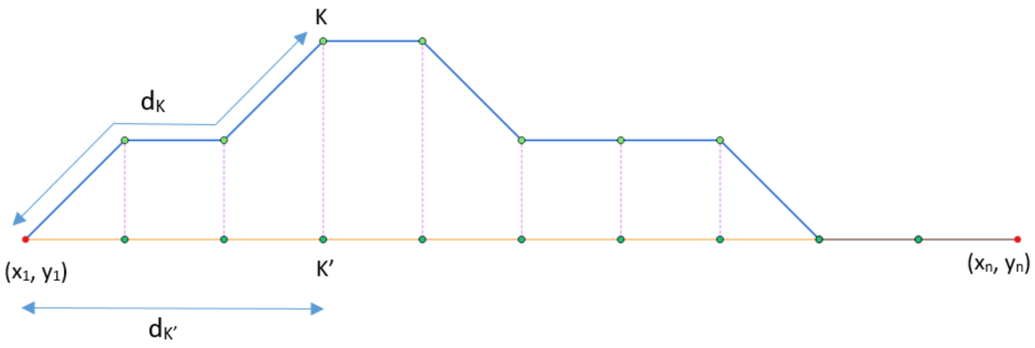


FIGURE 2.2: A FUNCTIONAL LINE SIMPLIFIED INTO A STRAIGHT-LINE SEGMENT. POINT K' AS A PROJECTION OF VERTEX K WILL BE ASSIGNED WITH THE ACCUMULATED LENGTH OF THE ORIGINAL LINE FROM ONE END POINT TO VERTEX K.

### *2.2.2 SELF (Semantically Enriched Line simpliFication)*

SELF is a data structure that preserves the attributes of the original line or any semantic annotations associated with individual locations or segments of that line [*Spaccapietra et al.*2008] into the generalized version. SELF has many variations depending on how rich the semantics attached to the simplified line are.

The *basic variant* of SELF attaches the original line length (e.g., kilometric travel distance) to the simplified line. In this variant, a line with end points *1* (start), *n* (end), and total length ($d_n$) will be represented by a simplified line defined as follows [*Stefanakis* 2015]:

$$[x_1, y_1, x_n, y_n, d_n] \text{ (SELF variant: basic)}$$

An *advanced variant* for function lines will also tag the accumulated length per vertex along the line. Hence, each vertex *K* of the original line will orthogonally be projected on the simplified line (Fig. 2.2) and the footprint point *K′* will be assigned the accumulated length $d_k$ from point *1* (start) to vertex *K* along the original line. If $d_k'$ is the

Euclidean distance of point $K'$ from end point 1, the simplified line will be represented as follows [*Stefanakis* 2015]:

[$x_1$, $y_1$, $x_n$, $y_n$, $d_n$, ARRAY {($d_k'$, $d_k$); k=2, …, n-1}] (SELF variant: advanced-function)

### 2.3 Methodology

A Polyline can be represented as a sequence of points {$P_1$, $P_2$ …. Pn}, where $P_i$ is a vertex on the polyline and n is the number of points on the polyline. The simplified line using Douglas-Peucker algorithm with the user defined threshold, always has the number of vertices which is less than or equal to the number of vertices on the original line.

The goal of the SELF structure is to retain the accumulated length at each point on the original line, along with the accumulated length of the corresponding point on the simplified line in the SELF structure.

The algorithm is divided into six steps:

1. Finding the orthogonally projected vertex on the simplified line for each point on the original line
2. Identifying the category of the original line. The categories are static functional lines, static non-functional lines [*Stefanakis* 2015]
3. Calculating the accumulated length at each intermediate point on the original line and corresponding projected point on the simplified line
4. Remove the individual segment based on the segment compression threshold and the points based on the point compression threshold
5. Identifying and managing special cases
6. Computation of the accumulated length on the original line at any point on the simplified line

### 2.3.1 Orthogonal projection of the point on the simplified line:

The algorithm runs ST_SIMPLIFY (geometry, threshold) method in PostgreSQL which operates based on the Douglas-Peucker algorithm. Each point on the original line is projected vertically on the simplified line.

In Fig. 2.3 the coordinate of point “C” which is the perpendicular projection of point “D” on the line “AB” can be computed by algorithm 2.1. Fig. 2.4 illustrates the orthogonal projection of all the points on the original line.

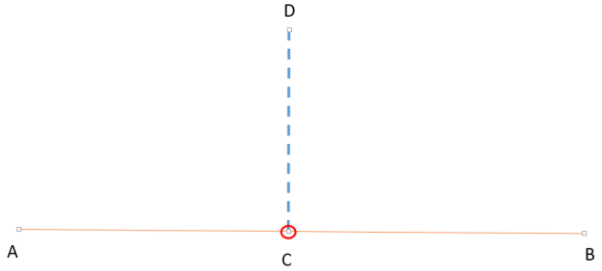


FIGURE 2.3: POINT ‘D’ IS VERTICALLY PROJECTED ON THE LINE ‘AB’

```
Algorithm 2.1: Finding the orthogonal projection of the point on the simplified line (Fig. 2.3)
Input:
    1. Starting Point of the simplified line (A)
    2. Ending Point of the simplified line (B)
    3. Point on the original line to be projected on the simplified line (D)

Output:
    1. Orthogonally projected point on the simplified line returned as geometry type
Steps:
    1. Retrieve the X and Y coordinate of the three input points
    2. Calculate the displacement ($d_{AB}$) = $(X (B) – X (A))^{2} + (Y (B) – Y (A))^{2}$;
    3. Find the Unit point ($U_{P}$) = $((X (D)-X (A)) *(X (B) – X (A)) + (Y (D) – Y (A)) *(Y (B) – Y (A)) )/d_{AB}$;
    4. Finally, the X and Y coordinate of Point ‘C’ can be obtained by,
     $X(C) = X (A) + ( U_{P} * (X (B) – X (A)))$;
     $Y(C) = Y (A) + ( U_{P} * (Y (B) – Y (A)))$;
```

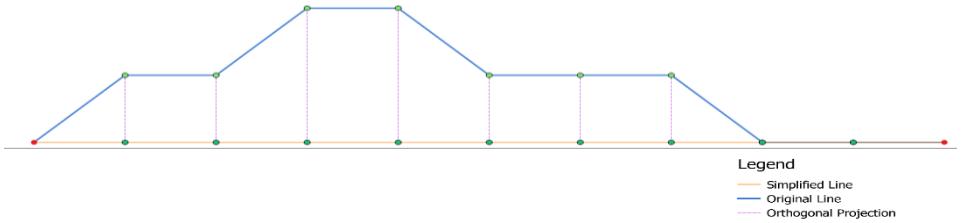


FIGURE 2.4: ILLUSTRATES THE ORTHOGONAL PROJECTION OF THE POINT ON THE SIMPLIFIED LINE WHERE THE STARTING AND THE ENDING POINTS OF BOTH SIMPLIFIED AND ORIGINAL VERSIONS ARE THE SAME.

### 2.3.2 Identifying the line type:

In case of a functional line, each point on the simplified line corresponds to a single point on the original line. On the other hand, a point on the simplified line may correspond to multiple points on the original line and these types of lines belong to a non-functional category. The SELF method proposes the decomposition of non-function into a set of functions [*Stefanakis* 2015]. In Fig. 2.5 the point "B" on the simplified line corresponds to the points "$B_1$" and "$B_2$". Similarly, the points "$C_1$" and "$C_2$" on the original line are projected to the same point "C" on the simplified line. Therefore, the original line should be decomposed into three parts: {1…$C_1$}, {$C_1$….$A_1$}, {$A_1$…n} as shown in the Fig. 2.6 by algorithm 2.2

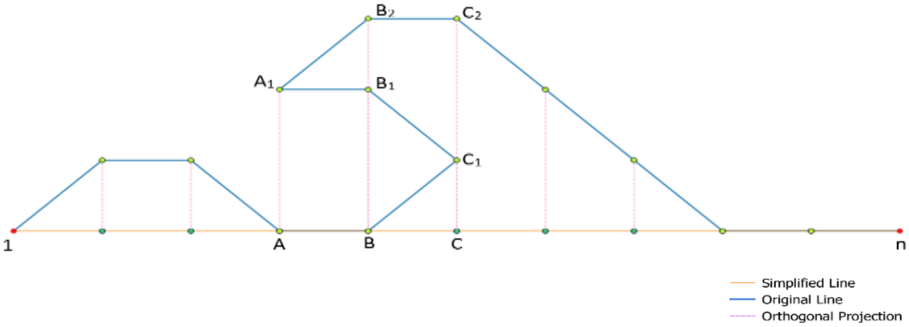


FIGURE 2.5: A NON-FUNCTION LINE SIMPLIFIED INTO A STRAIGHT-LINE SEGMENT

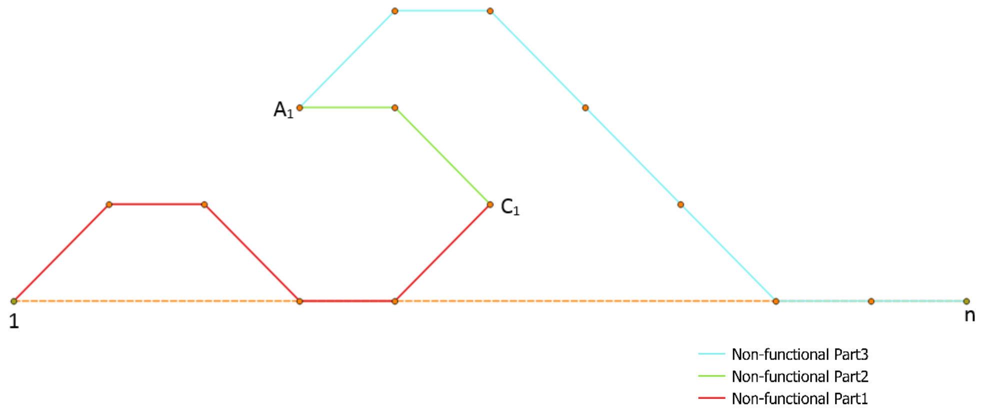


FIGURE 2.6: RESULT OF DECOMPOSING THE NON-FUNCTION INTO THREE FUNCTIONS

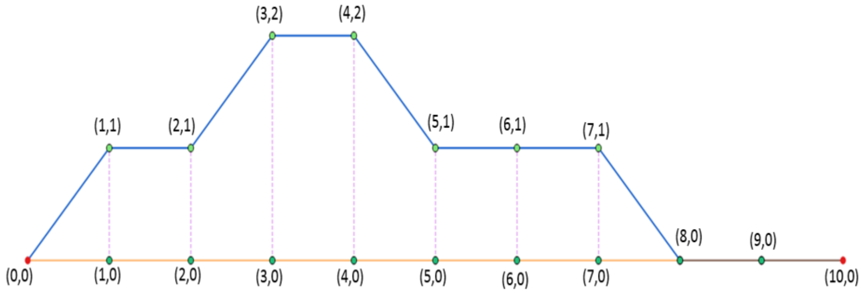
**Algorithm 2.2:** Decomposing the Non-functional line into functions

```
Input:
    1. Geometry of the non-functional line to be decomposed
    2. Threshold value for running Douglas-Peucker(DP) algorithm
Output:
    1. Functional lines as an array
Steps:
    1. Define array of geometry to store the functional lines (A)
    2. Create a new function(F1) to store the points
    3. Store the starting point of the non-functional line in F1
    4. Run the DP algorithm for getting the simplified line geometry
    5. FOR EACH point 'P' on the original line (P excludes the ending point of a line)
        1. Find the orthogonally projected point 'P' on the simplified line (P')
        2. Find the orthogonal projection of the point P+1 (P1')
        3. Calculate the accumulated length at the point P'(L1), P1'(L2) on the
           simplified line
        4. If L2 > L1 then
            i. Add the point P+1 to F1
            ii. Create a line geometry using the points in F1
            iii. Add the created line to the array A
            iv. Create a new function (F2)
            v. Add the point P+1 to F2
        5. ELSE
            i. Add the point P+1 to F1
        6. END
    6. END
    7. RETURN the array A
```

### 2.3.3 Calculating the accumulated length at each point

Advanced variant for SELF structure, tags the accumulated length per vertex along the line. As shown in the Fig. 2.7, each point on the original line is projected orthogonally on the simplified line. For each vertex on the original line, the corresponding point on simplified is annotated with the accumulated length until the (n-1)$^{th}$ point on the original line. The entire SELF structure is represented as follows:

[ POINT (0,0), POINT (10,0), 11.6, {(1,1.4),(2,2.4),(3,3.8),(4,4.8),(5,6.2),(6,7.2),(7,8.2),(8,9.6),(9,10.6) } ]



| | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Original Line | Segment Length | | 1.4 | | 1 | | 1.4 | | 1 | | 1.4 | | 1 | | 1 | | 1.4 | | 1 | | 1 |
| | Accumulated Length | 0 | | 1.1 | | 2.4 | | 3.8 | | 4.8 | | 6.2 | | 7.2 | | 8.2 | | 9.6 | | 10.6 | | 11.6 |
| Simplified Line | Segment Length | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 |
| | Accumulated Length | 0 | | 1 | | 2 | | 3 | | 4 | | 5 | | 6 | | 7 | | 8 | | 9 | | 10 |

FIGURE 2.7: A TABLE SUMMARIZING THE LENGTH MEASURES OF A SIMPLE LINE OF 11 VERTICES

### 2.3.4 Compression Levels

SELF structure generates a large volume of data which is proportional to the number of vertices in the original line. In order to diminish the volume, two compression methods can be applied: (a) point level (b) segment level. These methods are described in the following sections.

#### 2.3.4.1 Point Level

If the ratio between the original length connecting three intermediate points and the simplified length of the corresponding projected points on the simplified line is less than the given threshold, then the accumulated length at the middle point is not stored (Fig. 2.8).

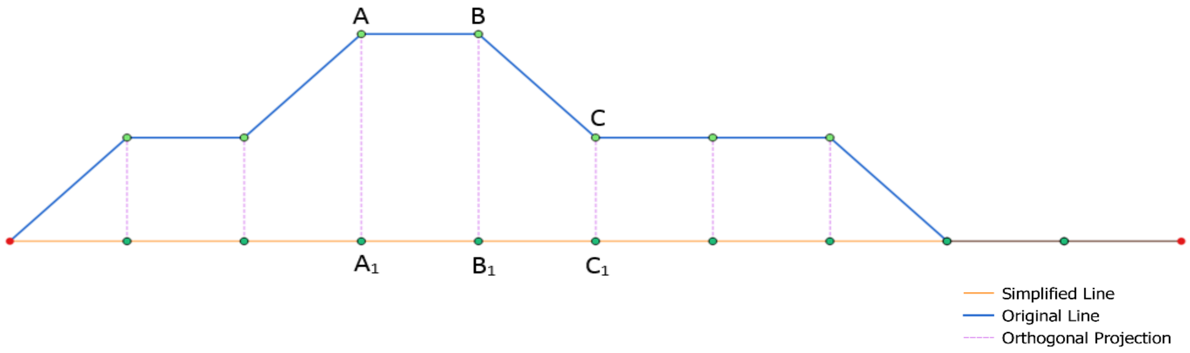


FIGURE 2.8: CONSECUTIVE POINTS A, B AND C ARE PROJECTED ON THE SIMPLIFIED LINE

$$\text{Point level compression ratio} = \frac{\text{Length of AC} - \text{Length of } A_1C_1}{\text{Length of AC}} \text{ X } 100 \quad \%$$

If three points form a straight line (collinear points), then the middle point is not stored. In Fig. 2.9 the accumulated length at point B is not stored as the points A, B, and C form the straight line.

[ POINT (0,0), POINT (10,0), 11.6 , {

(1,1.4),(2,2.4),(3,3.8),(4,4.8),(5,6.2),~~(6,7.2)~~,(7,8.2),(8,9.6),~~(9,10.6)~~ } ]

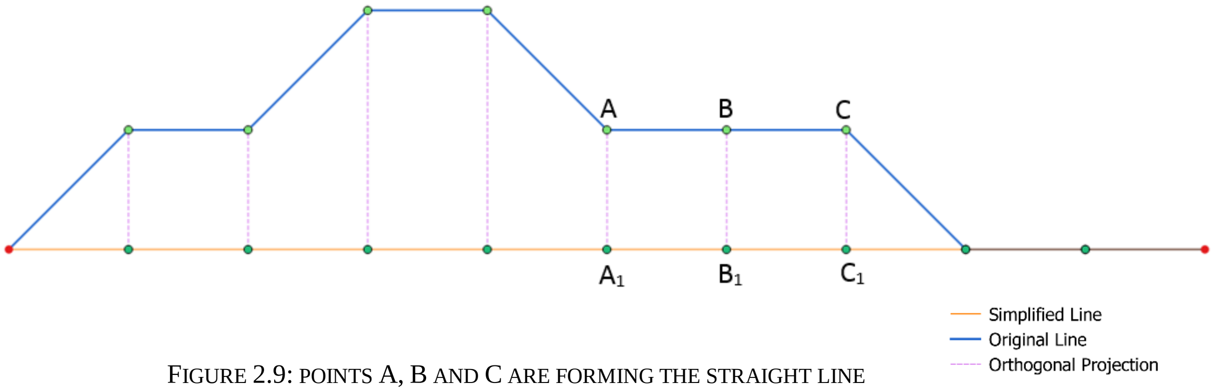


FIGURE 2.9: POINTS A, B AND C ARE FORMING THE STRAIGHT LINE

#### 2.3.4.2 Segment Level

In case of multiple straight lines forming the simplified line, the segment level threshold can be applied. In Fig. 2.10, if the ratio between original length of the segment ($L_{AB}$) and simplified length ($L'_{AB}$) of the segment is within the threshold, then all the points belonging to that segment are ignored.

$$\text{Segment level compression ratio} = \frac{\text{Original Length of AB - Simplified Length of AB}}{\text{Original Length of AB}} \text{ X } 100 \quad \%$$

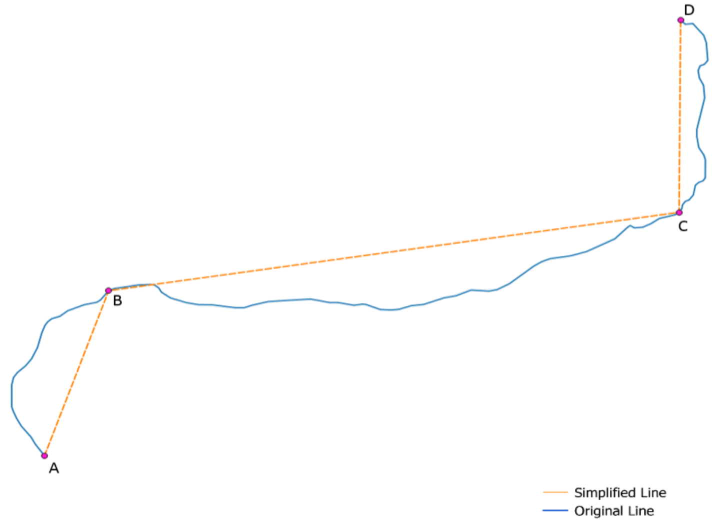


FIGURE 2.10: THE SIMPLIFIED LINE HAS THREE SEGMENTS AB, BC, AND CD

### 2.3.5 Special cases

Not all parts of the original line may always be bounded by the area defined by the perpendicular lines to the end points of the simplified line. In Fig. 2.11, the points A, B, C, D, E, F and G do not fall within the region of the simplified line.

In this case, the starting point 1 is assigned with the points A, B, C as an array of intervals and the end point n is assigned with the points D, E, F, and G.

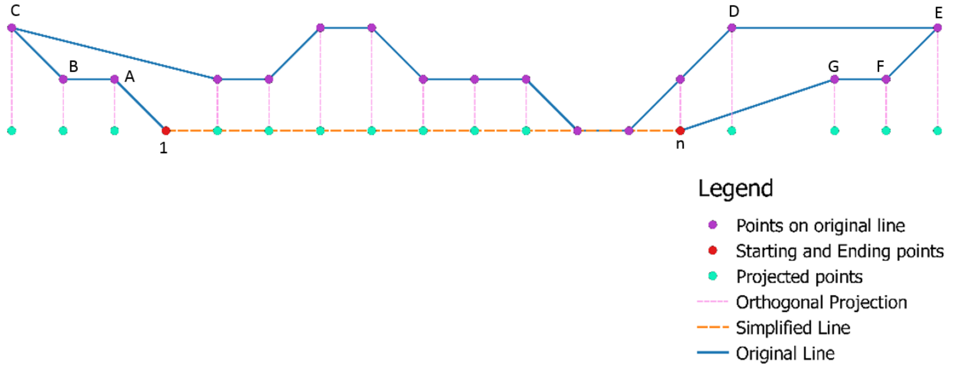


FIGURE 2.11: PROJECTION OF ORIGINAL POINTS ON THE SIMPLIFIED LINE

**Algorithm 2.4:** Computation of the accumulated length on the original line at any point of the simplified line

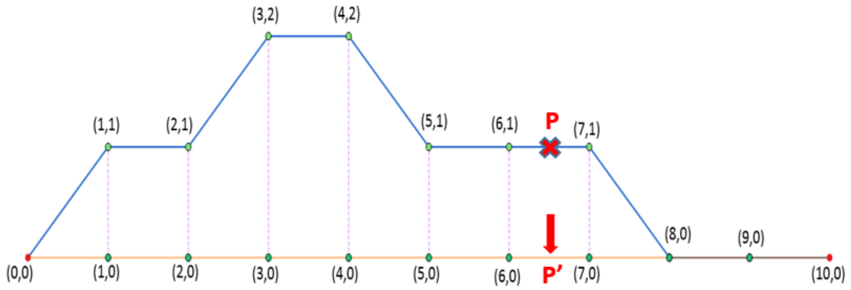
```
Input:
    1. Geometry of the simplified line
    2. SELF structure of the simplified line
    3. Point on the simplified line (P′)
Output:
    1. Computed length at P (P is the point projected at P′)
Steps:
    5. Find the distance between starting point and point P′ ($d_{P'}$)
    6. FOR EACH pair ($d_{k'}$,$d_k$) IN THE SELF array
    7. IF $d_{k'} > d_{P'}$
            a. Retrieve the pairs ($d_{k'}$,$d_k$) and ($d_{k-1'}$,$d_{k-1}$)
            b. Use linear interpolation within the retrieved pairs to compute the accumulated
               length $d_P$
    8. END
    9. RETURN $d_P$
```

**Algorithm 2.3:** Building the SELF structure

```
Input:
    1. Line geometry to be simplified
    2. Threshold value for running Douglas-Peucker (DP) algorithm
    3. Segment level compression threshold value (ST)
    4. Point level compression threshold value (PT)
Output:
    1. SELF advanced structure
Steps:
    1. Define the object of SELF structure (SELF)
    2. Add the starting point, ending point and the actual length of original line to
       SELF
    3. Define the two-dimensional array for storing the accumulated length (AL)
    4. Run DP algorithm for getting the simplified line geometry
    5. Find the number of segment in the simplified line
    6. FOR EACH segment(S) in the simplified line
        a. IF segment level compression ratio > ST THEN
            1. FOR EACH point 'P' on the Segment S (P excludes the starting
               and last two points on the original line)
                1. IF point level compression ratio at point P > PT THEN
                2. Find the orthogonally projected point on the simplified
                   line (P')
                3. Calculate the accumulate length at the point P(Lp) on
                   the original line
                4. IF P' IS NOT ON THE SIMPLIFIED LINE THEN
                    a. Accumulated length at the point P'(Lp') = 0 or
                       Length of the original line (decided based on
                       either P' is close to Starting point or ending
                       point)
                5. ELSE
                    a. Calculate the accumulated length at the point
                       P'(Lp') on the simplified line
                6. END
                7. Add Lp, Lp' to the array AL
                8. END
            2. END
        b. END
    7. END
    8. Add the accumulated length array to SELF
    9. RETURN the SELF structure (SELF)
```

### 2.3.6 Computation of the accumulated length on the original line at any point on the simplified line

The SELF structure built using algorithm 2.3 [*Appendix 1*] can be used to compute the accumulated length on the original line at any point on the simplified line. In Fig. 2.12, the accumulated length at ‘P’ can be calculated by applying a linear interpolation on the segment defined by the projection of vertices (6, 1) and (7, 1) on the simplified line. The algorithm 2.4 is used for computing the length at P.



| | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Original Line | Segment Length | | 1.4 | | 1 | | 1.4 | | 1 | | 1.4 | | 1 | | 1 | | 1.4 | | 1 | | 1 | |
| | Accumulated Length | 0 | | 1.1 | | 2.4 | | 3.8 | | 4.8 | | 6.2 | | 7.2 | | 8.2 | | 9.6 | | 10.6 | | 11.6 |
| Simplified Line | Segment Length | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| | Accumulated Length | 0 | | 1 | | 2 | | 3 | | 4 | | 5 | | 6 | | 7 | | 8 | | 9 | | 10 |

FIGURE 2.12: P′ (6.5, 0) IS THE ORTHOGONAL PROJECTION OF P (6.5, 1)

## 2.4 Implementation

The data structure has been implemented in PostgreSQL 9.4 using PL/pgSQL. The spatial extension PostGIS 2.3 has been installed in PostgreSQL 9.4 (PostgreSQL, PostGIS). The implemented algorithm takes a single linear feature and performs Douglas-Peucker line simplification, which is available in PostGIS (ST_SIMPLIFY). The simplified version is then associated with the SELF data structure and the compressed structure based on user-defined segment level threshold and the point level threshold. The user can select any point on the simplified line to retrieve the original accumulated distance. The experiments were performed on pipeline and river network data.

### 2.4.1 PostGIS Extension

Table 2.1 summarizes the built-in functions available with PostGIS extension that were utilized for developing the SELF data structure. For each function, the input and output parameters are also listed in the table.

TABLE 2.1: BUILT-IN POSTGIS FUNCTIONS USED IN DEVELOPING ALGORITHMS (SOURCE: HTTP://WWW.POSTGIS.NET/DOCS/)

| FUNCTION | INPUT | OUTPUT |
|---|---|---|
| **ST_NPoints** — Return the number of points (vertexes) in a geometry. | Line GEOMETRY | number of points in a geometry as INTEGER |
| **ST_PointN** — Return the Nth point in the Line geometry. | GEOMETRY of a line string, integer n | Nth point in a single line string as GEOMETRY |
| **ST_Length** — Returns the 2D length of the geometry in meters | GEOMETRY | 2D Cartesian length of the geometry as FLOAT |
| **ST_StartPoint** — Returns the first point of a LINESTRING geometry as a POINT. | Line GEOMETRY | Line GEOMETRY |
| **ST_EndPoint** — Returns the last point of a LINESTRING geometry as a POINT. | Line GEOMETRY | Point GEOMETRY |
| **ST_X** — Return the X coordinate of the point | Point GEOMETRY | FLOAT |
| **ST_Y** — Return the Y coordinate of the point | Point GEOMETRY | FLOAT |
| **ST_Distance** — For geometry type Returns the 2D Cartesian distance between two geometries in projected units (based on spatial ref). | GEOMETRY g1, GEOMETRY g2 | FLOAT |
| **ST_AsText** — Return the Well-Known Text (WKT) representation of the geometry/geography without SRID metadata. | GEOMETRY | TEXT |
| **ST_Simplify —** Returns a "simplified" version of the given geometry using the Douglas-Peucker algorithm. | GEOMETRY, THRESHOLD | SIMPLIFIED GEOMETRY |
| **ST_MakeLine** — Creates a Line string from array of points | GEOMETRY array | GEOMETRY |

### 2.4.2 SELF functions

Using PL/pgSQL – procedural language for PostgreSQL, the SELF structure algorithms were added as new (user defined) functions. Eight new functions were implemented. The example statement for calling each user defined function is shown in

Table 2.2 along with the output. The functions are executed on two different linear geometries "testroad" (Fig. 2.12) and "nfroad" (Fig. 2.6) where "geom" is the geometry column in the corresponding PostGIS table.

TABLE 2.2: USER-DEFINED FUNCTIONS AND EXAMPLE STATEMENTS FOR CALLING THE FUNCTIONS

| FUNCTION | INPUT | OUTPUT |
|---|---|---|
| **SELF_PP_POINT**— Returns orthogonal projection of a point on the simplified line. | Starting Point, Ending Point, Point to be projected | Point GEOMETRY |
| Select ST_ASTEXT(**SELF_PP_POINT**(ST_MakePoint(0,0), ST_MakePoint(10,0),ST_MakePoint(5,1)));<br>**Output :** POINT(5 0) | | |
| **SELF_SLP_DIFF** – Function for finding the slope difference between three consecutive points. This function is used to discard the middle point of three consecutive points which form the straight line. Returns the array of slope difference values for each point on the line (excludes starting and ending point). | line geometry | Array of numbers<br>0 – Three points form straight line<br>> 0 – Positive slope between three points<br>< 0 – Negative slope between three points |
| select **SELF_SLP_DIFF**(geom) from testroad;<br>**Output :** {-100,100,-100,-100,100,0,-100,100,0}<br>"testroad" contains 11 points. Excluding the starting and ending points the output array contains the slope difference for the 9 intermediate points. | | |
| **SELF_ACC_LEN** – Function to calculate the accumulated length at each point on the line | line geometry | Array of numerical values |
| select **SELF_ACC_LEN**(geom) from testroad;<br>**Output :** {1.41,2.4,3.8,4.8,6.2,7.2,8.2,9.6,10.6,11.6} | | |
| **SELF_CHK_PT**— Function to check whether the projected point is on the simplified line or NOT. | Starting Point, Ending Point, Point to be projected | Returns the number based on the following criteria:<br>0 – On the line<br>1 – Close to end point<br>2 – Close to Starting point |
| select **SELF_CHK_PT**(ST_MAKEPOINT(0,0),ST_MAKEPOINT(10,0),ST_MAKEPOINT(5,0));<br>**Output :** 0 | | |
| **SELF_ADV_CB**— To build the SELF structure | line geometry, Douglas Peucker threshold, Segment Level compression ratio, Point level compression ratio | Advanced SELF Structure |
| select **SELF_ADV_CB**(geom,1000.0,0.0,0.0) from testroad;<br>**Output:** [ POINT (0,0), POINT (10,0), 11.6, {(1,1.4), (2,2.4), (3,3.8), (4,4.8), (5,6.2), (6,7.2), (7,8.2), (8,9.6), (9,10.6)} ] | | |
| **SELF_ADV_ASTEXT**– To display the SELF structure in user understandable format | SELF structure | Text explaining the SELF structure |

| | | |
|---|---|---|
| select **SELF_ADV_ASTEXT**(SELF_ADV_CB(geom,500.0,10.0,0.0)) from testroad;<br>**Output :** SPOINT(0 0) -- EPOINT(10 0) – Actual Length: 11.657 – Accumulated Distance:<br>","1.000,1.414","2.000,2.414","3.000,3.828","4.000,4.828","5.000,6.243","7.000,8.243","8.000,9.657” | | |
| **SELF_NS –** Decomposing the Non-functional lines into set of function lines | line geometry | Array of functional geometries |
| select ST_ASTEXT(UNNEST(**SELF_NS**(geom))) from nfroad;<br>**Output :** "LINESTRING(0 0,1 1,2 1,3 0,4 0,5 1)"<br>"LINESTRING(5 1,4 2,3 2)"<br>"LINESTRING(3 2,4 3,5 3,6 2,7 1,8 0,9 0,10 0)" | | |
| **SELF_BUILD –** Builds the SELF structure and returns the simplified geometry with SELF structure stored in the attribute table | line geometry, Douglas Peucker threshold, Segment Level compression ratio, Point level compression ratio | Simplified geometry with SELF structure stored in its attribute table |
| select **SELF_BUILD**(geom,1000.0,0.0,0.0) from testroad;<br>**Output:** Simplified geometry with SELF structure stored in the attribute table | | |
| **SELF_ITP_DIST_ML -** To interpolate the distance using SELF structure | Simplified line geometry, SELF structure, point | Array of interpolated distances |
| select<br>SELF_ITP_DIST_ML(ST_SIMPLIFY(geom,500.0),SELF_ADV_CB(geom,500.0,0.0,0.0),ST_MAKEPOINT(4,0))<br>from nfroad;<br>**Output :** {4.828,7.657,10.071} | | |

### 2.4.3 Experimental Data

To demonstrate the effectiveness of the SELF structure in interpolating the distance, experimentation is done on five different linear features with different values for segment level and point level compression. Three river streams which are part of the “North Tay River”, the “Waasis Stream” and the “South Branch Rusagonis Stream,” as well as two pipelines which are part of the “MNP Moncton Lateral” and the “MNP Utopia lateral” have been chosen. In order for the set of features to be representative of a wide range of topological characteristics, it was decided to select a set of linear features with different number of vertices.

TABLE 2.3: LENGTH AND THE NUMBER OF POINTS AVAILABLE IN THE SELECTED DATA

| FEATURE NAME | LENGTH (in meters) | TOTAL NUMBER OF VERTICES |
|---|---|---|
| NORTH TAY RIVER | 14723.185 | 440 |
| WAASIS STREAM | 9026.166 | 305 |
| SOUTH BRANCH RUSAGONIS | 19853.897 | 461 |
| MNP MONCTON LATERAL | 12271.853 | 250 |
| MNP UTOPIA LATERAL | 8381.864 | 570 |

The data has been downloaded from GeoNB (collaborative project of the Government of New Brunswick) website (GeoNB Data Catalogue) [2017]. Fig. 2.13 to 2.17 show the original and simplified versions (DP threshold 500.0m) of river streams and pipelines, where the total number of points on the original line is mentioned in the figure captions.

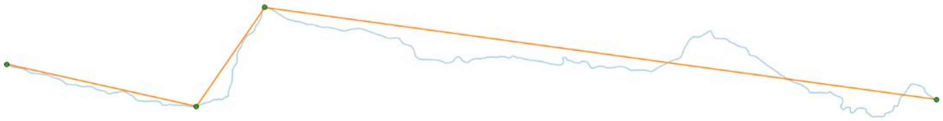


FIGURE 2.13: NORTH TAY RIVER (440 POINTS)

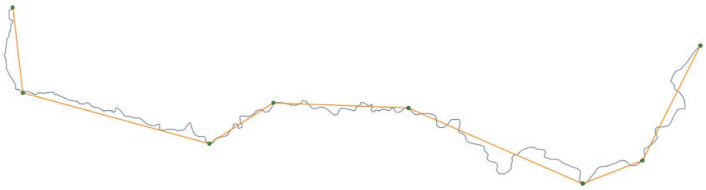


FIGURE 2.14: WAASIS STREAM (304 POINTS)

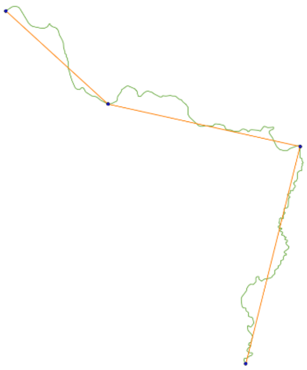


FIGURE 2.15: SB RUSAGONIS STREAM (461 POINTS)

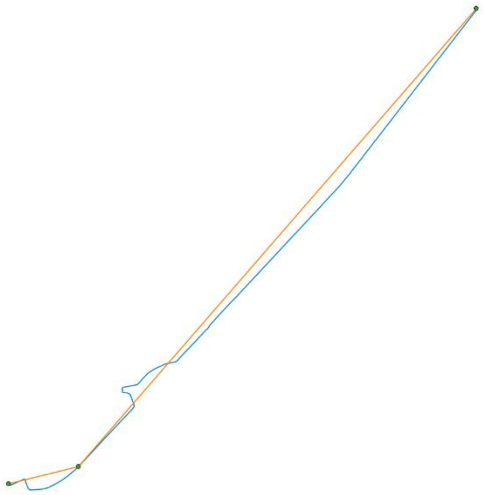


FIGURE 2.16: MNP MONCTON LATERAL (250 POINTS)

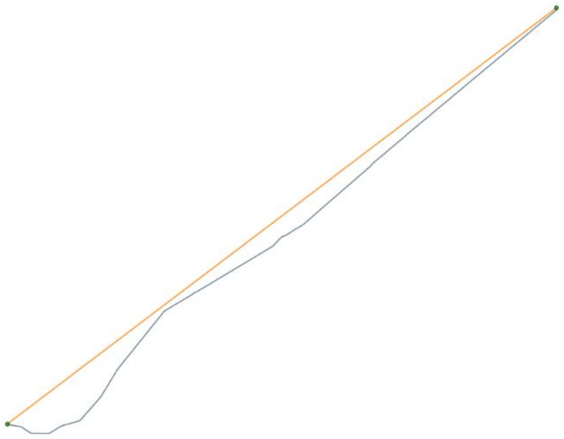


FIGURE 2.17: MNP UTOPIA LATERAL (570 POINTS)

### 2.4.4 Experiments

The SELF structure has been built on the original line shown in Fig. 2.18 with Douglas-Peucker threshold 500.0 meters and both the segment and point level threshold values as 0. The original distance at each point on the simplified line is listed in Fig. 2.19

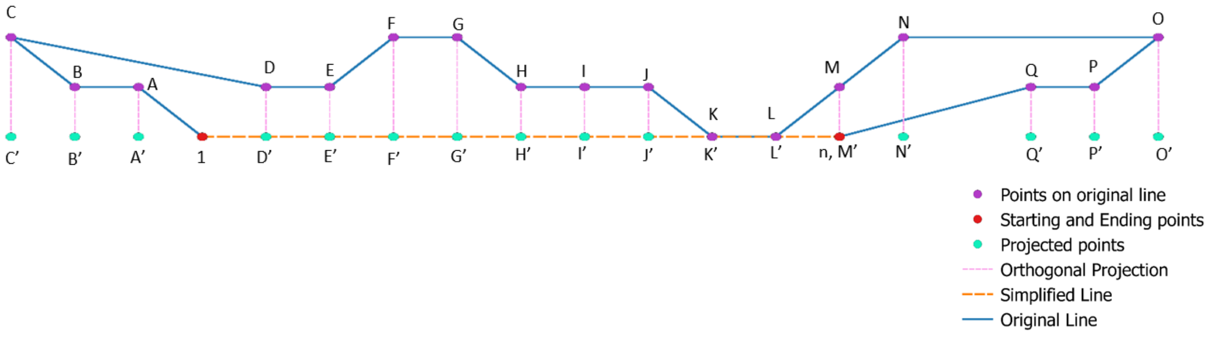
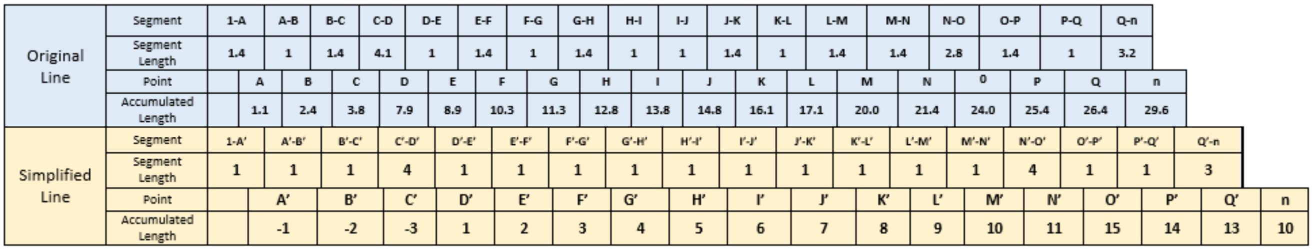


FIGURE 2.18: ORIGINAL AND SIMPLIFIED VERSION (500 M DP THRESHOLD) OF A SAMPLE LINE



| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Original Line | Segment | 1-A | A-B | B-C | C-D | D-E | E-F | F-G | G-H | H-I | I-J | J-K | K-L | L-M | M-N | N-O | O-P | P-Q | Q-n |
| | Segment Length | 1.4 | 1 | 1.4 | 4.1 | 1 | 1.4 | 1 | 1.4 | 1 | 1 | 1.4 | 1 | 1.4 | 1.4 | 2.8 | 1.4 | 1 | 3.2 |
| | Point | | A | B | C | D | E | F | G | H | I | J | K | L | M | N | 0 | P | Q | n |
| | Accumulated Length | | 1.1 | 2.4 | 3.8 | 7.9 | 8.9 | 10.3 | 11.3 | 12.8 | 13.8 | 14.8 | 16.1 | 17.1 | 20.0 | 21.4 | 24.0 | 25.4 | 26.4 | 29.6 |
| Simplified Line | Segment | 1-A′ | A′-B′ | B′-C′ | C′-D′ | D′-E′ | E′-F′ | F′-G′ | G′-H′ | H′-I′ | I′-J′ | J′-K′ | K′-L′ | L′-M′ | M′-N′ | N′-O′ | O′-P′ | P′-Q′ | Q′-n |
| | Segment Length | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 4 | 1 | 1 | 3 |
| | Point | | A′ | B′ | C′ | D′ | E′ | F′ | G′ | H′ | I′ | J′ | K′ | L′ | M′ | N′ | O′ | P′ | Q′ | n |
| | Accumulated Length | | -1 | -2 | -3 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 15 | 14 | 13 | 10 |

FIGURE 2.19: A TABLE SUMMARIZING ACCUMULATED LENGTH AT EACH VERTICES OF ORIGINAL AND SIMPLIFIED LINE SHOWN IN FIG.2.13

In Fig. 2.18 projection of points A, B and C are not falling on the simplified line. Thus, the accumulated length at the points A, B and C are stored along with the starting point of the simplified line because they are close to point 1. Similarly, the ending point ‘n’ consists of the accumulated length at points M, N, O, P and Q where the projection of points N, O, P and Q do not fall on the simplified line (Table. 2.4).

TABLE 2.4: INTERPOLATED DISTANCE AT EACH POINT CLICKED ON THE SIMPLIFIED LINE

| POINT CLICKED ON THE SIMPLIFIED LINE | 1 | D′ | E′ | F′ | G′ | H′ | I′ | J′ | K′ | L′ | M′ | n |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| INTERPOLATED DISTANCE | 0.0<br>1.4<br>2.4<br>3.8 | 7.9 | 8.95 | 10.3 | 11.3 | 12.8 | 13.8 | 14.8 | 16.1 | 17.1 | 20.0 | 29.6<br>20.0<br>21.4<br>24.0<br>25.4<br>26.4 |

The original line shown in Fig. 2.20 contains 496 points. Some of the points and their orthogonal projection are shown in Fig. 2.21 to 2.25.

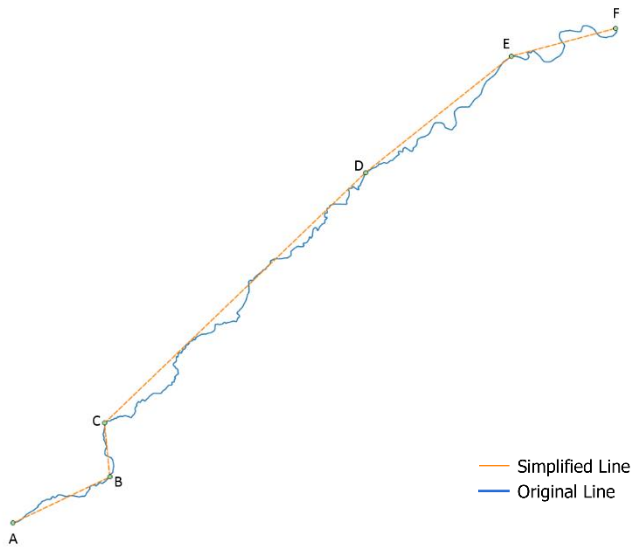


FIGURE 2.20: SIMPLIFIED LINE WITH 5 SEGMENTS AB, BC, CD, DE AND EF

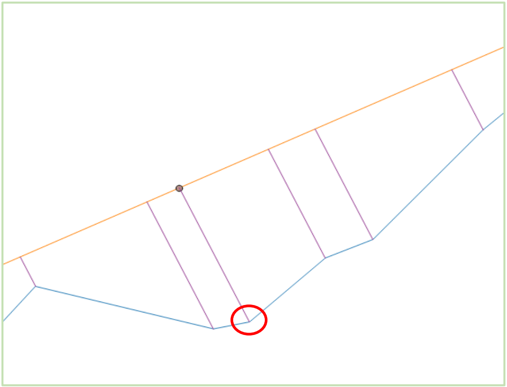


FIGURE 2.21: SAMPLE POINT1

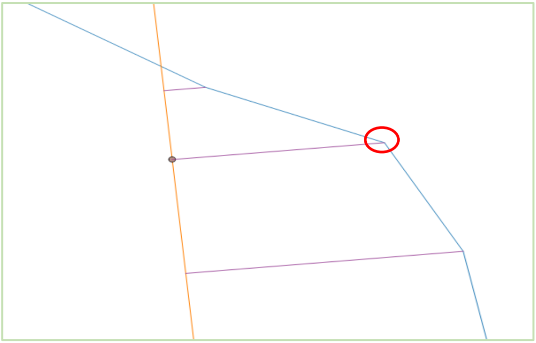


FIGURE 2.22: SAMPLE POINT2

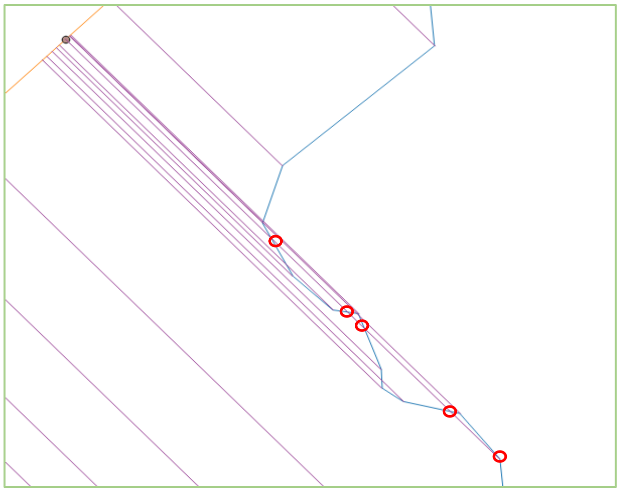


FIGURE 2.23: SAMPLE POINT3

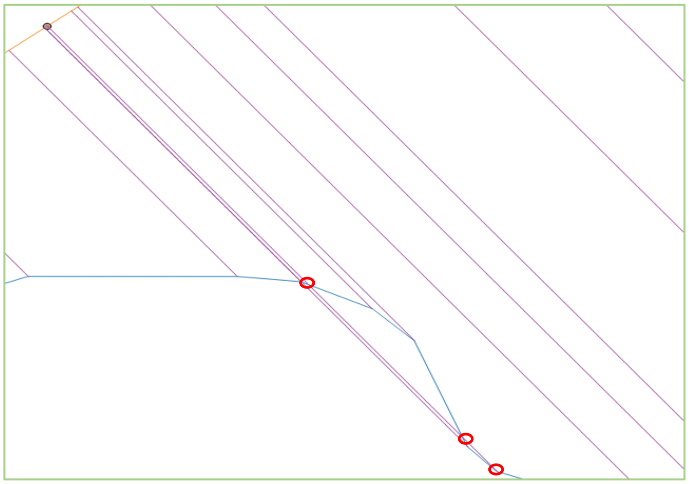


FIGURE 2.24: SAMPLE POINT4

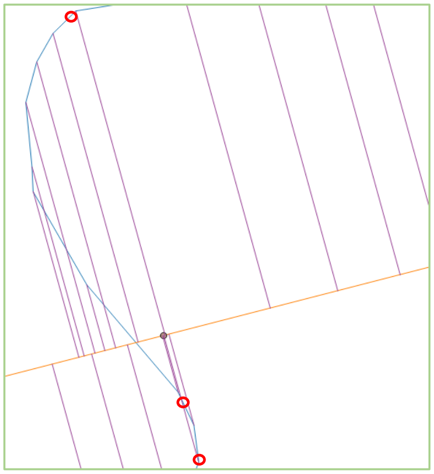
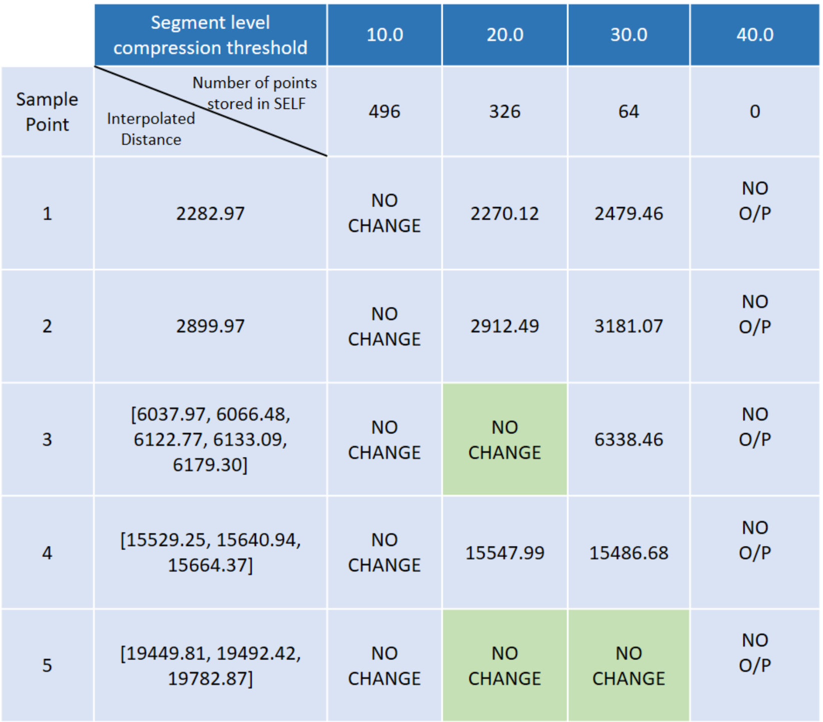


FIGURE 2.25: SAMPLE POINT5

Sample Point1 (Fig. 2.21) and Sample Point2 (Fig. 2.22) correspond to a point on the original line. Sample points 3 (Fig. 2.23), 4 (Fig. 2.24), 5 (Fig. 2.25) correspond to 5, 3, 3 points on the original line accordingly. Table.2.5 lists the interpolated distance at all 5 sample points at different values of segment level compression. At 40.0% (i.e. ratio between original length and simplified length of the segment < 40.0) segment compression level, all the points are lost. Wherever there is “NO CHANGE” the length is interpolated correctly.

TABLE 2.5: INTERPOLATED DISTANCE AT DIFFERENT VALUES FOR SEGMENT LEVEL COMPRESSION THRESHOLD



| | Segment level compression threshold | 10.0 | 20.0 | 30.0 | 40.0 |
|---|---|---|---|---|---|
| Sample Point | Number of points stored in SELF<br>Interpolated Distance | 496 | 326 | 64 | 0 |
| 1 | 2282.97 | NO CHANGE | 2270.12 | 2479.46 | NO O/P |
| 2 | 2899.97 | NO CHANGE | 2912.49 | 3181.07 | NO O/P |
| 3 | [6037.97, 6066.48, 6122.77, 6133.09, 6179.30] | NO CHANGE | NO CHANGE | 6338.46 | NO O/P |
| 4 | [15529.25, 15640.94, 15664.37] | NO CHANGE | 15547.99 | 15486.68 | NO O/P |
| 5 | [19449.81, 19492.42, 19782.87] | NO CHANGE | NO CHANGE | NO CHANGE | NO O/P |

There are three possibilities while running algorithm 2.4 to compute the accumulated length on the original line at any point of the simplified line from SELF structure. Depending on the outcome, the error in interpolation is classified as follows:

1. Interpolated distance is greater than original accumulated length – **Negative error**
2. Interpolated distance is less than original accumulated length – **Positive error**
3. Interpolated distance is equal to original accumulated length – **Zero error**

Similarly, the distance at all 496 points is interpolated by algorithm 2.4. Fig. 2.26 & 2.27 compares the maximum (Positive error), minimum (Negative error) and standard deviation in the interpolated distance at various levels of compression.

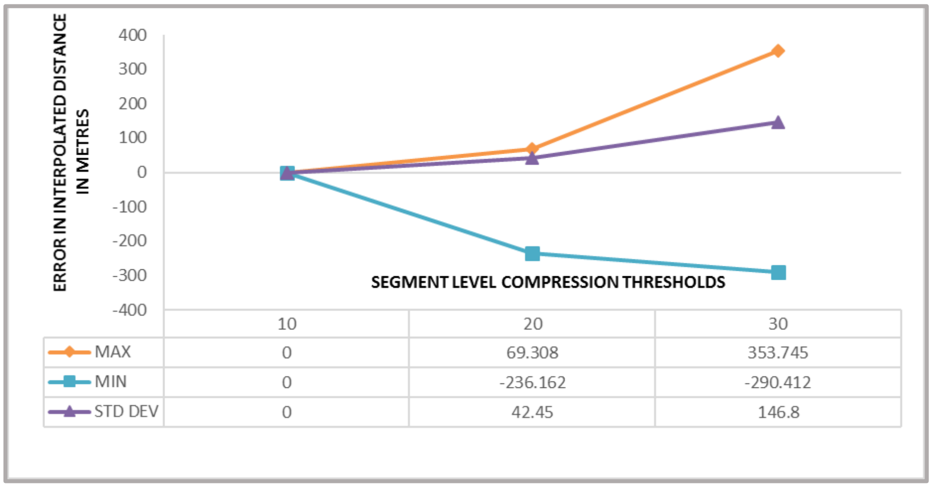


FIGURE 2.26: ERROR IN INTERPOLATED DISTANCE VS SEGMENT LEVEL COMPRESSION VALUES

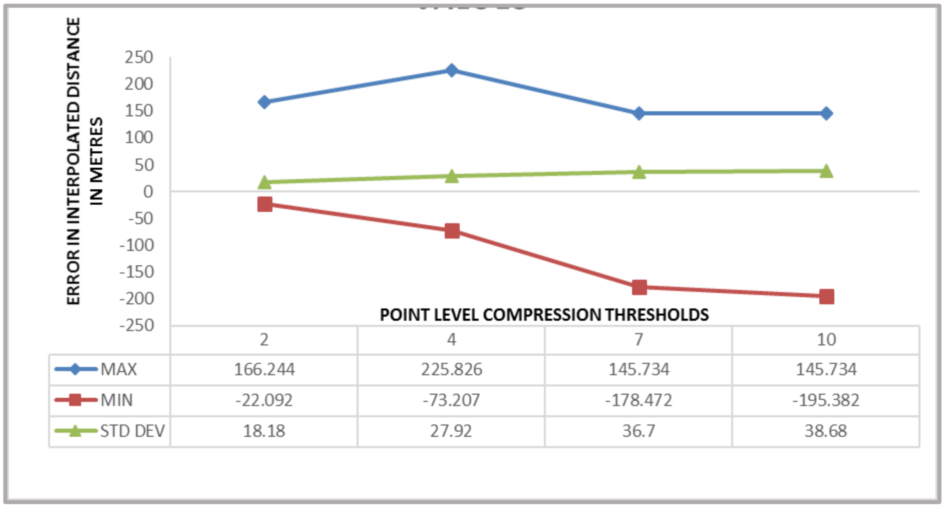
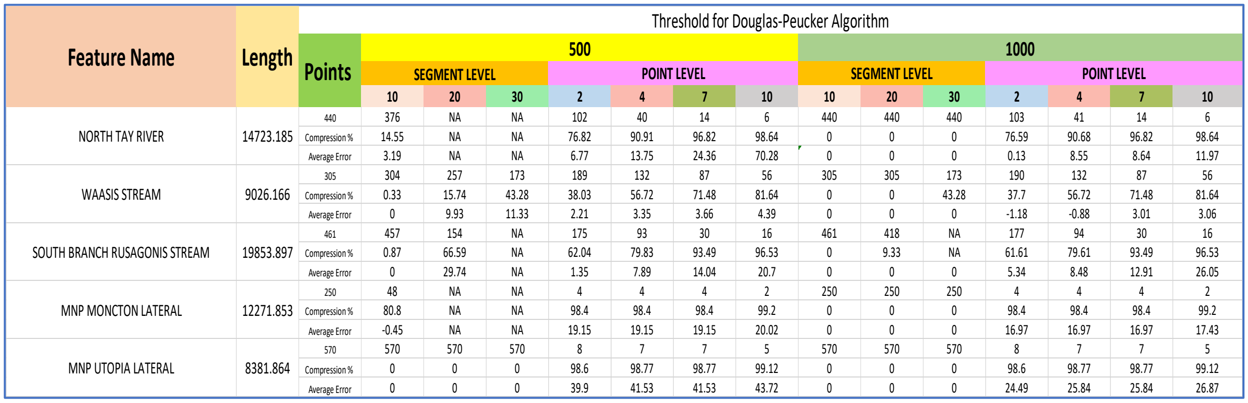


FIGURE 2.27: ERROR IN INTERPOLATED DISTANCE VS POINT LEVEL COMPRESSION VALUES

Segment level compression produces higher positive and negative error than point level compression (Fig. 2.26 & 2.27). As a consequence of segment level compression, the entirety of segments (continuous points) which have the segment level compression ratio within the user-defined threshold are eliminated. This leads to an error in interpolation for the points which belong to the eliminated segment, whereas point level compression discards only certain points which are within the point level threshold.

The distance interpolation algorithm (Algorithm 2.4) has been run with different levels of compression on these datasets (Table. 2.6).

TABLE 2.6: AVERAGE ERROR IN DISTANCE INTERPOLATION USING SELF-STRUCTURE



| Feature Name | Length | Points | Threshold for Douglas-Peucker Algorithm | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 500 | | | | | | | 1000 | | | | | | |
| | | | SEGMENT LEVEL | | | POINT LEVEL | | | | SEGMENT LEVEL | | | POINT LEVEL | | | |
| | | | 10 | 20 | 30 | 2 | 4 | 7 | 10 | 10 | 20 | 30 | 2 | 4 | 7 | 10 |
| NORTH TAY RIVER | 14723.185 | 440 | 376 | NA | NA | 102 | 40 | 14 | 6 | 440 | 440 | 440 | 103 | 41 | 14 | 6 |
| | | Compression % | 14.55 | NA | NA | 76.82 | 90.91 | 96.82 | 98.64 | 0 | 0 | 0 | 76.59 | 90.68 | 96.82 | 98.64 |
| | | Average Error | 3.19 | NA | NA | 6.77 | 13.75 | 24.36 | 70.28 | 0 | 0 | 0 | 0.13 | 8.55 | 8.64 | 11.97 |
| WAASIS STREAM | 9026.166 | 305 | 304 | 257 | 173 | 189 | 132 | 87 | 56 | 305 | 305 | 173 | 190 | 132 | 87 | 56 |
| | | Compression % | 0.33 | 15.74 | 43.28 | 38.03 | 56.72 | 71.48 | 81.64 | 0 | 0 | 43.28 | 37.7 | 56.72 | 71.48 | 81.64 |
| | | Average Error | 0 | 9.93 | 11.33 | 2.21 | 3.35 | 3.66 | 4.39 | 0 | 0 | 0 | -1.18 | -0.88 | 3.01 | 3.06 |
| SOUTH BRANCH RUSAGONIS STREAM | 19853.897 | 461 | 457 | 154 | NA | 175 | 93 | 30 | 16 | 461 | 418 | NA | 177 | 94 | 30 | 16 |
| | | Compression % | 0.87 | 66.59 | NA | 62.04 | 79.83 | 93.49 | 96.53 | 0 | 9.33 | NA | 61.61 | 79.61 | 93.49 | 96.53 |
| | | Average Error | 0 | 29.74 | NA | 1.35 | 7.89 | 14.04 | 20.7 | 0 | 0 | 0 | 5.34 | 8.48 | 12.91 | 26.05 |
| MNP MONCTON LATERAL | 12271.853 | 250 | 48 | NA | NA | 4 | 4 | 4 | 2 | 250 | 250 | 250 | 4 | 4 | 4 | 2 |
| | | Compression % | 80.8 | NA | NA | 98.4 | 98.4 | 98.4 | 99.2 | 0 | 0 | 0 | 98.4 | 98.4 | 98.4 | 99.2 |
| | | Average Error | -0.45 | NA | NA | 19.15 | 19.15 | 19.15 | 20.02 | 0 | 0 | 0 | 16.97 | 16.97 | 16.97 | 17.43 |
| MNP UTOPIA LATERAL | 8381.864 | 570 | 570 | 570 | 570 | 8 | 7 | 7 | 5 | 570 | 570 | 570 | 8 | 7 | 7 | 5 |
| | | Compression % | 0 | 0 | 0 | 98.6 | 98.77 | 98.77 | 99.12 | 0 | 0 | 0 | 98.6 | 98.77 | 98.77 | 99.12 |
| | | Average Error | 0 | 0 | 0 | 39.9 | 41.53 | 41.53 | 43.72 | 0 | 0 | 0 | 24.49 | 25.84 | 25.84 | 26.87 |

It can be seen from Fig. 2.28 that the percentage error increases when the level of compression is increased. Noticeably, average error for “NORTH TAY RIVER” suddenly increases after 7% of point level compression. The increase in compression level will discard more points from SELF structure (Fig. 2.29), though this number would change due to the different topological complexity of the datasets. Consequently, the level of compression can be decided based on the application and the required accuracy in distance interpolation.

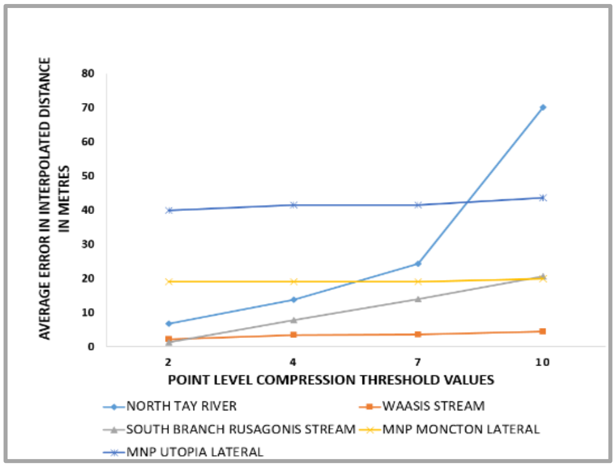


FIGURE 2.28: AVERAGE ERROR IN INTERPOLATED DISTANCE VS POINT LEVEL COMPRESSION VALUES

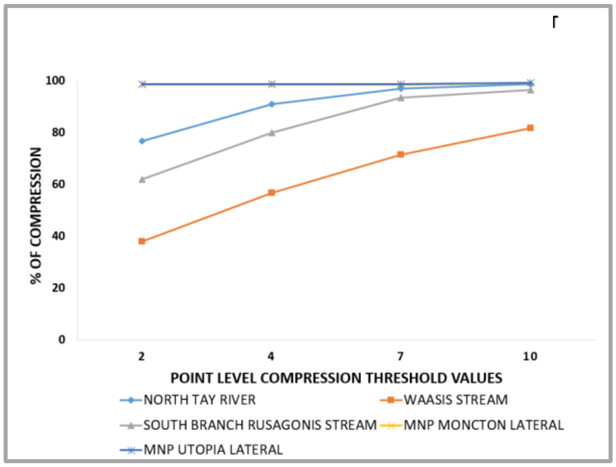


FIGURE 2.29 : % OF COMPRESSION VS POINT LEVEL COMPRESSION VALUES

## 2.5 Conclusions

This paper involved implementing the SELF (Semantically Enriched Line simpliFication) data structure to preserve the geometric characteristics associated to the actual linear features. Currently, the data structure has been implemented in PostgreSQL 9.4 with PostGIS extension using PL/pgSQL to support static and non-functional polylines and tested with both synthetic and real world features.

The algorithm applies two kinds of compression: point level and segment level. The segment level compression eliminates entire segments (continuous points) which has the segment level compression ratio within the user-defined threshold, while point level compression discards only certain points which are within the point level threshold. However, the results of the experiments indicate that the different topological complexity of the datasets play a major role in distance interpolation error.

Future work includes the implementation of the SELF structure extension to support spatio temporal lines. This will result in an enriched library of PL/pgSQL function to support the simplification of both static and dynamic lines.

Recoding this library to other programming languages (such as Python) so that it can be embedded into other commercial or open source GIS software packages is another future goal. Lastly, special attention will be given in developing a framework to facilitate the adoption of the SELF structure in various application domains with need for semantically enhanced multiscale representation of linear features. Applications may need to retain the accumulated length of the road, positional velocity, speed limit or accumulated gas consumption in the road network. In a hydrographic network, the river depth or width can be expressed using the SELF structure.

**Acknowledgements**

This research has received funding from the Natural Sciences and Engineering Research Council of Canada (NSERC), Discovery Grants Program.

The authors wish to thank the anonymous reviewers and the editor for their valuable comments.

# REFERENCES

Abam, M.A., et al., 2010. Streaming algorithms for line simplification. *Discrete & Computational Geometry*, 43 (3): 497–515. doi:10.1007/s00454-008-9132-4

Alvares, L.O., et al., 2007. A model for enriching trajectories with semantic geographical information. In: *The Proceedings of the 15th annual ACM international symposium on advances in geographic information systems*, 7–9 November, Seattle, WA. Article No. 22.

Cromley, R.G., 1991. Hierarchical methods of line simplification. *Cartography and Geographic Information Science*, 18 (2): 125–131. doi:10.1559/152304091783805563

Douglas, D.H. and Peucker, T.K., 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer*, 10 (2): 112–122. doi:10.3138/FM57-6770-U75U-7727

GeoNB Data Catalogue, Retrieved from http://www.snb.ca/geonb1/e/DC/catalogue-E.asp Published on: March 11th 2014, Retrieved on : Feb 6th 2017

Parent, C. et al., 2013. Semantic Trajectories Modeling and Analysis. *ACM Computing Surveys, Vol. 45, No.4, Article 42*. doi: http://dx.doi.org/10.1145/2501654.2501656

PostGIS Reference. Chapter 8. Retrieved from http://postgis.net/docs/reference.html#Management_Functions, Published on: 26th September 2016, Accessed on: 5th February 2017

PostgreSQL 9.4.11 Documentation Chapter 40. PL/pgSQL - SQL Procedural Language. Retrieved from https://www.postgresql.org/docs/9.4/static/plpgsql-statements.html, Published on: 18th December 2014, Accessed on : 5th February 2017

QiuLei Guo and Hassan A. Karimi, 2016. A Topology-Inferred Graph-Based Heuristic Algorithm for Map Simplification. In: *Transactions in GIS,* doi:10.1111/tgis.12188

Richter, K.F., Schmid, F., and Laube, P., 2012. Semantic trajectory compression: representing urban movement in a nutshell. *Journal of Spatial Information Science*, (4): 3–30.

Robinson, Joel L. Morrison, Phillip C. Muehrcke, A. Jon Kimerling, Stephen C. Guptill, Elements of Cartography, 6th Edition ISBN: 978-0-471-55579-7

Shahriari, N., and V. Tao, 2002. Minimizing Positional Errors in Line Simplification Using Adaptive Tolerance Values. In: *Symposium on Geospatial Theory, Processing and Application,* 4(3), 213-223.

Spaccapietra, S., et al., 2008. A conceptual view on trajectories. *Data & Knowledge Engineering,* 65 (1): 126–146. Retrieved from: http://www.sciencedirect.com/science/article/pii/S0169023X07002078

Stefanakis, E., 2015. SELF: Semantically enriched Line simpliFication. In: *International Journal of Geographical Information Science*, Vol. 29, Iss. 10, 2015, Pages 1826-1844doi: 10.1080/13658816.2015.1053092

Tienaah, T., Stefanakis, E., and Coleman, D., 2015. Contextual Douglas-Peucker simplification. Geomatica Journal, 69 (3), 327-338, https://doi.org/10.5623/cig2015-306 .

Weibel, R., 1996. A typology of constraints to line simplification. In: Proceedings of 7th international symposium on spatial data handling, 12–16 August, Delft. IGU: 533–546.

Weibel, R., 1997. Generalization of spatial data: principles and selected algorithms. In: M. Kreveld, et al., eds. Algorithmic foundations of geographic information systems. Berlin: Springer: 99–152.

Wu,Shil – Ting and Mercedes Rocio Gonzales Marquez 2003. Proceedings of the XVI Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI'03). 1530-1834/03 doi :10.1109/SIBGRA.2003.1240992.

Yan, Z., et al. 2011. SeMiTri: A framework for semantic annotation of heterogeneous trajectories. In: The proceedings of the 14th international conference on Extending Database Technology (EDBT 2011). New York: ACM: 259–270.

# 3. Semantically enriched simplification of trajectories

## Abstract

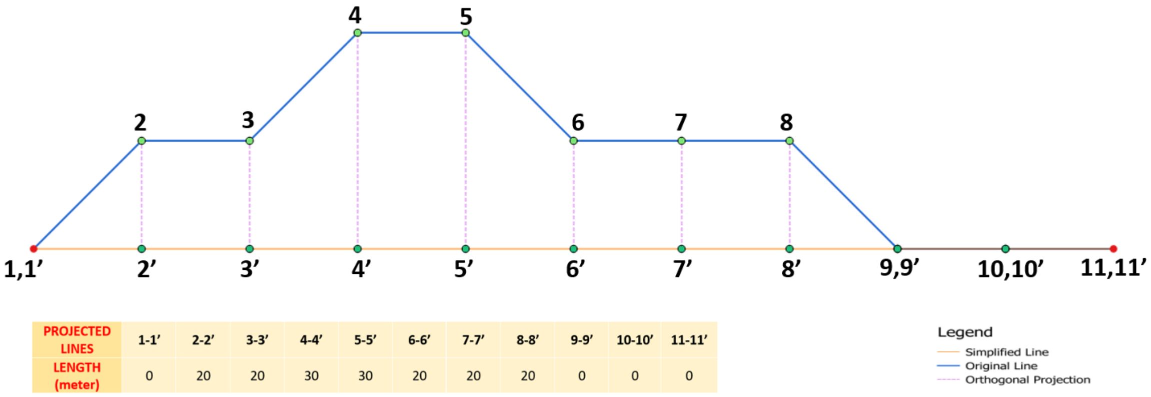
Moving objects that are equipped with GPS devices generate huge volumes of spatio-temporal data. This spatial and temporal information is used in tracing the path travelled by the object, so called trajectory. It is often difficult to handle this massive data as it contains millions of raw data points. The number of points in a trajectory is reduced by trajectory simplification techniques. While most of the simplification algorithms use the distance offset as a criterion to eliminate the redundant points, temporal dimension in trajectories should also be considered in retaining the points which convey both the spatial and temporal characteristics of the trajectory. In addition to that the simplification process may result in losing the semantics associated with the intermediate points on the original trajectories. These intermediate points can contain attributes or characteristics depending on the application domain.  For example, a trajectory of a moving vessel can contain information about distance travelled, bearing, and current speed. This paper presents the implementation of the Synchronous Euclidean Distance (SED) based simplification to consider the temporal dimension and building the Semantically Enriched Line simpliFication (SELF) data structure to preserve the semantic attributes associated to individual points on actual trajectories. The SED based simplification technique and the SELF data structure have been implemented in PostgreSQL *9.4* with PostGIS extension using PL/pgSQL to support dynamic lines. Extended experimental work has been carried out to better understand the impact of SED based simplification over conventional Douglas-Peucker algorithm to both synthetic and real trajectories.  The efficiency of SELF

structure in regard to semantic preservation has been tested at different levels of simplification.

## 3.1 Introduction

Over the years, technological advancements have enabled the usage of GPS devices in moving objects. These devices generate streams of points (locations) which form a path travelled by the moving object during a particular period of time. This traced path is known as trajectory. Trajectory data is commonly utilized in urban planning, fleet management systems, and other location-based service applications. With every trajectory containing enormous amount of data points, it is often required to reduce the data according to the application domain. The concept of trajectory reduction has evolved from the algorithms used in cartographic generalization for linear geometric features also known as simplification [*Keates* 1989]. The basic idea is to retain certain points which are more significant in forming the trajectory than other points as they better convey the trajectory characteristics for a particular context. For example, the point at which a sudden speed change occurs is more important than other points in vehicle movement tracking. The conventional generalization techniques for linear features (e.g., rivers, pipelines, and roads) remove the high-density vertices based on a given criterion.

The Douglas-Peucker (DP) algorithm is a recursive approach for simplifying lines which takes the original linear geometry and a threshold distance as input. The simplified version is generated by controlling the offset while minimizing the distortion. At the end of a recursive process, only a subset of the vertices is retained to form the simplified geometry. The resultant geometry ends up in reduction in length [*Douglas et al.* 1973]. DP simplification algorithm does not consider temporal dimension (time) associated with the vertices of the trajectories. Furthermore, as a result of simplification the semantics (e.g. speed, heading and distance travelled) at each point on the original line (trajectory) are not preserved in the simplified line.  As a result, Douglas-Peucker algorithm has limited scope to be utilized in trajectory simplification. For example, in Fig. 3.1, only the first and last points of original line are retained in the simplified line for a 50-meter threshold distance, because none of the perpendicular offset is greater than 50 meters regardless the temporal data associated with the intermediate points. Depending on the threshold distance some intermediate points can also be retained using the Douglas-Peucker algorithm**.** As DP algorithm has the limitation of not being able to consider the temporal dimension of a trajectory, the notion of the Synchronous Euclidean Distance (SED) was introduced by *Meratnia and de By*.



| PROJECTED LINES | 1-1' | 2-2' | 3-3' | 4-4' | 5-5' | 6-6' | 7-7' | 8-8' | 9-9' | 10-10' | 11-11' |
|---|---|---|---|---|---|---|---|---|---|---|---|
| LENGTH (meter) | 0 | 20 | 20 | 30 | 30 | 20 | 20 | 20 | 0 | 0 | 0 |

FIGURE 3.1: COMPARISON OF ORIGINAL AND THE SIMPLIFIED VERSION WHEN DP-THRESHOLD IS 50 METERS

This chapter presents an implementation of DP with the notion of SED and combining it with SELF (Semantically Enriched Line simpliFication) data structure for dynamic linear features that preserves the semantic attributes (speed, heading and distance travelled) associated with individual locations of original trajectory. These attributes are associated with the DP-SED based simplified trajectory as an array of values corresponding to multiple locations along the simplified trajectory [*Stefanakis* 2015].

The purpose of this chapter is twofold. First, to involve the temporal dimension in the simplification of trajectories. Second, to retain the semantic (speed, heading and distance travelled) attributes associated with individual locations of original trajectories. The latter has been done by associating the semantic values to the simplified geometry as an array of values corresponding to multiple locations along the simplified geometry [*Stefanakis* 2015].

The overall objectives of this research work are:

1. To implement SED based trajectory simplification technique to consider spatio-temporal data in trajectory generalization
2. To implement the SELF structure to support dynamic polylines and to test with both synthetic and real world features.
3. To compare the interpolated semantic values using SELF structure at different levels of trajectory generalization

The chapter is organized as follows. Section 3.2 provides a literature review about trajectory simplification and briefly describes the SED based simplification and SELF structure for dynamic linear features. Section 3.3 presents the steps followed to implement SED simplification technique and build the SELF structure in PostgreSQL/PostGIS. Section 3.4 presents PostGIS functionality and explanations of the experiments with various real-world trajectories. Section 3.5 summarizes the contribution of this paper and introduces future developments for the SELF data structure with respect to testing with various application domains.

## 3.2 Literture Review

### *3.2.1 Trajectory Generalization*

Over the years the usage of GPS devices in moving vehicles have increased exponentially and massive amount of data is being generated by these devices. The generated data is used in various public and business applications such as urban transportation planning, fleet management and traffic modelling [*K. Buchin et al.* 2008]. The enormous volume of data makes it impossible to analyze the data manually. For example, during the trip length of 30 minutes, if the location is being recorded for every 5 seconds a total of 360 points are recorded. In a day, the dataset contains 17,280 points. It necessitates to identify the methods for reducing the complexity of the dataset. The concept of reducing a trajectory dataset is called trajectory reduction or simplification. The idea of trajectory reduction has evolved from cartographic generalization. Simplification, the common cartographic generalization technique, has been a key research area for cartographers over the years [*Cromley* 1991, *Weibel* 1997, *Robinson et al.* 2005, *Wu et al.* 2003].

The Douglas-Peucker (DP) algorithm has been revamped by many researchers since it was introduced in 1973. The problem of topological inconsistency between original and simplified geometry produced by DP algorithm was addressed by avoiding self-intersections on the simplified geometry [*Wu et al.* 2003]. The problem of limited data storage space is addressed by experimenting the line simplification algorithms in a streaming environment [*Abam* 2010]. Various techniques have been proposed to enforce the topological constraints while simplifying a polyline [*Shahriari and Tao* 2002, *Tienaah et al.* 2015, *QiuLei et al.* 2016].

Meanwhile, enriching the content of linear features has gained attention to address the problem of annotating trajectories with semantic data. [*Alvares et al.* 2007, *Yan et al.* 2011, *Richter et al.* 2012, *Parent et al.* 2013, *Stefanakis* 2015]. The trajectory sample points have been transformed into stops and moves by adding semantic information [*Alvares et al.* 2007]. Though the implemented model has shown significant compression of trajectories while enabling efficient query processing, the preprocessing of adding semantic information to trajectories is a time-consuming operation. The semantic enrichment platform SeMiTri, multi-tiered approach, was presented to handle heterogeneous trajectories (includes both fast and slow-moving objects). The trajectory generalization platform based on Hidden Markov Model (HMM) technique has not considered trajectories in large scale [*Yan, Z., et al.* 2011]. *Richter et al.* [2012] extended concepts of network-constrained indexing in mobility object to embed human movement with the individual locations on the trajectory. The algorithm they proposed enables a user to determine the reference point and all possible movement change descriptions from that point but is limited only to urban transport network.

The problem of spatial relation violation while compressing the trajectories was addressed to maintain disjoint topological relation and direction relations between the original and generalized trajectory [*Stefanakis* 2012]. The author has extended DP algorithm to maintain the topological consistency between the trajectory and its simplified version.

### *3.2.2 Synchronous Euclidean Distance (SED)*

Most of the simplification algorithms are suitable for generalizing linear geometries. In these algorithms the data points are retained only based on the perpendicular distance between data points and the proposed generalized version of it. While these algorithms can also be applied on trajectory datasets, using the perpendicular distance as a criterion becomes inappropriate as trajectories are not just linear geometries. Trajectories represent historical trace of points by associating temporal dimension with spatial data. With the above idea, the notion of the Synchronous Euclidean Distance (SED) was introduced to achieve reduction of trajectories while retaining the spatio temporal characteristics of the trajectory [*Meratnia and de By* 2004]. The authors have implemented and tested the DP-SED algorithm (extension of Douglas-Peucker algorithm with the notion of SED). The proposed algorithm retains the spatiotemporal characteristics while reducing the trajectories efficiently. Fig. 3.2 demonstrates how SED is calculated between simplified and original geometry.

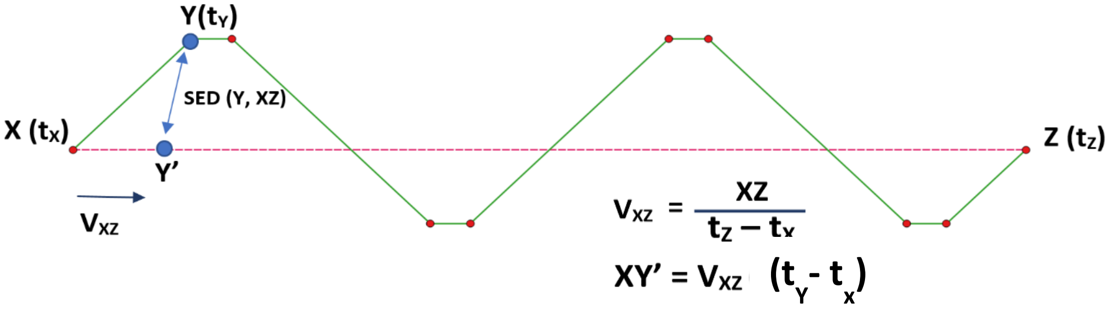


FIGURE 3.2: THE SYNCHRONOUS EUCLIDEAN DISTANCE (SED).

In Fig. 3.2, the locations X, Y, Z represent the position of a moving vessel at the timestamps $t_X$, $t_Y$, $t_Z$ where $t_X < t_Y < t_Z$. The spatiotemporal footprint of **Y** (i.e. **Y′**) is calculated with respect to the velocity of trip $\mathbf{V}_{XZ}$. The Euclidean distance YY′ is known as the SED for the point Y. The perpendicular distance applied in DP algorithm is lesser or equal to the SED (YY′) as the line YY′ is not perpendicular to the straight line XZ.

### *3.2.3 Semantically Enriched Line Simplification (SELF)*

On the one hand, efficient generalization of trajectories can be achieved by DP-SED algorithm while retaining the spatiotemporal characteristics of the trajectory. However, the generalized version does not retain the semantics associated with the individual points on the original trajectory. SELF data structure has been introduced by *Stefanakis* [2015] to enrich the simplified line to convey some semantics associated with the original version. In the attempt of enriching the content of the linear geometries while reducing the number of points, SELF data structure is proposed to preserve the attributes of the original line or any semantic annotations associated with individual locations or segments of that line [*Spaccapietra et al.* 2008] into the generalized version [*Stefanakis* 2015]. The author has proposed many variations of SELF and the choices can be made based on how rich the semantics attached to the simplified line are.

The *basic variant* of SELF attaches the original line length (e.g., kilometric travel distance) to the simplified line. In this variant, a line with end points 1 (start), n (end), and total length ($d_n$) is represented by a simplified line defined as follows [Stefanakis 2015]:

$$[x_1, y_1, x_n, y_n, d_n] \text{ (SELF variant: basic)}$$

An *advanced variant* for function lines also tags the accumulated length per vertex along the line. Hence, each vertex *K* of the original line is orthogonally projected on the simplified line (Fig. 3.1) and the footprint point *K′* is assigned the accumulated length $d_k$ from point *1* (start) to vertex *K* along the original line. If $d_k'$ is the Euclidean distance of point *K′* from end point *1*, the simplified line can be represented as follows [Stefanakis 2015]:

[$x_1$, $y_1$, $x_n$, $y_n$, $d_n$, ARRAY {($d_k'$, $d_k$); k=2, …, n-1}] (SELF variant: advanced-function)

In this paper, the advanced variant of SELF structure has been extended to tag trajectory semantics: speed, heading, and distance travelled. Each point on the original trajectory is projected on the generalized version based on SED. The footprint of each point will be assigned with speed, heading and distance travelled at that point.

[$x_1$, $y_1$, $x_n$, $y_n$, $d_n$, ARRAY {($d_k'$, speed, heading, $d_k$); k=1, …, n}] (SELF variant: dynamic lines).

## 3.3 Methodology

### ***3.3.1 SED Simplification***

Trajectories are formed by connecting a series of raw mobility data points. These individual data points include the spatiotemporal locations (latitude, longitude, time). The simplified line using Douglas-Peucker algorithm with the user defined threshold always considers perpendicular distance as a criterion to eliminate the redundant points.

The goal of the SED based simplification is to also consider temporal dimension of trajectory data while generalizing the trajectory.

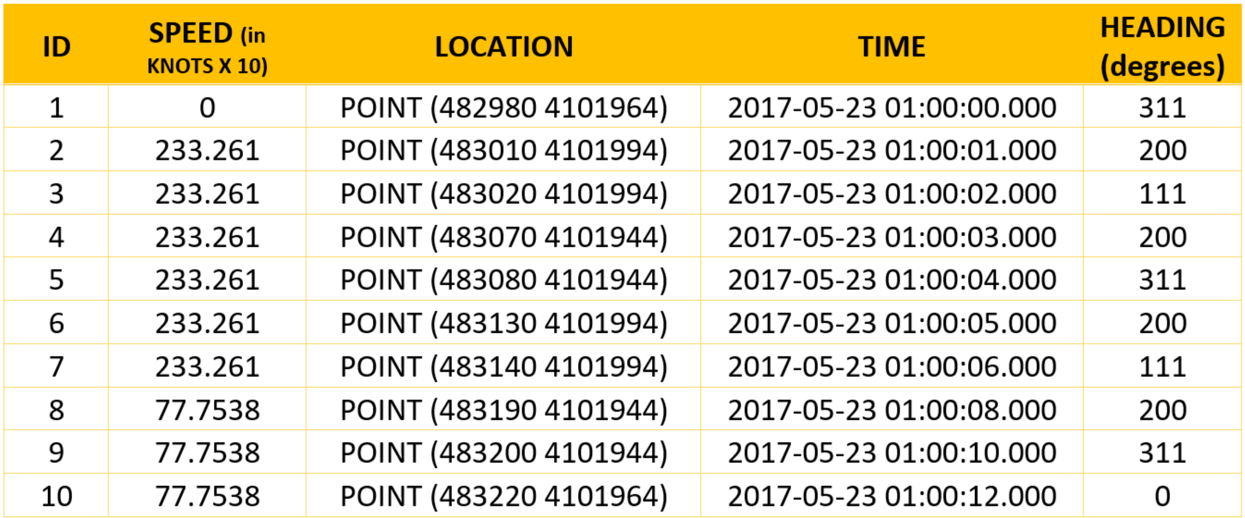
The algorithm is divided into four steps:

1. Constructing the trajectory using the individual points (Trajectory Reconstruction)
2. Calculating the average velocity of the trip
3. Identify the corresponding SED point for every point on the original line
4. Removing the points by comparing the SED against the simplification threshold

#### 3.3.1.1 Constructing the trajectory using the individual points (Trajectory Reconstruction)

Raw points ordered by the timestamp are connected sequentially to form the trajectory. These data points are in the form (time, latitude, longitude). Ten (10) points along with their corresponding attributes are given in Table 3.1; their individual locations are mapped in Figure 3.3 and the constructed trajectory is shown in Figure 3.4.

TABLE 3.1: ATTRIBUTE TABLE FOR THE TRAJECTORY POINTS SHOWN IN FIGURE.3.3



| ID | SPEED (in KNOTS X 10) | LOCATION | TIME | HEADING (degrees) |
|---|---|---|---|---|
| 1 | 0 | POINT (482980 4101964) | 2017-05-23 01:00:00.000 | 311 |
| 2 | 233.261 | POINT (483010 4101994) | 2017-05-23 01:00:01.000 | 200 |
| 3 | 233.261 | POINT (483020 4101994) | 2017-05-23 01:00:02.000 | 111 |
| 4 | 233.261 | POINT (483070 4101944) | 2017-05-23 01:00:03.000 | 200 |
| 5 | 233.261 | POINT (483080 4101944) | 2017-05-23 01:00:04.000 | 311 |
| 6 | 233.261 | POINT (483130 4101994) | 2017-05-23 01:00:05.000 | 200 |
| 7 | 233.261 | POINT (483140 4101994) | 2017-05-23 01:00:06.000 | 111 |
| 8 | 77.7538 | POINT (483190 4101944) | 2017-05-23 01:00:08.000 | 200 |
| 9 | 77.7538 | POINT (483200 4101944) | 2017-05-23 01:00:10.000 | 311 |
| 10 | 77.7538 | POINT (483220 4101964) | 2017-05-23 01:00:12.000 | 0 |

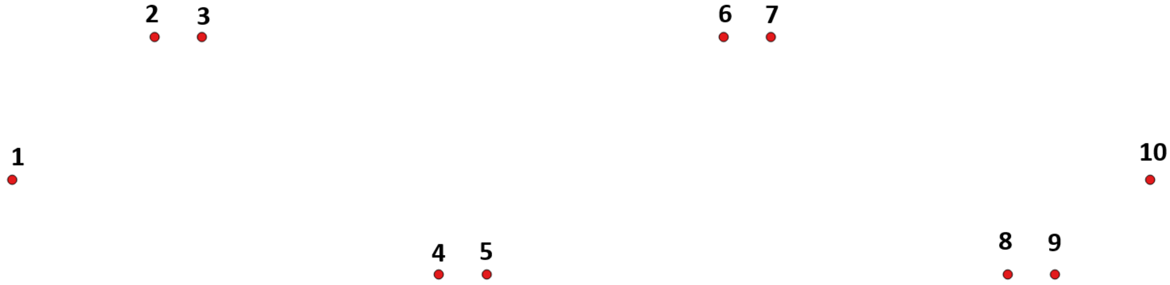


FIGURE 3.3: INDIVIDUAL POINTS ON THE TRAJECTORY

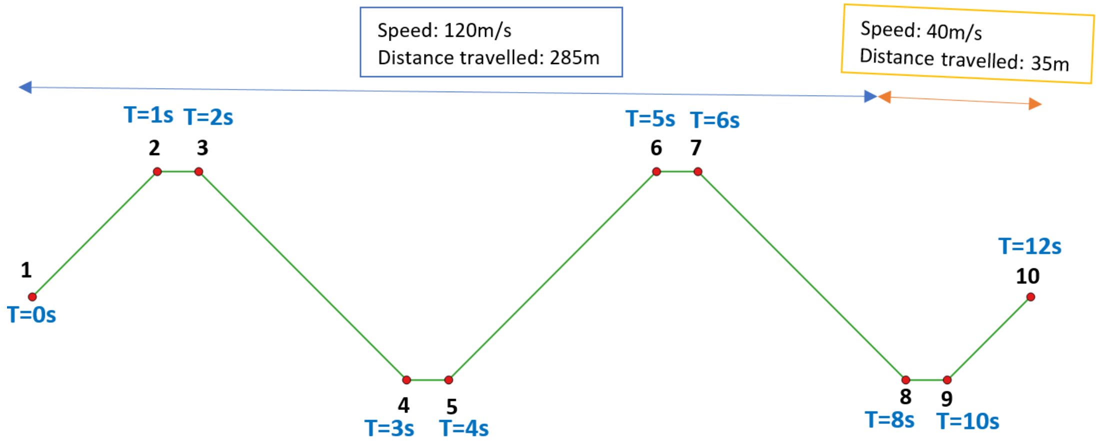


FIGURE 3.4: FORMED TRAJECTORY AFTER CONNECTING INDIVIDUAL POINTS

### 3.3.1.2 Calculating the average velocity of the trip

Average velocity of the trip is defined as the ratio between the straight-line length of the trip and the total duration of the trip.

Average velocity = (Straight line distance between starting and ending point)/ (Total duration of the trip)

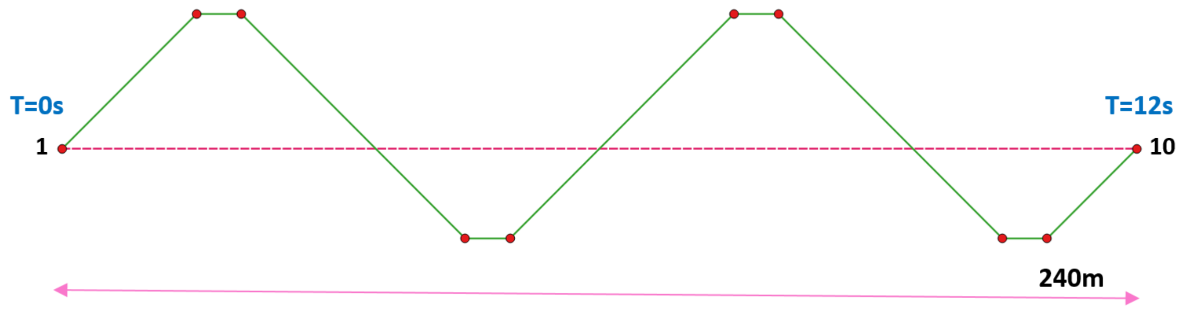
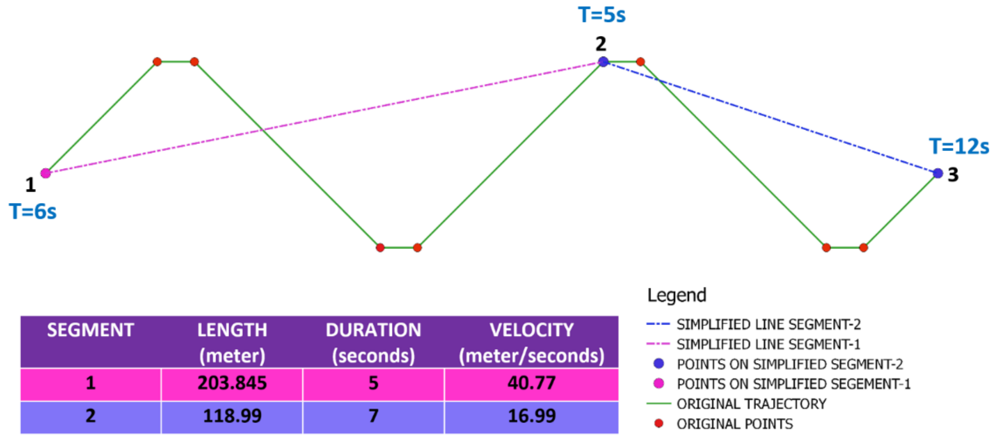


FIGURE 3.5: STRAIGHT LINE CONNECTING STARTING AND ENDING POINTS OF TRAJECTORY

In Fig. 3.5, the average velocity is 20 meters/second. In case of multiple segments connecting the starting and ending points, the average velocity is calculated for each segment and retrieved as an array of numeric values (Fig. 3.6).


T=5s
2
T=12s
3
1
T=6s
Legend
SIMPLIFIED LINE SEGMENT-2
SIMPLIFIED LINE SEGMENT-1
POINTS ON SIMPLIFIED SEGMENT-2
POINTS ON SIMPLIFIED SEGEMENT-1
ORIGINAL TRAJECTORY
ORIGINAL POINTS

| SEGMENT | LENGTH (meter) | DURATION (seconds) | VELOCITY (meter/seconds) |
|---|---|---|---|
| 1 | 203.845 | 5 | 40.77 |
| 2 | 118.99 | 7 | 16.99 |

FIGURE 3.6: AVERAGE VELOCITY CALCULATION FOR MULTI SEGMENT LINE

**Algorithm 3.1:** Finding the average velocity of the trip (Fig. 3.5)

```
Input:
    1. Raw mobility points ordered by timestamp
Output:
    1. Average velocity as a numeric value
Steps:
    1. Retrieve the starting and ending points on trip
    2. Calculate the straight-line distance between starting and ending
       points of the trip(d)
    3. Find the duration of trip(t) = ending time – starting time
    4. Average Velocity = d/t
    5. RETURN the average velocity
```

### 3.3.1.3 Identify the corresponding SED point for every point on the original line

For each point on the original line the corresponding SED point is identified by calculating the distance from starting point to the SED point.

Distance to SED point = Average Velocity * Time of travel at the original point

TABLE 3.2: DISTANCE TO EACH SED POINT ON STRAIGHT LINE FROM STARTING POINT (FIG. 3.7)

| Point ID | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| SED Points | 1 | 2′ | 3′ | 4′ | 5′ | 6′ | 7′ | 8′ | 9′ | 10 |
| Time (Sec) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 10 | 12 |
| Distance to SED point (metre) | 0 | 20 | 40 | 60 | 80 | 100 | 120 | 160 | 200 | 240 |

Based on the distance to SED point, for each point algorithm 3.3 is executed to find the location of the SED points.

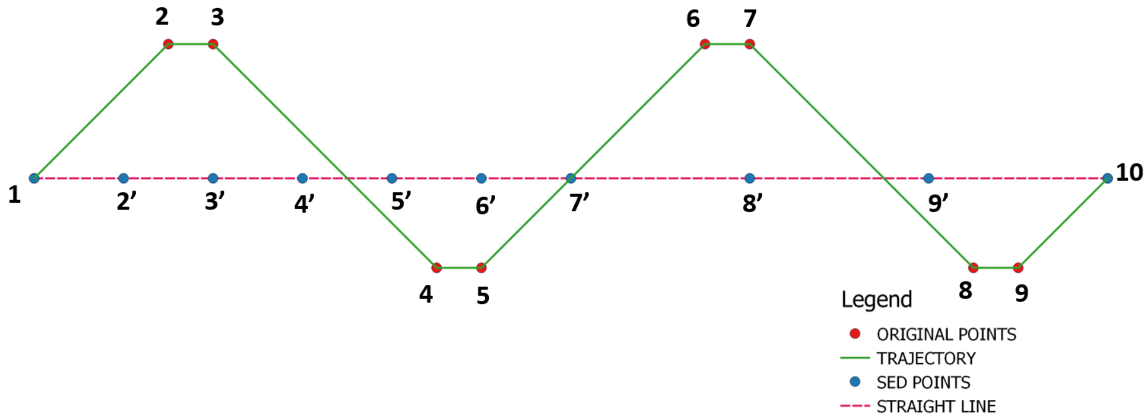


FIGURE 3.7: INDIVIDUAL POINTS ON THE TRAJECTORY AND THE CORRESPONDING SED POINTS ON THE STRAIGHT LINE

**Algorithm 3.2:** Calculating distance to each SED point from starting point on straight line

```
Input:
    1. Raw mobility points ordered by timestamp
Output:
    1. Distance to each point as an array of numeric value
Steps:
    1. Define the numeric array (SED_DIST) to store the distance values
    2. Calculate the average velocity of the trip (Algorithm 1)
    3. FOR EACH point 'P' in the input data
        a. Find the time difference between P and starting point
        b. Multiply the time difference and average velocity of the trip
        c. Add the multiplied value to SED_DIST
    4. END
    5. RETURN the array SED_DIST
```

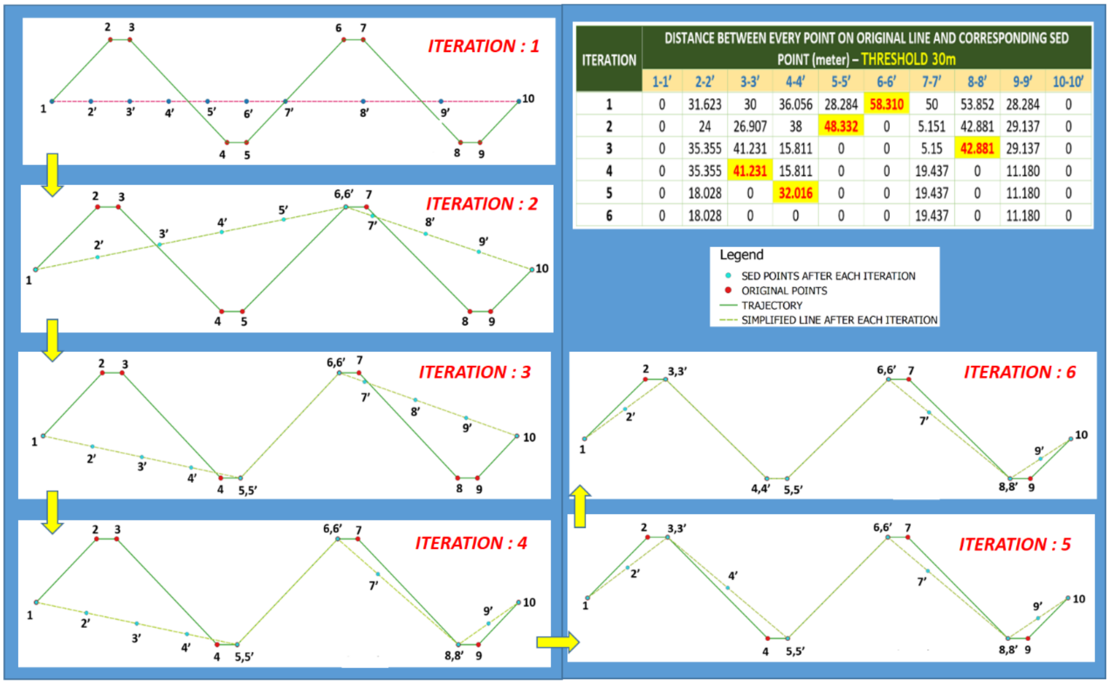
**Algorithm 3.3: Finding the point on the line based on the distance from starting point**

```
Input:
    1. Starting point (SP)
    2. Ending point (EP)
    3. Distance from starting point ($d_P$)
Output:
    1. Point on the straight line returned as geometry type (C)
Steps:
    1. Calculate the distance between SP and EP (d)
    2. Find the ratio(r) between d and $d_P$
    3. The X and Y coordinate of point 'C' can be obtained by,
       X(C) = (1-r) * X(SP) + r * X(EP);
       Y(C) = (1-r) * Y(SP) + r * Y(EP);
    4. RETURN the point 'C'
```

### 3.3.1.4 Removing the points by comparing SED distance against simplification threshold

SED based simplification algorithm takes the set of points ordered by timestamp and a threshold distance as input. The recursive algorithm divides the trajectory based on the SED against the threshold distance. Once the corresponding locations of all the points on the original trajectory are identified on the straight line connecting the first to the last point, the algorithm finds all points for which the SED is longer than threshold distance.

The point with longest SED is marked to be retained for the next iteration. For the next iteration, two straight line segments are compared against original trajectory. When there are no points found with an SED longer that the threshold, the algorithm terminates.  Fig. 3.8 demonstrates the SED based simplification algorithm in generalizing a trajectory. In the iteration 1, for each point on the original trajectory its corresponding SED projection on the straight line connecting 1 and 10 is found. Point 6 has the maximum SED and greater than the threshold (30 meters). So, point 6 is retained. For the next iteration, the intermediate simplified line contains two segments 1-6 and 6-10. Again, for each original point its corresponding SED projection point is found on the intermediate simplified version. This time point 5 has the maximum SED (>30 meters). At the end of second iteration the intermediate simplified line contains three segments 1-5, 5-6, 6-10.  The iteration continues until there are no points to be retained. In this case, the algorithm terminates in the 6th iteration as then none of the points have an SED greater than the threshold (30 meters).



| ITERATION | DISTANCE BETWEEN EVERY POINT ON ORIGINAL LINE AND CORRESPONDING SED POINT (meter) – THRESHOLD 30m | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1-1′ | 2-2′ | 3-3′ | 4-4′ | 5-5′ | 6-6′ | 7-7′ | 8-8′ | 9-9′ | 10-10′ |
| 1 | 0 | 31.623 | 30 | 36.056 | 28.284 | 58.310 | 50 | 53.852 | 28.284 | 0 |
| 2 | 0 | 24 | 26.907 | 38 | 48.332 | 0 | 5.151 | 42.881 | 29.137 | 0 |
| 3 | 0 | 35.355 | 41.231 | 15.811 | 0 | 0 | 5.15 | 42.881 | 29.137 | 0 |
| 4 | 0 | 35.355 | 41.231 | 15.811 | 0 | 0 | 19.437 | 0 | 11.180 | 0 |
| 5 | 0 | 18.028 | 0 | 32.016 | 0 | 0 | 19.437 | 0 | 11.180 | 0 |
| 6 | 0 | 18.028 | 0 | 0 | 0 | 0 | 19.437 | 0 | 11.180 | 0 |

FIGURE 3.8: THE SED BASED SIMPLIFICATION ALGORITHM

**Algorithm 3.4: SED based simplification**

```
Input:
    1. Raw mobility points ordered by timestamp
    2. Simplification threshold (T)
Output:
    1. Retained points after simplification as an array (SP) of geometry type
Steps:
    1. Define the geometry array for storing the retained points (SP)
    2. Construct the trajectory by joining all the raw data points in the order of timestamp
    3. Add first and last points of trajectory to the array SP
    4. Form the straight line (L) by joining starting point and ending point of the trajectory
    5. FOR EACH point ‘P’ in the raw mobility data
        a. Find it’s corresponding SED point (P’) on the line L
        b. Find the distance($d_P$) between P and P’
        c. IF $d_P > T$ AND $d_P$ is the furthest distance THEN
            i. Add the point ‘P’ to the geometry array SP
        d. END
        e. EXIT WHEN no more points with furthest distance greater than T
    6. END
    7. Form the intermediate simplified line based on the retained points and assign it to L
    8. Recursive call to the loop
    9. RETURN the geometry array(SP)
```

### *3.3.2 Building SELF structure based on SED simplification*

The goal of the SELF structure is to retain the semantics at each point on the original trajectory along with the corresponding SED point on the generalized version.

The algorithm is divided into four steps.

1. Finding the SED projection for each point on the original trajectory on the simplified trajectory
2. Calculating the accumulated distance at each point on the original trajectory and its SED projection point on the generalized trajectory
3. Remove the individual points based on the change in speed and heading
4. Interpolation of the semantics on the original trajectory at any point on the generalized version

#### 3.3.2.1 Finding the SED projection for each point on the original trajectory on the simplified trajectory

The algorithm takes as input the generalized trajectory obtained as an output of algorithm 3.4. Then each point on the original trajectory is projected on the generalized version based on SED.

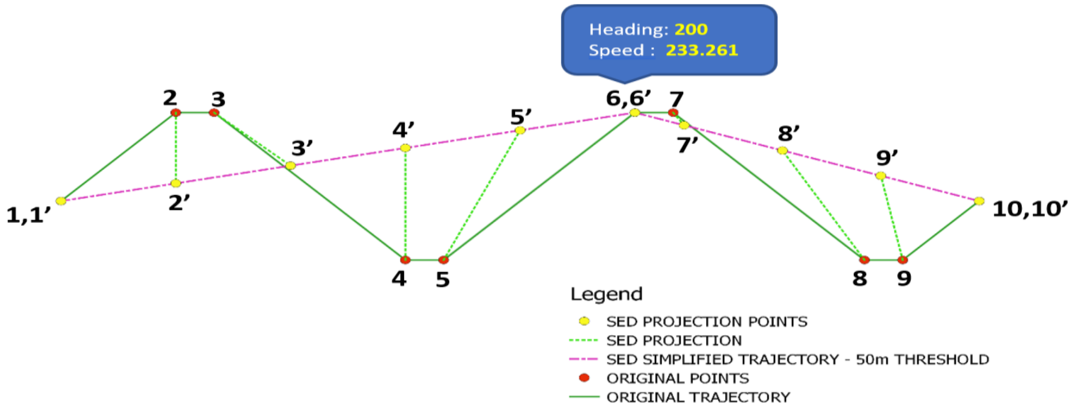


FIGURE 3.9: SED PROJECTION OF EACH POINT ON THE SIMPLIFIED TRAJECTORY

**Algorithm 3.5: Finding the SED projection for each point on the original trajectory on the generalized version**

```
Input:
    1. Geometry of the generalized version
    2. Constructed original trajectory by connecting raw data points (ordered by timestamp)

Output:
    1. SED projection points as an array (SP) of geometry type
Steps:
    1. Define the geometry array for storing the SED projection points (SP)
    2. FOR EACH segment (S) in the geometry of generalized version
        a. Calculate the average velocity (V) over the segment S using algorithm 1
        b. FOR EACH point (P) in the segment 'S'
                i. Find the time difference (T) between P and starting point of the segment
               ii. Multiply T and V to get the distance to SED point from starting point of
                   the segment
              iii. Use multiplied value to find the point(P') on the segment by algorithm 3
               iv. Add the point P' to the array SP
        c. END
    3. END
    4. RETURN the array SP
```

#### 3.3.2.2 Calculating the accumulated distance at each point on the original trajectory and its SED projection point on the generalized trajectory

SELF structure for dynamic lines stores the semantics associated with individual points on the original trajectory along the generalized version. As shown in Fig. 3.9, each point on the original trajectory is projected based on the SED on the generalized version.

For each point on the original trajectory, the corresponding SED point is tagged with the semantics. The entire SELF structure is represented as follows:

[ **SPOINT (482980 4101964), EPOINT (483220 4101964), 322.843**,  
**{(0.000,0,311,0.00),**  
**(30.594,233.261,200,42.426), (61.188,233.261,111,52.426),**  
**(91.782,233.261,200,123.137),**  
**(122.376,233.261,311,133.137), (152.971,233.261,200,203.848),**  
**(166.523,233.261,111,213.848), (193.628,77.7538,200,284.559),**  
**(220.734,77.7538,311,294.559), (247.839,77.7538,0,322.843)}**]

whereas, the semantic array contains the values in the order (accumulated length on the generalized version, speed, heading, accumulated length on the original trajectory).

TABLE 3.3: TABLE SUMMARIZING SEMANTICS AT EACH LOCATIONS OF ORIGINAL AND GENERALIZED TRAJECTORY IN FIG. 3.8

| POINT ON ORIGINAL TRAJECTORY | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| ACCUMULATED DISTANCE ON ORIGINAL TRAJECTORY (m) | 0.0 | 42.43 | 52.43 | 123.14 | 133.14 | 203.85 | 213.85 | 284.60 | 294.60 | 322.84 |
| CORRESPONDING POINT ON SIMPLIFIED TRAJECTORY | 1′ | 2′ | 3′ | 4′ | 5′ | 6′ | 7′ | 8′ | 9′ | 10′ |
| ACCUMULATED DISTANCE ON SIMPLIFIED TRAJECTORY (m) | 0.0 | 30.59 | 61.19 | 91.78 | 122.38 | 152.97 | 166.52 | 193.63 | 220.73 | 247.84 |
| HEADING (degrees) | 311 | 200 | 111 | 200 | 311 | 200 | 111 | 200 | 311 | 0 |
| SPEED (in KNOTS X 10) | 0 | 233.26 | 233.26 | 233.26 | 233.26 | 233.26 | 233.26 | 77.75 | 77.75 | 77.75 |

**Algorithm 3.6:** Building the SELF structure

```
Input:
    1. Raw mobility data points
    2. Threshold value for running SED based simplification (algorithm 3.4)
Output:
    1. SELF advanced structure
Steps:
    1. Define the object of SELF structure (SELF)
    2. Construct the original trajectory by ordering the points based on timestamp
    3. Add the starting point, ending point and the actual length of original trajectory to
       SELF
    4. Define the array for storing the accumulated length and other semantics (AL)
    5. Run SED based simplification algorithm for getting the generalized version
    6. Find the number of segments in the generalized version
    7. FOR EACH segment(S) in the simplified line
    8.     FOR EACH point 'P' on the Segment S
    9.         Find the SED projection point on the generalized version (P')
    10.        Calculate the accumulate length at the point P(Lp) on the original trajectory
    11.        Calculate the accumulated length at the point P'(Lp') on the simplified line
    12.        Add Lp, Lp' and other semantics (heading, speed) to the array AL
    13.    END
    14. END
    15. Add the array AL to SELF
    16. RETURN the SELF structure (SELF)
```

#### 3.3.2.3 Semantic based Compression Levels

SELF structure generates a large volume of data which is proportional to the number of points in the original trajectory. The number of points to be stored in SELF structure can be diminished by applying a semantic based compression: (a) Speed based compression (b) Heading based compression. These methods are described in following sub-sections.

##### 3.3.2.3.1 Speed based compression

If the ratio of speed between two consecutive points on the original trajectory is less than the given threshold, then the semantics at second point is not stored.

$$\text{Speed based compression ratio} = \frac{\textbf{Speed at point `6' -- Speed at point `7'}}{\textbf{Speed at point `6'}} \text{ X } 100\ \%$$

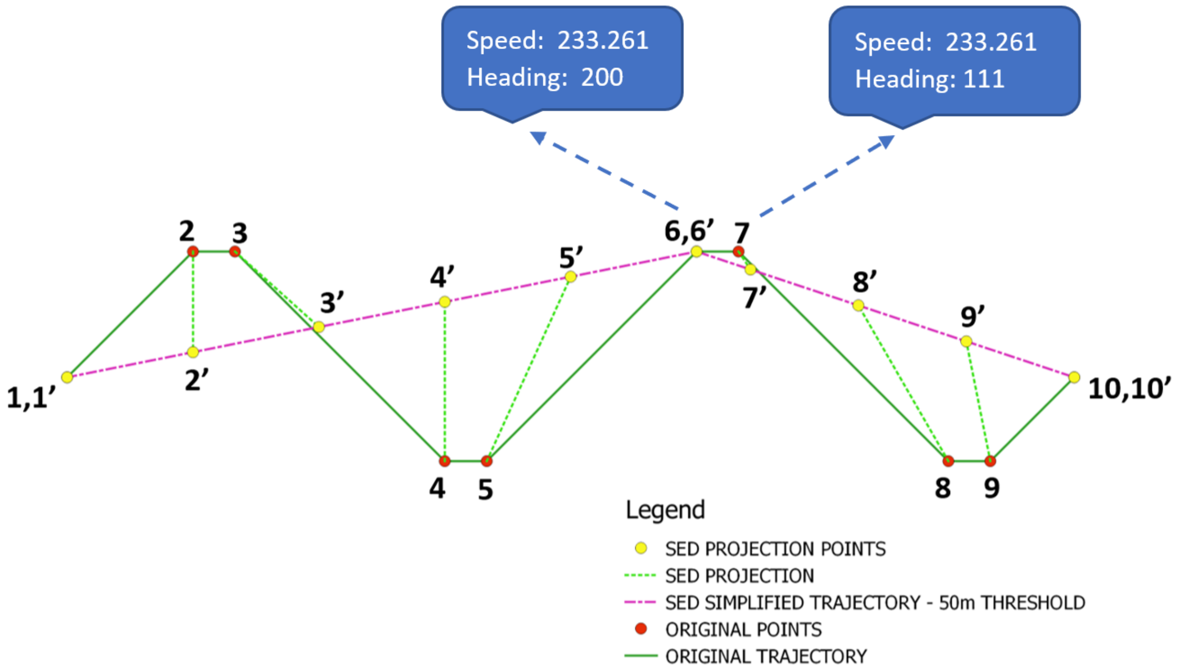


FIGURE 3.10: SED PROJECTION OF EACH POINT ON THE SIMPLIFIED TRAJECTORY AND THE SEMANTICS

In Fig. 3.10 the semantic at point 7 is not stored as the ratio of speed between the points 6 and 7 is zero.

#### 3.3.2.3.2 Heading based compression

In case of heading based compression, the semantics of the points are not stored if the ratio of heading between two consecutive points is less than the threshold.

$$\text{Heading based compression ratio} = \frac{\textbf{Heading at point '6'} - \textbf{Heading at point '7'}}{\textbf{Heading at point '6'}} \text{ X } 100\ \%$$

In Fig. 3.10 the heading based compression ratio between points 6 and 7 is 44.5 %. The semantics at point 7 will not be stored in SELF structure when the heading based compression ratio applied as 50.0 %.

The self structure after applying both the speed based and heading based compression thresholds as (10.0, 10.0) for trajectory data given in Table 3.1 is:

TABLE 3.4: SPEED AND HEADING BASED COMPRESSION RATIO BETWEEN POINTS (DATA SOURCE : TABLE 3.1)

| POINTS | 1-2 | 2-3 | 3-4 | 4-5 | 5-6 | 6-7 | 7-8 | 8-9 | 9-10 |
|---|---|---|---|---|---|---|---|---|---|
| SPEED BASED COMPRESSION RATIO | 100.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 66.67 | 0.0 | 0.0 |
| HEADING BASED COMPRESSION RATIO | 35.69 | 44.5 | 80.18 | 55.5 | 35.69 | 44.5 | 80.18 | 55.5 | 100.0 |

[ **SPOINT (482980 4101964), EPOINT (483220 4101964), 322.843**,
**{(0.000,0,311,0.00),**
**(30.594,233.261,200,42.426), ~~(61.188,233.261,111,52.426)~~,**
**~~(91.782,233.261,200,123.137)~~,**
**~~(122.376,233.261,311,133.137)~~, ~~(152.971,233.261,200,203.848)~~,**
**~~(166.523,233.261,111,213.848)~~, (193.628,77.7538,200,284.559),**
**~~(220.734,77.7538,311,294.559)~~, (247.839,77.7538,0,322.843)}**]

### 3.3.2.4 Interpolation of the semantics on the original trajectory at any point on the generalized version

The SELF structure built using algorithm 3.6 can be used to interpolate the semantics on the original trajectory at any point on the generalized version of it. In Fig. 3.11, the semantics at 'P' can be calculated by applying a linear interpolation on the segment defined by the projection of vertices '8' and '9' on the simplified trajectory. The algorithm 3.7 is used for computing the semantics at P.

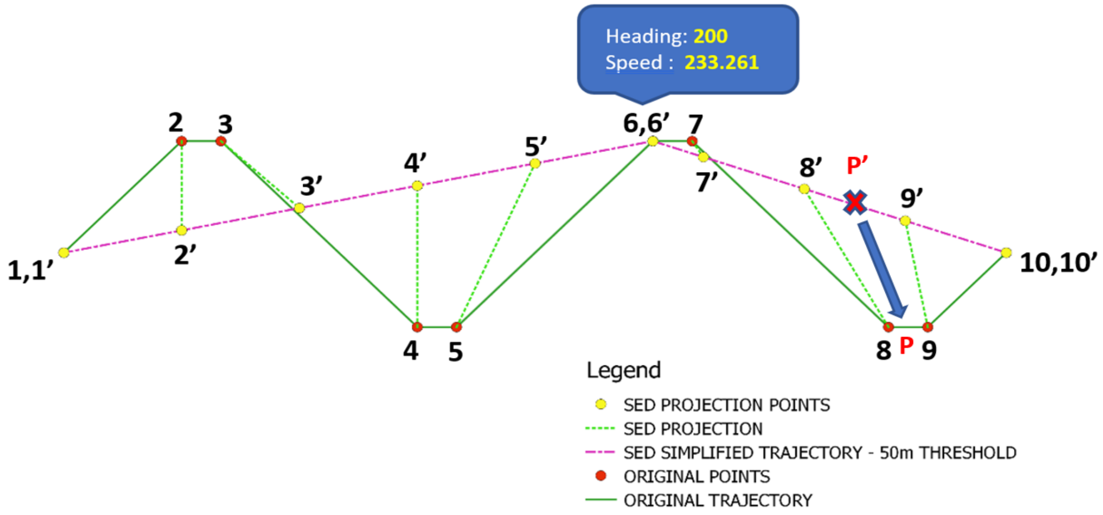


FIGURE 3.11: P' IS THE SED PROJECTION OF P

**Algorithm 3.7:** Interpolation of the semantics on the original trajectory at any point on the generalized version

```
Input:
    1. Geometry of the generalized version
    2. SELF structure of the generalized version
    3. Point on the generalized version at which the semantics have to be interpolated ($P'$)
Output:
    1. Interpolated semantics at $P$ ($P$ is the point projected at $P'$)
Steps:
    1. Find the distance between starting point and point $P'$ ($d_{P'}$)
    2. FOR EACH value IN THE SELF array
    3. IF simplified accumulated length > $d_{P'}$
        a. Retrieve the semantics at previous position and the next position
        b. Use linear interpolation within the retrieved semantics to interpolate the
           semantics at point $P$
    4. END
    5. RETURN the interpolated semantics
```

## 3.4 Implementation

SED based simplification algorithm and SELF data structure have been implemented in PostgreSQL 9.4 using PL/pgSQL. The spatial extension PostGIS 2.3 has been installed in PostgreSQL 9.4 (PostgreSQL, PostGIS). The implemented algorithm takes a set of raw mobility points and simplification threshold as input. The simplified version is then associated with the SELF data structure. The user can select any point on the simplified trajectory, to retrieve the original semantics. The experiments were performed on a sea vessel trajectory dataset obtained in Aegean Sea, Greece.

### 3.4.1 PostGIS Extension

Table 3.5 summarizes the built-in functions available with PostGIS extension that were utilized for implementing SED based simplification algorithm and developing the SELF data structure. The table describes each function’s input and output parameters.

Table 3.5: Built-in PostGIS functions used in developing algorithms (Source: http://www.postgis.net/docs/)

| FUNCTION | INPUT | OUTPUT |
|---|---|---|
| **ST_PointN** — Return the Nth point in the Line geometry. | GEOMETRY of a line string, integer n | Nth point in a single line string as GEOMETRY |
| **ST_Length** — Returns the 2D length of the geometry in meters | GEOMETRY | 2D Cartesian length of the geometry as FLOAT |
| **ST_StartPoint** — Returns the first point of a LINESTRING geometry as a POINT. | Line GEOMETRY | Line GEOMETRY |
| **ST_EndPoint** — Returns the last point of a LINESTRING geometry as a POINT. | Line GEOMETRY | Point GEOMETRY |
| **ST_X** — Return the X coordinate of the point | Point GEOMETRY | FLOAT |
| **ST_Y** — Return the Y coordinate of the point | Point GEOMETRY | FLOAT |
| **ST_Distance** — For geometry type Returns the 2D Cartesian distance between two geometries in projected units (based on spatial ref). | GEOMETRY g1,<br>GEOMETRY g2 | FLOAT |
| **ST_AsText** — Return the Well-Known Text (WKT) representation of the geometry/geography without SRID metadata. | GEOMETRY | TEXT |
| **ST_Simplify —** Returns a "simplified" version of the given geometry using the Douglas-Peucker algorithm. | GEOMETRY,<br>THRESHOLD | SIMPLIFIED GEOMETRY |
| **ST_MakeLine** — Creates a Line string from array of points | GEOMETRY array | GEOMETRY |

### 3.4.2 SELF functions

Using PL/pgSQL, the procedural language for PostgreSQL, both the SED based simplification and SELF structure algorithms were added as new (user defined) functions. Eleven new functions were implemented. The example statement for calling each user defined function is shown in Table 6 along with the output.  The functions are called on the synthetic trajectory dataset 'TR2' (Fig. 3.4).

Table 3.6: User-defined functions and example statements for calling the functions

| FUNCTION | INPUT | OUTPUT |
|---|---|---|
| **SELF_AVG_VLCY**— Returns the average velocity of the trip. | Raw mobility data points (Spatial relation) | Numeric value |
| select **SELF_AVG_VLCY** ('TR2' :: regclass);<br>**Output:** 20.0 | | |

<table>
<tr><td><b>SELF_ORIG_GEOM</b> – Function for constructing the trajectory from raw mobility data points.</td><td>Raw mobility data points (Spatial relation)</td><td>Line geometry</td></tr>
<tr><td colspan="3">select ST_ASTEXT (<b>SELF_ORIG_GEOM</b> (‘TR2’ :: regclass));<br><b>Output:</b> "LINESTRING (482980 4101964,483010 4101994,483020 4101994,483070 4101944,483080 4101944,483130 4101994,483140 4101994,483190 4101944,483200 4101944,483220 4101964)"</td></tr>
<tr><td><b>FIND_POINT</b> – Function for finding the point on the line based on the distance from starting point</td><td>Starting point, Ending Point, Distance to the point to be found</td><td>Point geometry</td></tr>
<tr><td colspan="3">select ST_ASTEXT (<b>FIND_POINT</b> (ST_MAKEPOINT (0,0), ST_MAKEPOINT (10,0), <b>5.0</b>));<br><b>Output:</b> POINT (5,0)</td></tr>
<tr><td><b>SELF_SED_SP</b> — Function for calculating distance to each SED point from starting points on straight line</td><td>Raw mobility data points (Spatial relation)</td><td>Numeric array</td></tr>
<tr><td colspan="3">select <b>SELF_SED_SP</b> (‘TR2’ :: regclass);<br><b>Output:</b> { 0 , 20.0 , 40.0 , 60.0 , 80.0 , 100.0 , 120.0 , 160.0 , 200.0 , 240.0 }</td></tr>
<tr><td><b>COUNT_POINTS</b>— Function to count the number of points in the line geometry</td><td>Line geometry</td><td>Count of points as a numeric value</td></tr>
<tr><td colspan="3">select <b>COUNT_POINTS</b> (SELF_ORIG_GEOM (‘TR2’ :: regclass)));<br><b>Output:</b> 10</td></tr>
<tr><td><b>SORT_ARRAY</b>– Function to sort numeric array in descending order</td><td>Numeric array</td><td>Numeric array sorted in descending order</td></tr>
<tr><td colspan="3">select <b>SORT_ARRAY</b> ( ARRAY [ 1 , 2 , 3 , 2 , 5 ] );<br><b>Output:</b> { 5 , 3 , 2 , 2 , 1 }</td></tr>
<tr><td><b>CHK_PT</b> – Function to check if the point is already present in the geometry array</td><td>Point geometry array, Point geometry</td><td>Returns the number based on the following criteria:<br>0 – Not present<br>1 - Present</td></tr>
<tr><td colspan="3">select <b>CHK_PT</b> (ARRAY [ POINT (1,1) , POINT (1,2) ] , ST_MAKEPOINT(1,1) );<br><b>Output:</b> 1<br>(The point (1,1) is present in the array)</td></tr>
<tr><td><b>SED_SIMPFY</b> – Function to build the simplified geometry of trajectory based on the SED distance</td><td>Raw mobility data points (Spatial relation), Simplification threshold</td><td>Simplified geometry as an array of points</td></tr>
<tr><td colspan="3">select ST_ASTEXT (UNNEST (<b>SED_SIMPFY</b> (‘TR2’ :: regclass , 50.0 )));<br><b>Output:</b> "POINT (482980 4101964)"<br>"POINT (483130 4101994)"<br>"POINT (483220 4101964)"</td></tr>
<tr><td><b>SELF_DYN_STR_ML_SP</b> – To build the SELF structure and to return the simplified geometry with SELF structure stored in the attribute table</td><td>Raw mobility data points (Spatial relation), Simplification threshold, Speed based compression threshold, heading based compression threshold</td><td>SELF structure for dynamic lines</td></tr>
</table>

| | | |
|---|---|---|
| select UNNEST (SELF_ARRAY (**SELF_DYN_STR_ML_SP**('TR2' :: regclass,50.0,.0.0,0.0)));<br>[ POINT (482980 4101964), POINT (483220 4101964), 322.843 , {(0.000,0,511,0.00), (30.594,233.261,400,42.426), (61.188,233.261,211,52.426), (91.782,233.261,400,123.137), (122.376,233.261,511,133.137), (152.971,233.261,400,203.848), (166.523,233.261,211,213.848), (193.628,77.7538,400,284.559), (220.734,77.7538,511,294.559), (247.839,77.7538,0,322.843)}] | | |
| **SELF_DYN_ASTEXT** – To display the SELF structure in user readable format | SELF structure | SELF structure in Text format |
| select **SELF_DYN_ASTEXT** (SELF_DYN_STR_ML_SP ( 'TR2' :: regclass , 50.0 ) );<br>SPOINT (482980 4101964) -- EPOINT (483220 4101964) – AL : 322.843 - AD: {(0.000,0,511,0.00), (30.594,233.261,400,42.426), (61.188,233.261,211,52.426), (91.782,233.261,400,123.137), (122.376,233.261,511,133.137), (152.971,233.261,400,203.848), (166.523,233.261,211,213.848), (193.628,77.7538,400,284.559), (220.734,77.7538,511,294.559), (247.839,77.7538,0,322.843)}] | | |
| **SELF_ITP_DIST_ML_SP** - To interpolate the semantics using SELF structures | Simplified line geometry, SELF structure, point , pointer | Array of interpolated semantic value |
| select **SELF_ITP_DIST_ML_SP**(ST_MAKELINE(SED_SIMPFY('TR2' :: regclass, 50.0 )),SELF_DYN_STR_ML_SP('TR2'::regclass, 50.0),ST_MAKEPOINT(483100,4101988),2)<br>**Output :** 233.261 | | |

### 3.4.3 Experimental Data

To demonstrate the effectiveness of the SED based simplification and SELF structure in interpolating the semantics, experimentation is done on different trajectory datasets with different values for speed based and heading based simplification. The experiments ran over the vessel trajectories for August 2013 in the Aegean Sea as collected by the MarineTraffic Automatic Identification System (AIS) [*MarineTraffic* 2017]. The ship speed is measured in knots multiplied by 10 and heading represents the azimuth of the ship bow in degrees. In order for the set of features to be representative for a wide range of spatio-temporal characteristics, it was decided to choose trajectories with different number of mobility data points. TR1 in Table 3.7 refers to the trajectory in Fig. 3.4

TABLE 3.7: LENGTH AND THE NUMBER OF POINTS AVAILABLE IN THE SELECTED TRAJECTORY DATASET

| TRAJECTORY NAME | LENGTH (in meters) | TOTAL NUMBER OF POINTS | STARTING TIME | ENDING TIME |
|---|---|---|---|---|
| TR1 | 322.842 | 10 | "2017-05-23 01:00:00" | "2017-05-23 01:00:12" |
| TR2 | 40933.052 | 55 | "2013-08-08 02:08:00" | "2013-08-08 06:01:00" |
| TR3 | 1032677.072 | 500 | "2013-08-01 00:02:00" | "2013-08-02 15:17:00" |
| TR4 | 6961025.269 | 5030 | "2013-08-01 00:04:00" | "2013-08-31 21:42:00" |
| TR5 | 17207258.378 | 8553 | "2013-08-01 00:02:00" | "2013-08-31 23:39:00" |

Fig. 3.12 to 3.15 show the original trajectories listed in Table 3.7.

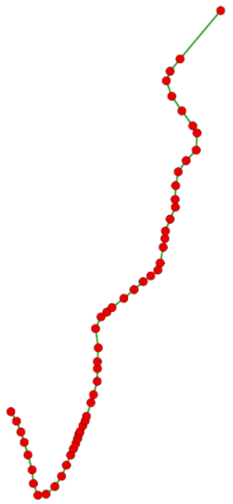


FIGURE 3.12: TRAJECTORY – TR2

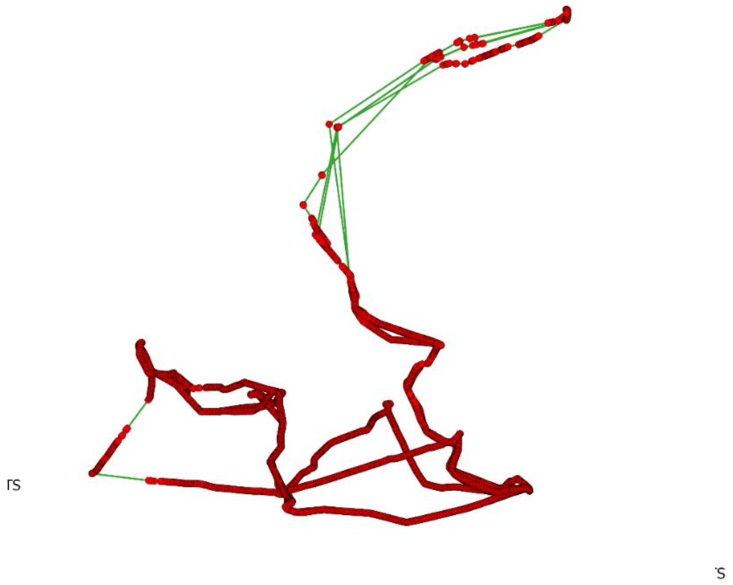


FIGURE 3.13: TRAJECTORY – TR4

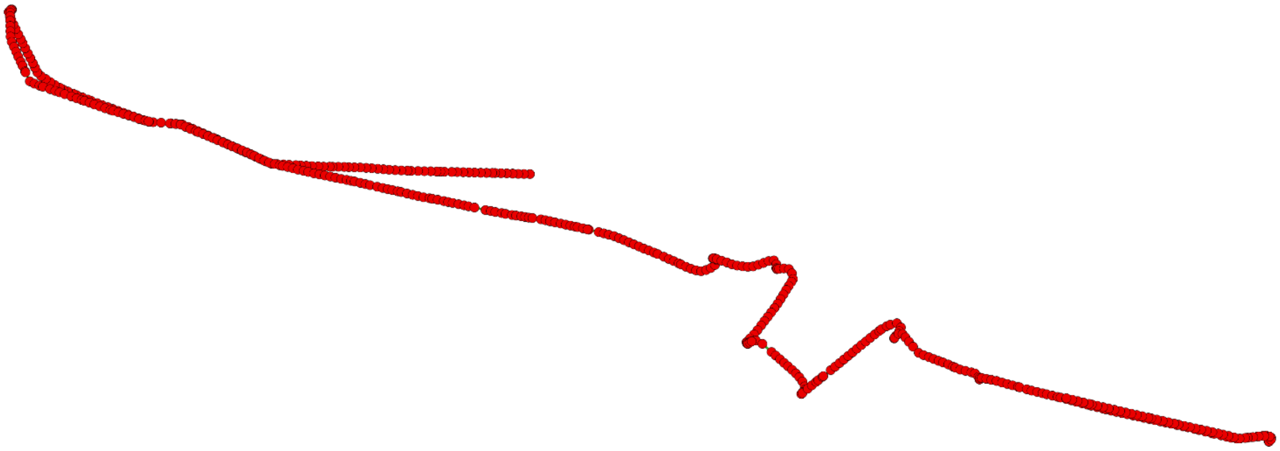


FIGURE 3.14: TRAJECTORY – TR3

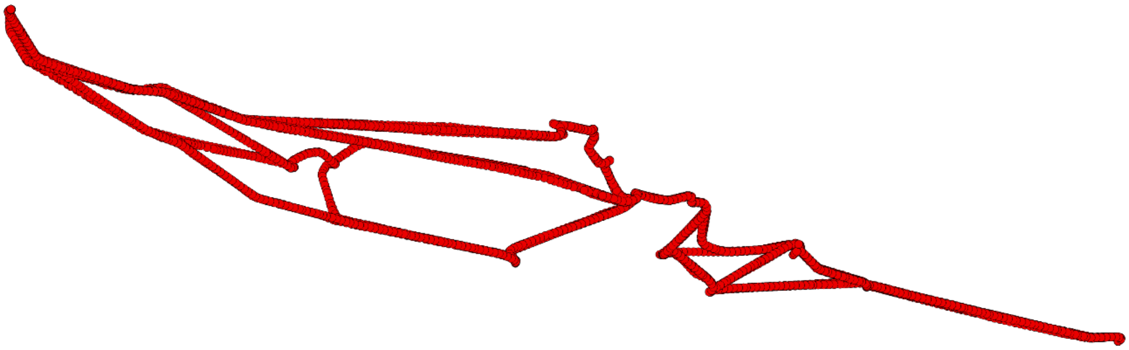


FIGURE 3.15: TRAJECTORY – TR5

### 3.4.4 Experiments

#### 3.4.4.1 SED based Simplification:

The SED based simplified version and the original trajectory is shown in Fig. 3.16 with a simplification threshold of 90.0 meters.

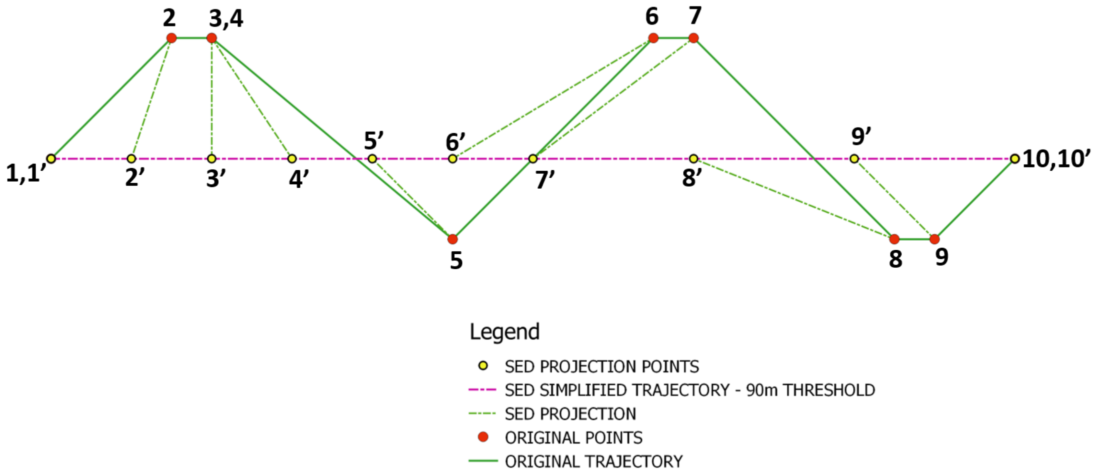


FIGURE 3.16: ORIGINAL AND SIMPLIFIED VERSION (90 M THRESHOLD) OF A SAMPLE TRAJECTORY (TABLE. 3.8)

TABLE 3.8: ATTRIBUTE TABLE OF THE TRAJECTORY SHOWN IN FIG. 3.16

| ID | SPEED (in KNOTS X 10) | LOCATION | TIME | HEADING (degrees) |
|---|---|---|---|---|
| 1 | 0 | POINT (482980 4101964) | 2017-05-23 01:00:00.000 | 311 |
| 2 | 233.261 | POINT (483010 4101994) | 2017-05-23 01:00:01.000 | 200 |
| 3 | 0 | POINT (483020 4101994) | 2017-05-23 01:00:02.000 | 0 |
| 4 | 233.261 | POINT (483020 4101994) | 2017-05-23 01:00:03.000 | 111 |
| 5 | 233.261 | POINT (483080 4101944) | 2017-05-23 01:00:04.000 | 311 |
| 6 | 233.261 | POINT (483130 4101994) | 2017-05-23 01:00:05.000 | 200 |
| 7 | 233.261 | POINT (483140 4101994) | 2017-05-23 01:00:06.000 | 111 |
| 8 | 77.7538 | POINT (483190 4101944) | 2017-05-23 01:00:08.000 | 200 |
| 9 | 77.7538 | POINT (483200 4101944) | 2017-05-23 01:00:10.000 | 311 |
| 10 | 77.7538 | POINT (483220 4101964) | 2017-05-23 01:00:12.000 | 0 |

In Fig. 3.16, the locations ‘3’ and ’4’ represent the same point as the vessel has stopped at location ‘3’ and stayed there for a minute before leaving. Even though the locations ‘3’ and ‘4’ are same, their SED projections are different. The cause is due to the points ‘3’ and ‘4’ have different timestamps.

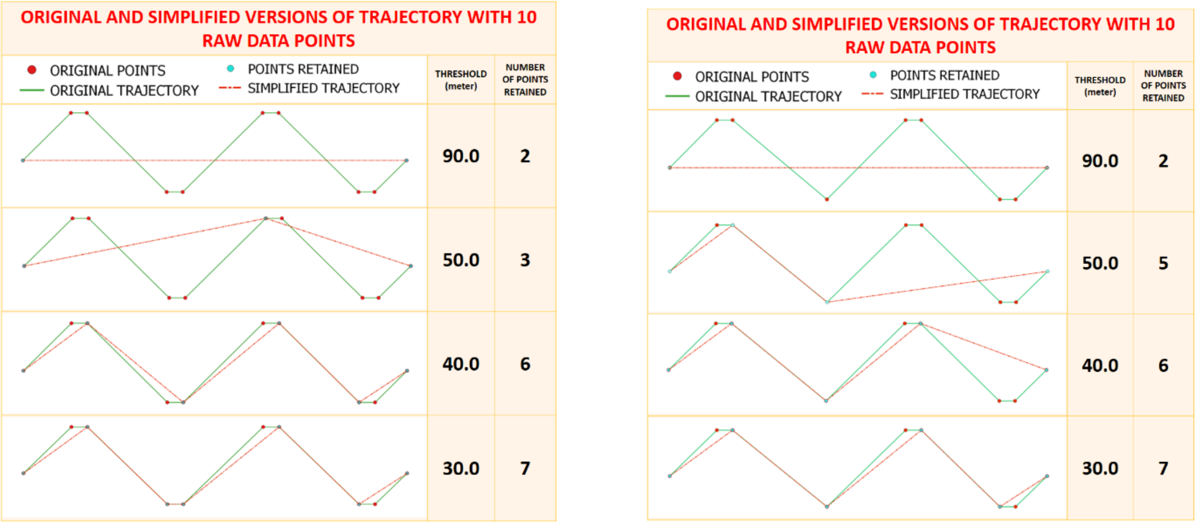
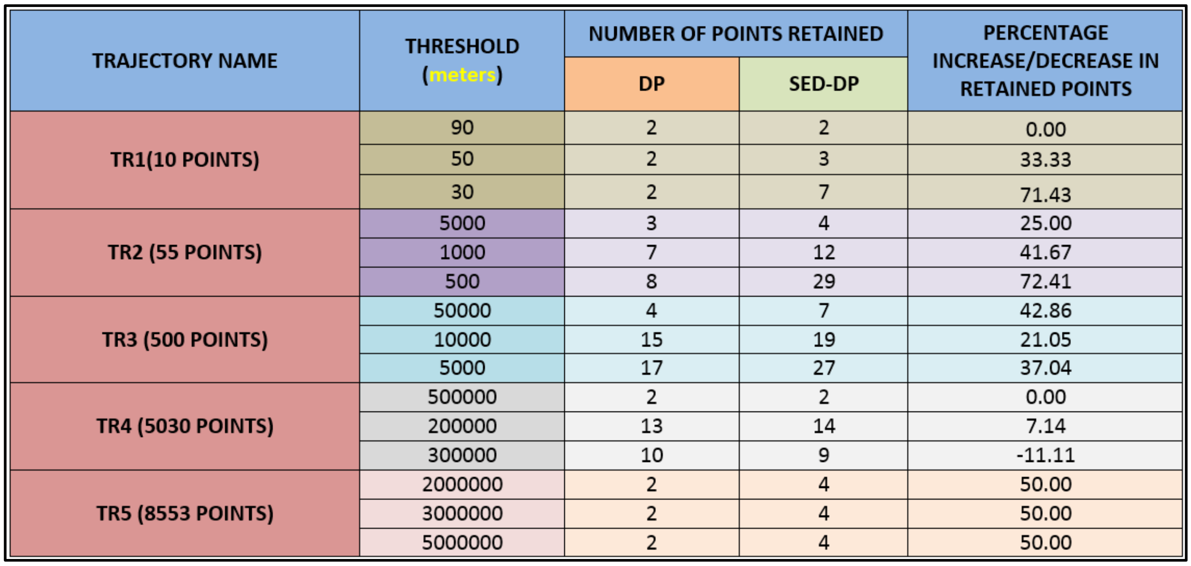


FIGURE 3.17: COMPARISON OF ORIGINAL AND SIMPLIFIED TRAJECTORY AT DIFFERENT LEVELS OF SIMPLIFICATION

Comparing the number of points retained by SED-DP simplification with DP simplification indicates that SED-DP simplification always retains more number of points than DP simplification (Fig. 3.18).

TABLE 3.9: LENGTH AND THE NUMBER OF POINTS AVAILABLE IN THE SELECTED DATA



| TRAJECTORY NAME | THRESHOLD (meters) | NUMBER OF POINTS RETAINED | | PERCENTAGE INCREASE/DECREASE IN RETAINED POINTS |
|---|---|---|---|---|
| | | DP | SED-DP | |
| TR1(10 POINTS) | 90 | 2 | 2 | 0.00 |
| | 50 | 2 | 3 | 33.33 |
| | 30 | 2 | 7 | 71.43 |
| TR2 (55 POINTS) | 5000 | 3 | 4 | 25.00 |
| | 1000 | 7 | 12 | 41.67 |
| | 500 | 8 | 29 | 72.41 |
| TR3 (500 POINTS) | 50000 | 4 | 7 | 42.86 |
| | 10000 | 15 | 19 | 21.05 |
| | 5000 | 17 | 27 | 37.04 |
| TR4 (5030 POINTS) | 500000 | 2 | 2 | 0.00 |
| | 200000 | 13 | 14 | 7.14 |
| | 300000 | 10 | 9 | -11.11 |
| TR5 (8553 POINTS) | 2000000 | 2 | 4 | 50.00 |
| | 3000000 | 2 | 4 | 50.00 |
| | 5000000 | 2 | 4 | 50.00 |

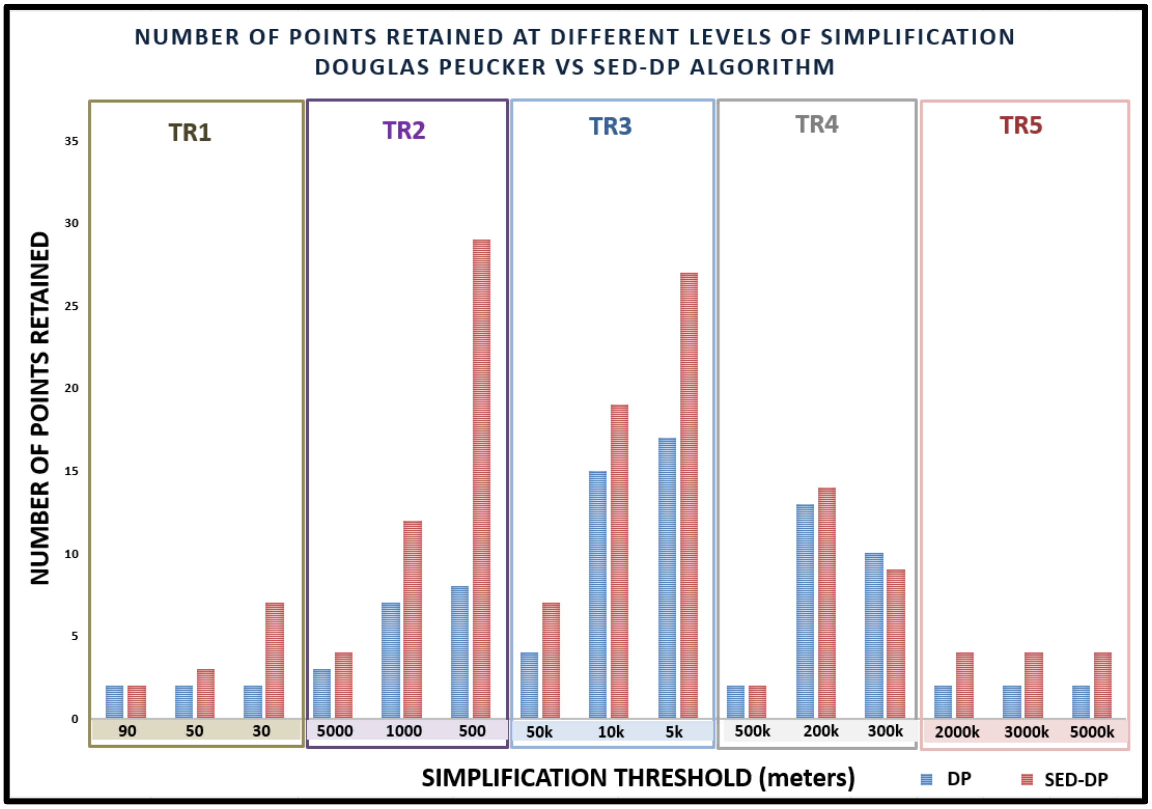


FIGURE 3.18: COMPARISON OF NUMBER OF POINTS RETAINED AT DIFFERENT LEVELS OF SIMPLIFICATION BETWEEN DP AND SED-DP ALGORITHM

### 3.4.4.2 SELF STRUCTURE

The SELF structure has been built on the original trajectory shown in Fig. 3.16 with an SED based simplification threshold of 90.0 meters and both the speed and heading based compression levels set as 0. The original semantics at each point on the simplified trajectory are listed in Table 3.10

TABLE 3.10: SEMANTICS AT EACH POINTS OF ORIGINAL TRAJECTORY AND ITS SIMPLIFIED VERSION SHOWN IN FIG. 3.16

| POINT ON ORIGINAL TRAJECTORY | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| ACCUMULATED DISTANCE ON ORIGINAL TRAJECTORY (m) | 0.0 | 42.43 | 52.43 | 52.43 | 130.53 | 201.24 | 211.24 | 281.95 | 291.95 | 320.23 |
| CORRESPONDING POINT ON SIMPLIFIED TRAJECTORY | 1′ | 2′ | 3′ | 4′ | 5′ | 6′ | 7′ | 8′ | 9′ | 10′ |
| ACCUMULATED DISTANCE ON SIMPLIFIED TRAJECTORY (m) | 0.0 | 20.0 | 40.0 | 60.0 | 80.0 | 100.0 | 120.0 | 160.0 | 200.0 | 240.0 |
| HEADING | 311 | 200 | 111 | 200 | 311 | 200 | 111 | 200 | 311 | 0 |
| SPEED (in KNOTS) | 0 | 233.26 | 0 | 233.26 | 233.26 | 233.26 | 233.26 | 77.75 | 77.75 | 77.75 |

TABLE 3.11: INTERPOLATED SEMANTICS AT EACH POINT CLICKED ON THE SIMPLIFIED TRAJECTORY

| POINT CLICKED ON THE SIMPLIFIED TRAJECTORY | 1′ | 2′ | 3′ | 4′ | 5′ | 6′ | 7′ | 8′ | 9′ | 10′ |
|---|---|---|---|---|---|---|---|---|---|---|
| INTERPOLATED DISTANCE | 0.0 | 42.43 | 52.43 | 52.43 | 130.53 | 201.24 | 211.24 | 281.95 | 291.95 | 320.23 |
| INTERPOLATED HEADING | 311 | 200 | 111 | 200 | 311 | 200 | 111 | 200 | 311 | 0 |
| INTERPOLATED SPEED (in KNOTS) | 0 | 233.26 | 0 | 233.26 | 233.26 | 233.26 | 233.26 | 77.75 | 77.75 | 77.75 |

In Fig. 3.16 the projected points 3′ and 4′ correspond to the same location (points 3 and 4) on the original trajectory as the original accumulated distance on original trajectory at the points 3 and 4 is 52.43 meters.

The original trajectory shown in Fig. 3.19 contains 500 points.  Ten points have been chosen randomly to check the effectiveness of SELF data structure in interpolating the semantics. Table 3.12 lists the interpolated semantics at all 10 sample points at different values of speed based compression threshold.

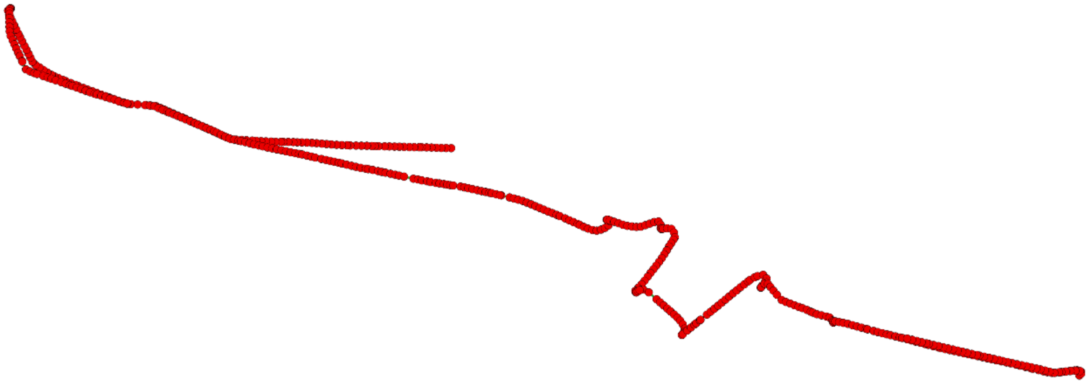
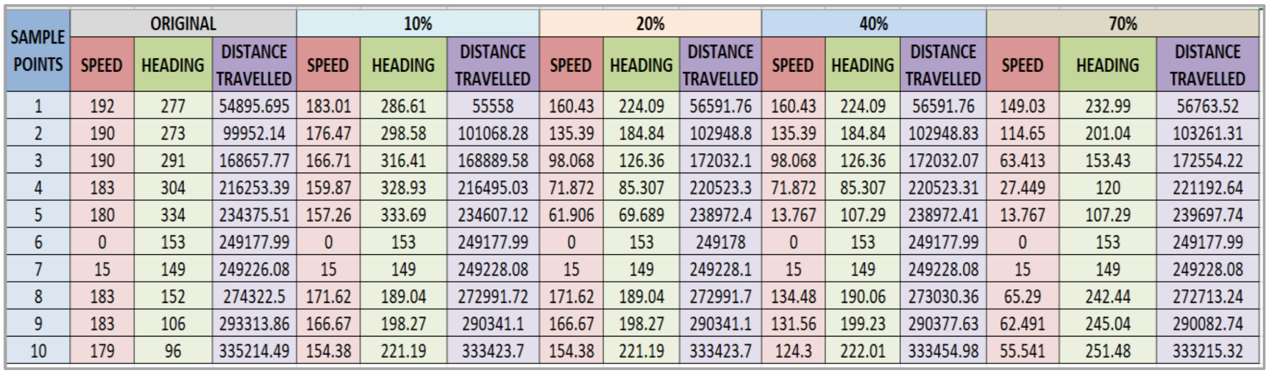


FIGURE 3.19: ORIGINAL TRAJECTORY WITH 500 RAW DATA POINTS

TABLE 3.12: INTERPOLATED SEMANTICS AT DIFFERENT LEVELS OF SPEED BASED COMPRESSION THRESHOLD



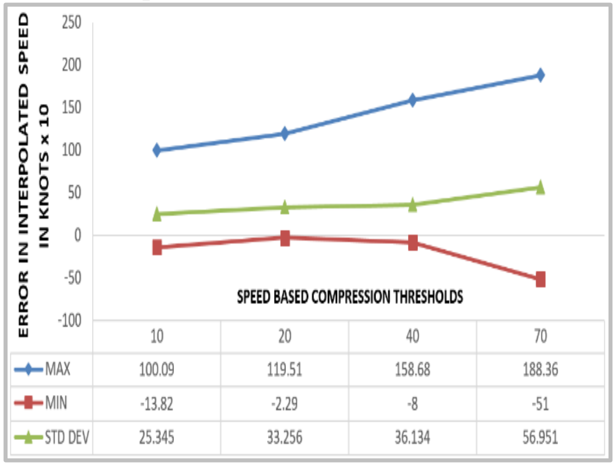
| SAMPLE POINTS | ORIGINAL | | | 10% | | | 20% | | | 40% | | | 70% | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | SPEED | HEADING | DISTANCE TRAVELLED | SPEED | HEADING | DISTANCE TRAVELLED | SPEED | HEADING | DISTANCE TRAVELLED | SPEED | HEADING | DISTANCE TRAVELLED | SPEED | HEADING | DISTANCE TRAVELLED |
| 1 | 192 | 277 | 54895.695 | 183.01 | 286.61 | 55558 | 160.43 | 224.09 | 56591.76 | 160.43 | 224.09 | 56591.76 | 149.03 | 232.99 | 56763.52 |
| 2 | 190 | 273 | 99952.14 | 176.47 | 298.58 | 101068.28 | 135.39 | 184.84 | 102948.8 | 135.39 | 184.84 | 102948.83 | 114.65 | 201.04 | 103261.31 |
| 3 | 190 | 291 | 168657.77 | 166.71 | 316.41 | 168889.58 | 98.068 | 126.36 | 172032.1 | 98.068 | 126.36 | 172032.07 | 63.413 | 153.43 | 172554.22 |
| 4 | 183 | 304 | 216253.39 | 159.87 | 328.93 | 216495.03 | 71.872 | 85.307 | 220523.3 | 71.872 | 85.307 | 220523.31 | 27.449 | 120 | 221192.64 |
| 5 | 180 | 334 | 234375.51 | 157.26 | 333.69 | 234607.12 | 61.906 | 69.689 | 238972.4 | 13.767 | 107.29 | 238972.41 | 13.767 | 107.29 | 239697.74 |
| 6 | 0 | 153 | 249177.99 | 0 | 153 | 249177.99 | 0 | 153 | 249178 | 0 | 153 | 249177.99 | 0 | 153 | 249177.99 |
| 7 | 15 | 149 | 249226.08 | 15 | 149 | 249228.08 | 15 | 149 | 249228.1 | 15 | 149 | 249228.08 | 15 | 149 | 249228.08 |
| 8 | 183 | 152 | 274322.5 | 171.62 | 189.04 | 272991.72 | 171.62 | 189.04 | 272991.7 | 134.48 | 190.06 | 273030.36 | 65.29 | 242.44 | 272713.24 |
| 9 | 183 | 106 | 293313.86 | 166.67 | 198.27 | 290341.1 | 166.67 | 198.27 | 290341.1 | 131.56 | 199.23 | 290377.63 | 62.491 | 245.04 | 290082.74 |
| 10 | 179 | 96 | 335214.49 | 154.38 | 221.19 | 333423.7 | 154.38 | 221.19 | 333423.7 | 124.3 | 222.01 | 333454.98 | 55.541 | 251.48 | 333215.32 |

There are three possible outcomes when running algorithm 7 to interpolate the semantics on the original trajectory at any points of the simplified version from SELF structure. The type of error in interpolation is classified depending on the outcome given in Table 3.13

TABLE 3.13: POSSIBLE OUTCOME AND ERROR CLASSIFICATION

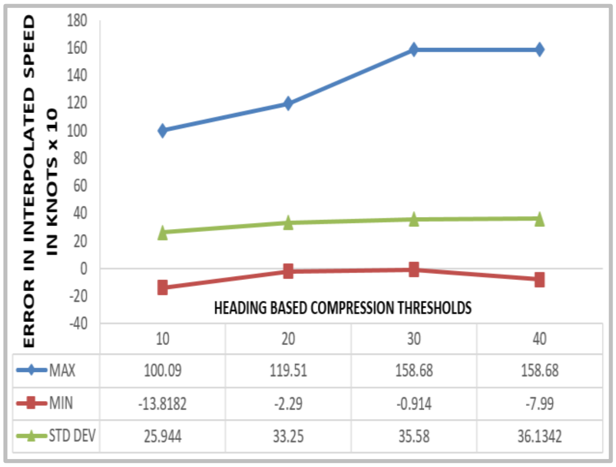
| POSSIBLE OUTCOME | ERROR CLASSIFICATION |
|---|---|
| Interpolated Semantics > Actual Semantics | Negative |
| Interpolated Semantics < Actual Semantics | Positive |
| Interpolated Semantics = Actual Semantics | Zero |

Similarly, the semantics at all 500 points is interpolated by algorithm 3.7. Fig 3.20-3.25 compares the maximum (positive error), minimum (Negative error), and standard deviation in the interpolated semantics at different levels of speed and heading based compression.



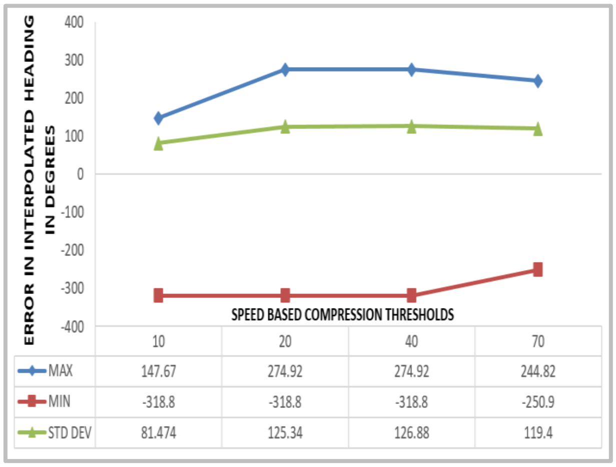
|  | 10 | 20 | 40 | 70 |
|---|---|---|---|---|
| MAX | 100.09 | 119.51 | 158.68 | 188.36 |
| MIN | -13.82 | -2.29 | -8 | -51 |
| STD DEV | 25.345 | 33.256 | 36.134 | 56.951 |

FIGURE 3.20: ERROR IN INTERPOLATED SPEED VS SPEED BASED COMPRESSION



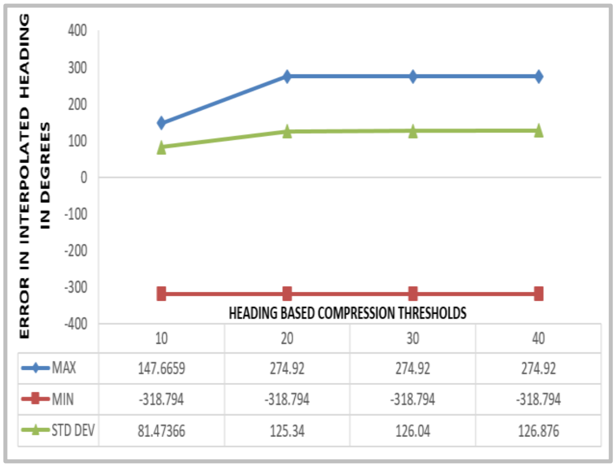
|  | 10 | 20 | 30 | 40 |
|---|---|---|---|---|
| MAX | 100.09 | 119.51 | 158.68 | 158.68 |
| MIN | -13.8182 | -2.29 | -0.914 | -7.99 |
| STD DEV | 25.944 | 33.25 | 35.58 | 36.1342 |

FIGURE 3.21: ERROR IN INTERPOLATED SPEED VS HEADING BASED COMPRESSION



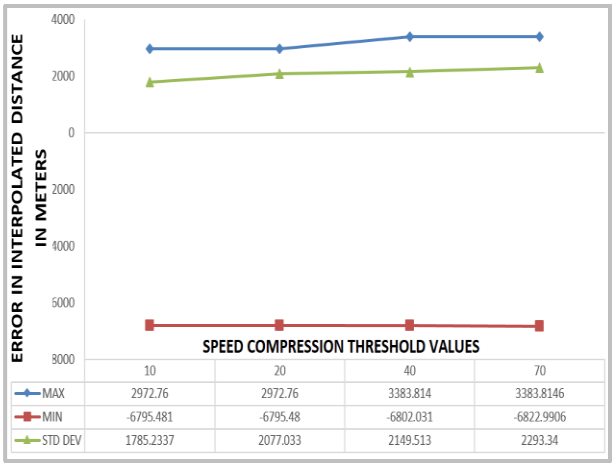
|  | 10 | 20 | 40 | 70 |
|---|---|---|---|---|
| MAX | 147.67 | 274.92 | 274.92 | 244.82 |
| MIN | -318.8 | -318.8 | -318.8 | -250.9 |
| STD DEV | 81.474 | 125.34 | 126.88 | 119.4 |

FIGURE 3.22: ERROR IN INTERPOLATED HEADING VS SPEED BASED COMPRESSION



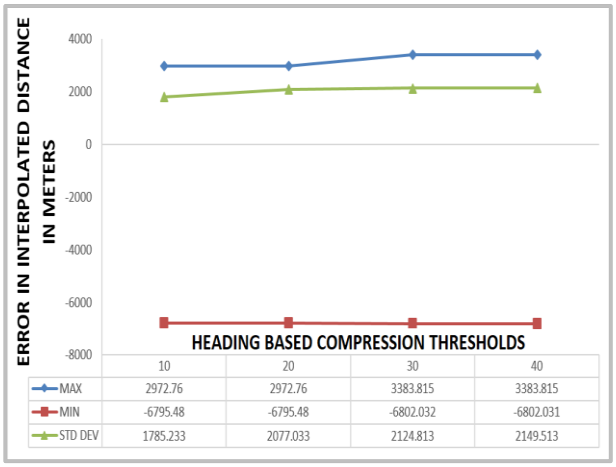
|  | 10 | 20 | 30 | 40 |
|---|---|---|---|---|
| MAX | 147.6659 | 274.92 | 274.92 | 274.92 |
| MIN | -318.794 | -318.794 | -318.794 | -318.794 |
| STD DEV | 81.47366 | 125.34 | 126.04 | 126.876 |

FIGURE 3.23: ERROR IN INTERPOLATED HEADING VS HEADING BASED COMPRESSION



| | 10 | 20 | 40 | 70 |
|---|---|---|---|---|
| MAX | 2972.76 | 2972.76 | 3383.814 | 3383.8146 |
| MIN | -6795.481 | -6795.48 | -6802.031 | -6822.9906 |
| STD DEV | 1785.2337 | 2077.033 | 2149.513 | 2293.34 |

FIGURE 3.24: ERROR IN INTERPOLATED DISTANCE VS SPEED BASED COMPRESSION

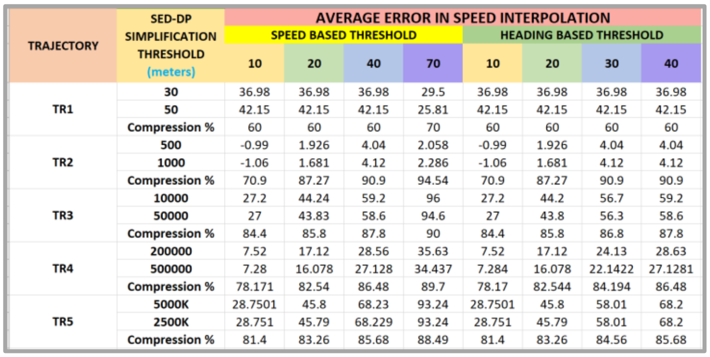


| | 10 | 20 | 30 | 40 |
|---|---|---|---|---|
| MAX | 2972.76 | 2972.76 | 3383.815 | 3383.815 |
| MIN | -6795.48 | -6795.48 | -6802.032 | -6802.031 |
| STD DEV | 1785.233 | 2077.033 | 2124.813 | 2149.513 |

FIGURE 3.25: ERROR IN INTERPOLATED DISTANCE VS HEADING BASED COMPRESSION

Speed based compression and heading based compression produce almost the same amount of error in the interpolated semantics (Fig. 3.20-3.25). The different spatio-temporal characteristics of the datasets play major role in semantic interpolation error. The semantic interpolation algorithm (Algorithm 3.7) has run with different levels of compression on these trajectory datasets (Table.3.14-3.16).

TABLE 3.14: AVERAGE ERROR IN SPEED INTERPOLATION



| TRAJECTORY | SED-DP SIMPLIFICATION THRESHOLD (meters) | AVERAGE ERROR IN SPEED INTERPOLATION | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | SPEED BASED THRESHOLD | | | | HEADING BASED THRESHOLD | | | |
| | | 10 | 20 | 40 | 70 | 10 | 20 | 30 | 40 |
| TR1 | 30 | 36.98 | 36.98 | 36.98 | 29.5 | 36.98 | 36.98 | 36.98 | 36.98 |
| | 50 | 42.15 | 42.15 | 42.15 | 25.81 | 42.15 | 42.15 | 42.15 | 42.15 |
| | Compression % | 60 | 60 | 60 | 70 | 60 | 60 | 60 | 60 |
| TR2 | 500 | -0.99 | 1.926 | 4.04 | 2.058 | -0.99 | 1.926 | 4.04 | 4.04 |
| | 1000 | -1.06 | 1.681 | 4.12 | 2.286 | -1.06 | 1.681 | 4.12 | 4.12 |
| | Compression % | 70.9 | 87.27 | 90.9 | 94.54 | 70.9 | 87.27 | 90.9 | 90.9 |
| TR3 | 10000 | 27.2 | 44.24 | 59.2 | 96 | 27.2 | 44.2 | 56.7 | 59.2 |
| | 50000 | 27 | 43.83 | 58.6 | 94.6 | 27 | 43.8 | 56.3 | 58.6 |
| | Compression % | 84.4 | 85.8 | 87.8 | 90 | 84.4 | 85.8 | 86.8 | 87.8 |
| TR4 | 200000 | 7.52 | 17.12 | 28.56 | 35.63 | 7.52 | 17.12 | 24.13 | 28.63 |
| | 500000 | 7.28 | 16.078 | 27.128 | 34.437 | 7.284 | 16.078 | 22.1422 | 27.1281 |
| | Compression % | 78.171 | 82.54 | 86.48 | 89.7 | 78.17 | 82.544 | 84.194 | 86.48 |
| TR5 | 5000K | 28.7501 | 45.8 | 68.23 | 93.24 | 28.7501 | 45.8 | 58.01 | 68.2 |
| | 2500K | 28.751 | 45.79 | 68.229 | 93.24 | 28.751 | 45.79 | 58.01 | 68.2 |
| | Compression % | 81.4 | 83.26 | 85.68 | 88.49 | 81.4 | 83.26 | 84.56 | 85.68 |

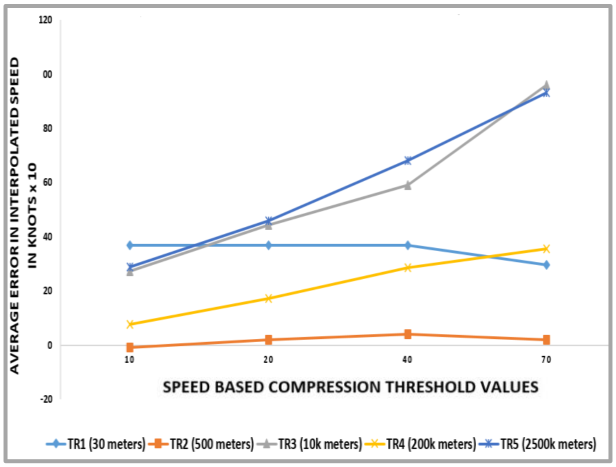
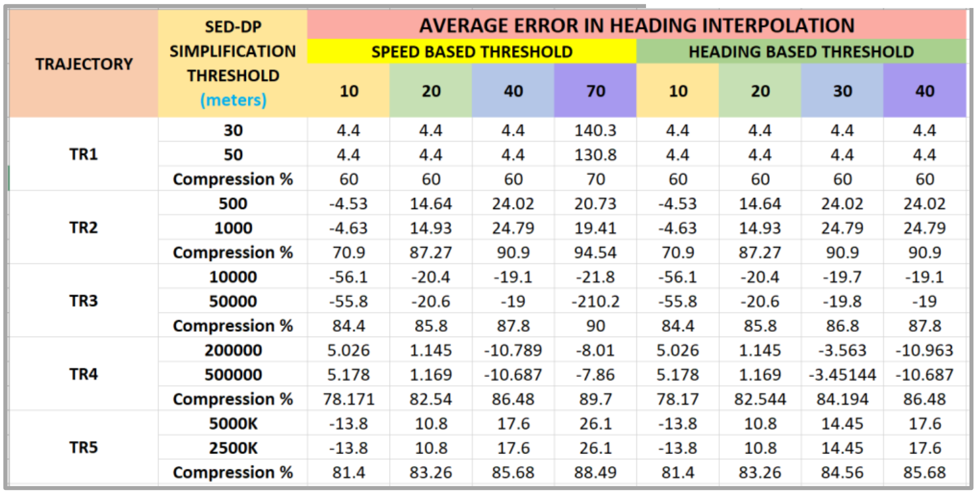


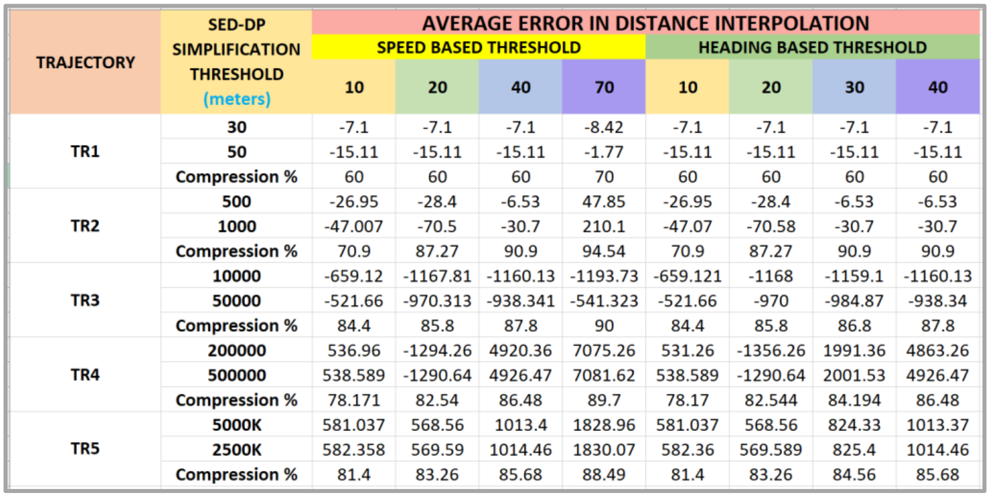
FIGURE 3.26: ERROR IN INTERPOLATED SPEED VS SPEED BASED COMPRESSION

TABLE 3.15: AVERAGE ERROR IN HEADING INTERPOLATION



| TRAJECTORY | SED-DP SIMPLIFICATION THRESHOLD (meters) | AVERAGE ERROR IN HEADING INTERPOLATION | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | SPEED BASED THRESHOLD | | | | HEADING BASED THRESHOLD | | | |
| | | 10 | 20 | 40 | 70 | 10 | 20 | 30 | 40 |
| TR1 | 30 | 4.4 | 4.4 | 4.4 | 140.3 | 4.4 | 4.4 | 4.4 | 4.4 |
| | 50 | 4.4 | 4.4 | 4.4 | 130.8 | 4.4 | 4.4 | 4.4 | 4.4 |
| | Compression % | 60 | 60 | 60 | 70 | 60 | 60 | 60 | 60 |
| TR2 | 500 | -4.53 | 14.64 | 24.02 | 20.73 | -4.53 | 14.64 | 24.02 | 24.02 |
| | 1000 | -4.63 | 14.93 | 24.79 | 19.41 | -4.63 | 14.93 | 24.79 | 24.79 |
| | Compression % | 70.9 | 87.27 | 90.9 | 94.54 | 70.9 | 87.27 | 90.9 | 90.9 |
| TR3 | 10000 | -56.1 | -20.4 | -19.1 | -21.8 | -56.1 | -20.4 | -19.7 | -19.1 |
| | 50000 | -55.8 | -20.6 | -19 | -210.2 | -55.8 | -20.6 | -19.8 | -19 |
| | Compression % | 84.4 | 85.8 | 87.8 | 90 | 84.4 | 85.8 | 86.8 | 87.8 |
| TR4 | 200000 | 5.026 | 1.145 | -10.789 | -8.01 | 5.026 | 1.145 | -3.563 | -10.963 |
| | 500000 | 5.178 | 1.169 | -10.687 | -7.86 | 5.178 | 1.169 | -3.45144 | -10.687 |
| | Compression % | 78.171 | 82.54 | 86.48 | 89.7 | 78.17 | 82.544 | 84.194 | 86.48 |
| TR5 | 5000K | -13.8 | 10.8 | 17.6 | 26.1 | -13.8 | 10.8 | 14.45 | 17.6 |
| | 2500K | -13.8 | 10.8 | 17.6 | 26.1 | -13.8 | 10.8 | 14.45 | 17.6 |
| | Compression % | 81.4 | 83.26 | 85.68 | 88.49 | 81.4 | 83.26 | 84.56 | 85.68 |

TABLE 3.16: AVERAGE ERROR IN DISTANCE INTERPOLATION



| TRAJECTORY | SED-DP SIMPLIFICATION THRESHOLD (meters) | AVERAGE ERROR IN DISTANCE INTERPOLATION | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | SPEED BASED THRESHOLD | | | | HEADING BASED THRESHOLD | | | |
| | | 10 | 20 | 40 | 70 | 10 | 20 | 30 | 40 |
| TR1 | 30 | -7.1 | -7.1 | -7.1 | -8.42 | -7.1 | -7.1 | -7.1 | -7.1 |
| | 50 | -15.11 | -15.11 | -15.11 | -1.77 | -15.11 | -15.11 | -15.11 | -15.11 |
| | Compression % | 60 | 60 | 60 | 70 | 60 | 60 | 60 | 60 |
| TR2 | 500 | -26.95 | -28.4 | -6.53 | 47.85 | -26.95 | -28.4 | -6.53 | -6.53 |
| | 1000 | -47.007 | -70.5 | -30.7 | 210.1 | -47.07 | -70.58 | -30.7 | -30.7 |
| | Compression % | 70.9 | 87.27 | 90.9 | 94.54 | 70.9 | 87.27 | 90.9 | 90.9 |
| TR3 | 10000 | -659.12 | -1167.81 | -1160.13 | -1193.73 | -659.121 | -1168 | -1159.1 | -1160.13 |
| | 50000 | -521.66 | -970.313 | -938.341 | -541.323 | -521.66 | -970 | -984.87 | -938.34 |
| | Compression % | 84.4 | 85.8 | 87.8 | 90 | 84.4 | 85.8 | 86.8 | 87.8 |
| TR4 | 200000 | 536.96 | -1294.26 | 4920.36 | 7075.26 | 531.26 | -1356.26 | 1991.36 | 4863.26 |
| | 500000 | 538.589 | -1290.64 | 4926.47 | 7081.62 | 538.589 | -1290.64 | 2001.53 | 4926.47 |
| | Compression % | 78.171 | 82.54 | 86.48 | 89.7 | 78.17 | 82.544 | 84.194 | 86.48 |
| TR5 | 5000K | 581.037 | 568.56 | 1013.4 | 1828.96 | 581.037 | 568.56 | 824.33 | 1013.37 |
| | 2500K | 582.358 | 569.59 | 1014.46 | 1830.07 | 582.36 | 569.589 | 825.4 | 1014.46 |
| | Compression % | 81.4 | 83.26 | 85.68 | 88.49 | 81.4 | 83.26 | 84.56 | 85.68 |

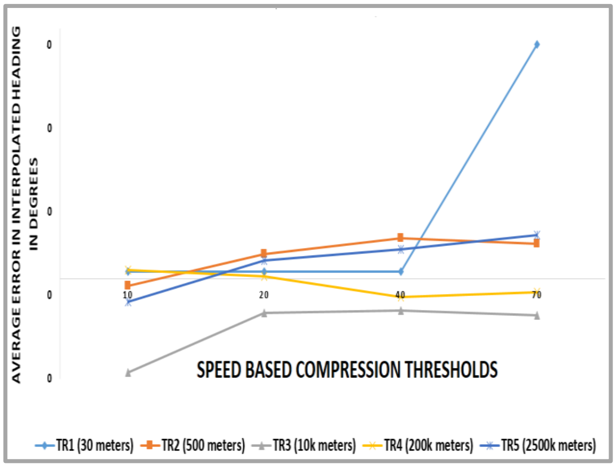


FIGURE 3.27: ERROR IN INTERPOLATED HEADING VS SPEED BASED COMPRESSION

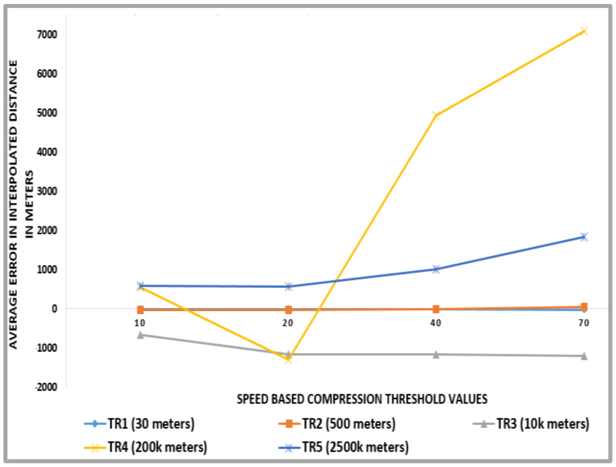


FIGURE 3.28: ERROR IN INTERPOLATED DISTANCE VS SPEED BASED COMPRESSION

It can be seen from Fig. 3.26-3.28 that there is a positive correlation between average error and the level of simplification. The percentage error in interpolation increases when the values for compression is also increased. Noticeably, average error in distance interpolation for "TR4" suddenly increases after 20% of speed based compression. The cause is due to an increase in compression level discards more points from SELF structure (Fig. 3.29, 3.30). But this number would change due to the different spatiotemporal characteristics of the datasets. So, the level of compression can be decided based on the application and the required accuracy in semantics interpolation.

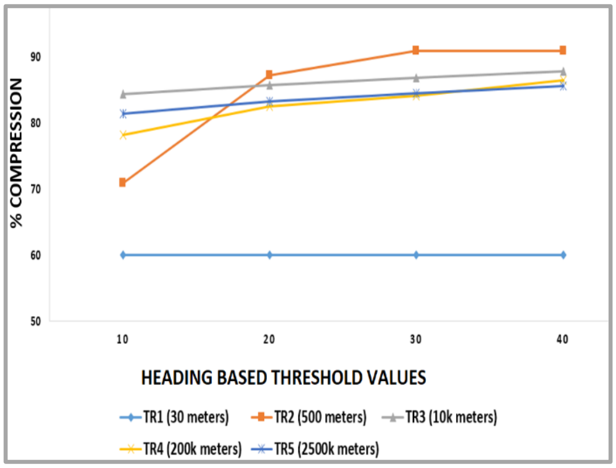


FIGURE 3.29 : % OF COMPRESSION VS HEADING BASED COMPRESSION

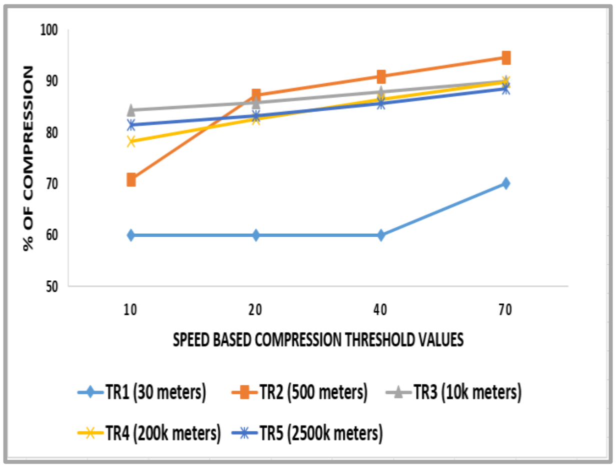


FIGURE 3.30 : % OF COMPRESSION VS SPEED BASED COMPRESSION VALUES

## 3.5 Conclusions

This paper summarizes the implementation and testing of a method for semantically enriched simplification of trajectories. The method combines the Synchronous Euclidean Distance (SED) based simplification algorithm and SELF (Semantically Enriched Line simpliFication) data structure to preserve the semantics associated with the actual trajectories. The method has been implemented in PostgreSQL 9.4 with PostGIS extension using PL/pgSQL to support dynamic lines and tested with both synthetic and real-world features.

The method applies two kinds of semantic based reduction: speed based and heading based. Both the compression techniques produce the same amount error in the interpolated semantics. However, the results of the experiments indicate that the different spatio-temporal characteristics of the datasets play a major role in the semantic interpolation error.

The comparison results between SED-DP simplification and DP simplification indicate that SED-DP simplification always retains more number of points which are more significant in forming the trajectory than other points as they better convey the trajectory characteristics for a particular context.

Future work includes the development of a visualization framework to provide an enhanced user experience. This will help in facilitating the adoption of the SELF structure in various application domains with need for semantically enhanced multiscale representation of linear features.

Integrating these libraries to a Graph database (such as Neo4j) so that the extended functionalities of Graph database can be utilized in trajectory data management is another future goal.

## Acknowledgements

This research has received funding from the Natural Sciences and Engineering Research Council of Canada (NSERC), Discovery Grants Program.

# REFERENCES

Abam, M.A., et al., 2010. Streaming algorithms for line simplification. *Discrete & Computational Geometry,* 43 (3), 497–515. doi:10.1007/s00454-008-9132-4

Alvares, L.O., et al., 2007. A model for enriching trajectories with semantic geographical information. In: *The Proceedings of the 15th annual ACM international symposium on advances in geographic information systems*, 7–9 November, Seattle, WA. Article No. 22.

Cromley, R.G., 1991. Hierarchical methods of line simplification. *Cartography and Geographic Information Science,* 18 (2), 125–131. doi:10.1559/152304091783805563

Douglas, D.H. and Peucker, T.K., 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer*, 10 (2), 112–122. doi:10.3138/FM57-6770-U75U-7727

MarineTraffic, 2017. Live-ships map: AIS [online]. Available from: http://www.marinetraffic.com/ais/ [Last visited, June 27, 2017]

K. Buchin, M. Buchin, and J. Gudmundsson. Detecting single file movement. In *GIS '08: Proceedings of the 16thACM SIGSPATIAL international conference on Advances in geographic information systems,* pages 1-10, New York, NY, USA, 2008. ACM.

Keates, J.S., 1989. *Cartographic design and production*. 2nd ed. Essex: Longman.

Meratnia, N. and de By, R.A., 2004. Spatiotemporal compression techniques for moving point objects. In: *Proceedings of the international conference on extending database technology* (EDBT). Berlin: Springer, 765–782. LNCS 2992.

Parent, C. et al., 2013. Semantic Trajectories Modeling and Analysis. *ACM Computing Surveys, Vol. 45, No.4, Article 42.* doi: http://dx.doi.org/10.1145/2501654.2501656

PostGIS Reference. Chapter 8. Retrieved from http://postgis.net/docs/reference.html#Management_Functions, Published on: 26th September 2016, Accessed on: 17th June 2017

PostgreSQL 9.4.11 Documentation Chapter 40. PL/pgSQL - SQL Procedural Language. Retrieved from https://www.postgresql.org/docs/9.4/static/plpgsql-statements.html, Published on: 18th December 2014, Accessed on: 18th June 2017

QiuLei Guo and Hassan A. Karimi, 2016. A Topology-Inferred Graph-Based Heuristic Algorithm for Map Simplification. In: *Transactions in GIS,* doi:10.1111/tgis.12188

Richter, K.F., Schmid, F., and Laube, P., 2012. Semantic trajectory compression: representing urban movement in a nutshell. *Journal of Spatial Information Science,* (4), 3–30.

Robinson, Joel L. Morrison, Phillip C. Muehrcke, A. Jon Kimerling, Stephen C. Guptill, Elements of Cartography, 6th Edition ISBN: 978-0-471-55579-7

Shahriari, N., and V. Tao, 2002. Minimizing Positional Errors in Line Simplification Us-ing Adaptive Tolerance Values. In: *Symposium on Geospatial Theory, Processing and Applica-tion,* 4(3), 213-223.

Spaccapietra, S., et al., 2008. A conceptual view on trajectories. *Data & Knowledge Engineering,* 65 (1), 126–146. Retrieved from : http://www.sciencedirect.com/science/article/pii/S0169023X07002078

Stefanakis, E., 2012. Trajectory generalization under space constraints. In: *The proceedings of the 7th international conference on geographic information science (GIScience 2012)*, 18–21 September, Columbus, OH.

Stefanakis, E., 2015. SELF: Semantically enriched Line simpliFication. In: *International Journal of Geographical Information Science,* Vol. 29, Iss. 10, 2015, Pages 1826-1844 doi: 10.1080/13658816.2015.1053092

Tienaah, T., Stefanakis, E., and Coleman, D., 2015.S Contextual Douglas-Peucker simplification. *Geomatica Journal*, 69 (3), 327-338, https://doi.org/10.5623/cig2015-306

Weibel, R., 1996. A typology of constraints to line simplification. In: *Proceedings of 7th international symposium on spatial data handling*, 12–16 August, Delft. IGU, 533–546.

Weibel, R., 1997. Generalization of spatial data: principles and selected algorithms. In: M. Kreveld, et al., eds. *Algorithmic foundations of geographic information systems*. Berlin: Springer, 99–152.

Wu, et alShil – Ting and Mercedes Rocio Gonzales Marquez 2003. *Proceedings of the XVI Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI'03)* 1530-1834/03 doi :10.1109/SIBGRA.2003.1240992

Yan, Z., et al. 2011. SeMiTri: A framework for semantic annotation of heterogeneous trajectories. *In: The proceedings of the 14th international conference on Extending Database Technology* (EDBT 2011). New York: ACM, 259–270.

# 4. Modelling and Analysis of Semantically Enriched Simplified Trajectories using Graph Databases

## Abstract

Graph databases are utilized in modelling the huge volume of spatio-temporal data generated by moving objects that are equipped with GPS devices. This modelled spatial and temporal information in graphs can be used in performing trajectory analysis such as optimum path finding or identification of collision risk. At the same time, this massive data becomes difficult to handle using graph databases as the millions of raw data points make the graph model complex. Thus, trajectory simplification techniques are applied to reduce the number of vertices representing a trajectory. However, elimination of intermediate points by simplification process leads to loss of semantics associated with the trajectories. These semantics are dependent on the application domain. For example, a trajectory of a moving vessel can convey information about time, distance travelled, bearing, or velocity. This research proposes a graph data model that enriches the simplified geometry of trajectories with the semantics lost during simplification process. Original trajectories initially modelled and stored in a PostgreSQL/PostGIS database are simplified according to both their spatial and temporal characteristics using the Synchronous Euclidean Distance (SED), while the Semantically Enriched Line simpliFication (SELF) data structure is built to preserve the semantics of the vertices eliminated during the simplification process. Then, enriched simplified trajectories are transferred to a Neo4j database and modelled in terms of nodes and edges using graphs. Trajectories can then be processed using Cypher query language and Neo4j spatial procedures. A visualization tool has been developed on top of Neo4j graph database to support the semantic retrieval and visualization of trajectories.

## 4.1 Introduction

GPS devices mounted on moving objects generate streams of geo-location data which describe the path travelled by the object during a period of time, this path is called trajectory. The advent of satellite technologies has enabled the usage of GPS devices on moving objects. Common application domains using trajectory data are city planning, transportation management systems, and other location-aware applications [*K. Buchin et al.* 2008]. In the era of big data, graph databases address the major challenges in management and analysis of voluminous data. The concept of storing and representing data in terms of nodes, edges and properties makes graph databases different from relational databases and this is well suited for trajectory data management systems [*Stefanakis* 2017]. Spatial analysis capabilities have already been added to graph database systems. For instance, Neo4j, one of the most prevalent graph database systems, provides of a spatial plugin called Neo4j Spatial to facilitate spatial operations on geospatial data modelled using graphs [*Neo4j Spatial Plugin* 2017].

Over the last decade, researchers have focused on modelling and analyzing raw trajectory data points using graph databases. Data reduction has always been necessary due to tremendous amount of data points contained in raw trajectories. The process of retaining only certain points which are significant in forming a trajectory is known as trajectory simplification and this has evolved from cartographic line simplification methods [*Keates* 1989]. The basic idea of trajectory simplification is to retain those vertices that better convey the trajectory characteristics for a particular application domain. For example, the point at which a vessel has halted for longer duration may be more important than other vertices in vessel movement tracking. The conventional cartographic simplification techniques have limited applicability to trajectory simplification as they remove high-density vertices based only on a threshold distance.

For example, the most common simplification algorithm Douglas-Peucker (DP) does not consider the temporal dimension (time) associated with the raw points of trajectories. This limits DP algorithm to be utilized in trajectory simplification. The introduction of Synchronous Euclidean Distance (SED) as a criterion in trajectory simplification has been applied to overcome this limitation [*Meratnia and de By* 2004]. Furthermore, trajectory simplification results in a loss of semantics (e.g. speed, heading and distance travelled) associated with the points that are eliminated during the simplification process. Semantically Enriched Line simpliFication (SELF) data structure has recently been proposed to retain the semantic attributes associated with individual locations of original trajectory [*Stefanakis* 2015].

The combination of SELF data structure with SED criterion has been implemented and tested in PostgreSQL/PostGIS using PL/pgSQL [*Tamilmani and Stefanakis* 2017]. It has shown that semantics associated with the original trajectory are well retained in the simplified versions. The advent of graph databases [*Neo4j* 2017] has introduced an alternative and usually more efficient way of modelling and analyzing transportation data, including trajectory data, than traditional databases such as relational or object relation ones. This paper investigates the advantages of adopting a graph database to model and analyze the semantically enriched simplified trajectories generated by the combination SELF data structure with SED criterion [*Tamilmani and Stefanakis* 2017].

The enriched simplified trajectories are extracted from PostgreSQL/PostGIS databases and modelled into a Neo4j graph database. The nodes in the graph database are associated with the semantic attributes (speed, heading, time, distance travelled, latitude and longitude) of the trajectories, while the edges of the graph contain the simplified and original distances between the nodes representing the vertices. These attributes are utilized in performing trajectory data analysis.

The contributions of this study are twofold. First, to propose a graph model for transferring enriched simplified trajectories from PostgreSQL to Neo4j and further analyzing them as graphs using Cypher query language and spatial procedures. The latter has been done by utilizing the Neo4j-spatial plugin that provides the geospatial analysis capabilities to Neo4j graph database [*Neo4j Spatial Plugin* 2017]. Second, to support the semantic retrieval and visualization of modelled graph data in Neo4j. For this reason, a visualization tool has been developed on top of Neo4j for semantic interpolation at different levels of trajectory simplification.

The paper is organized as follows. Section 4.2 provides a literature review about trajectory simplification and describes the SELF structure and it's variants. Section 4.3 presents the steps followed to transfer an enriched simplified trajectory from PostgreSQL to Neo4j and demonstrates how trajectories can be analyzed based on their spatial and aspatial attributes using Cypher query language and Neo4j spatial procedures. Section 4.4 presents the visualization tool developed for interpolating the semantic values at different levels of trajectory simplification. Section 4.5 summarizes the contribution of this paper and discusses the potential of applying the proposed framework in various application domains.

## 4.2 Literture Review

### *4.2.1 Graph databases in trajectory data analysis*

Trajectories are formed by connecting a series of raw mobility data points. These individual data points include temporal dimension apart from latitude and longitude. Over the years, tabular structured relational databases have been utilised in accommodating connected points forming trajectories. The relational data model and an extended query language was proposed for supporting the modelling and querying of real world transportation networks. The comprehensive framework was built on top of OGC-complaint data models to support an algebraic-based network model. The proposed model has not addressed trajectories with millions of points though [*Hadi et al.,* 2014]. In addition, relational databases inefficient in dealing with relationships because connectedness leads to an increase in number of joins between the tables, which in fact affects the performance of the database [*Przemysław et al.,* 2016]. Graph databases help in leveraging the complex structure and dynamic relationships in connected trajectory data. The simple collection of nodes (vertices) and relationships (edges) facilitates the modelling of all varieties of data, from biological structures, to the transportation data. Year by year the focus on utilizing graph databases in managing, processing and analyzing spatio-temporal data has increased as the graph's internal structure is in the form of a network [*Gurfraind et al.* 2016]. Similar to SQL in relational databases, graph databases are also equipped with multifaceted and robust query language for retrieving information. On the flipside, visualizing all the nodes and edges would pose additional challenges as the graph layout is limited to display only certain number of nodes. *C. Partl et al.* [2016] presented a technique called "Pathfinder" for visualizing and analyzing large graphs. The authors have followed a query-based approach for allowing the users to refine the data based on specific starting and ending nodes. However, the criteria for choosing these starting and ending nodes are not properly defined.

This study is focused on analyzing raw data points representing trajectories using graph database. As millions of raw data points make the graph model more complex, it is often required to reduce the tremendous amount of data points representing each trajectory. This process is also known as trajectory simplification.

### *4.2.2 Trajectory Simplification*

[Refer to Section 3.2.1]

### *4.2.3 Semantically Enriched Line Simplification (SELF)*

[Refer to Section 3.2.3]

## 4.3 Graph data model and analysis

Trajectories are formed by connecting a series of raw mobility data points. Over the years graph databases have been used in analyzing massive amounts of data generated by GPS devices mounted on mobility vehicles. For example, a trip duration of 60 minutes, with the location being recorded every 1 second, results into a total of 3600 points. In a day trip, the dataset would contain 86,400 points. The number of nodes in the graph model is directly proportional to the number of points in the trajectory. As the number of nodes increase the graph model becomes more complex. Hence, it is necessary to reduce the volume of the dataset by applying trajectory simplification techniques.

This study introduces a graph data model that integrates the SELF data structure in the description of the simplified trajectory (Fig. 4.1). Each simplified trajectory has an origin and a destination. The starting and ending points of the original trajectory are converted into origin and destination nodes. The “INTERMEDIATE” nodes are attributed with original distance, speed, heading, time, latitude and longitude. In fact, these are the nodes which have been retained during the simplification process. The “SIDE” nodes have the following properties: original distance, speed, heading and time. These nodes represent the vertices which were eliminated during the simplification process. So, these nodes do not carry the latitude and longitude as properties. The edges connecting two intermediate nodes are weighted with both the original and simplified distance between the corresponding points on the trajectory. The relationship between two nodes are labelled as “NEXT”.

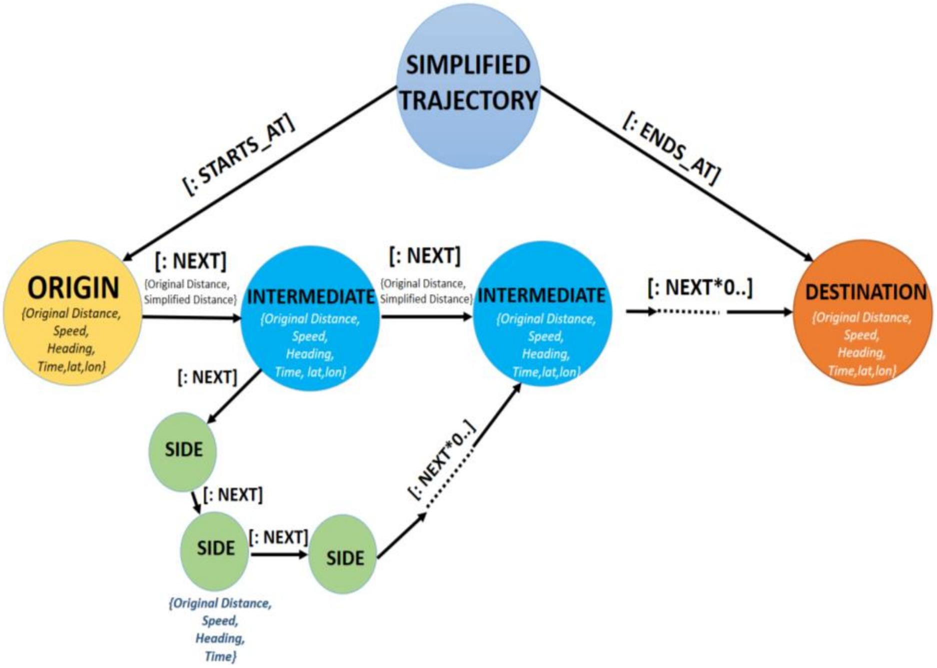

SIMPLIFIED TRAJECTORY
[: STARTS_AT]
[: ENDS_AT]
ORIGIN
{Original Distance, Speed, Heading, Time,lat,lon}
[: NEXT]
{Original Distance, Simplified Distance}
INTERMEDIATE
{Original Distance, Speed, Heading, Time, lat,lon}
[: NEXT]
{Original Distance, Simplified Distance}
INTERMEDIATE
{Original Distance, Speed, Heading, Time,lat,lon}
[: NEXT*0..]
DESTINATION
{Original Distance, Speed, Heading, Time,lat,lon}
[: NEXT]
SIDE
[: NEXT]
SIDE
[: NEXT]
SIDE
{Original Distance, Speed, Heading, Time}
[: NEXT*0..]

FIGURE 4.1: PROPOSED GRAPH MODEL FOR STORING A SIMPLIFIED TRAJECTORY ALONG WITH SEMANTICS USING SELF STRUCTURE

This study examines the modelling and analysis of enriched trajectories in a graph database. The approach involves three phases which are implemented in three system components (Fig. 4.2).

**Component 1:** Transferring the simplified geometry of a trajectory and SELF structure from PostgreSQL/PostGIS to Neo4j graph database using JDBC

**Component 2:** Performing spatial and attribute analysis on the modelled data using Cypher query language and Neo4j spatial procedures.

**Component 3:** Visualizing the simplified trajectory in web browser for semantic interpolation at different levels of trajectory simplification.

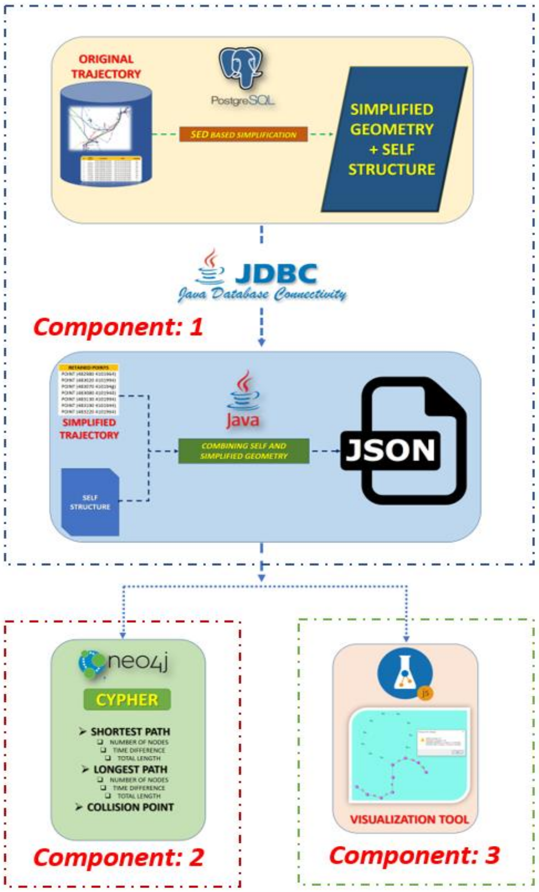


FIGURE 4.2: OVERALL SYSTEM ARCHITECTURE WITH THREE COMPONENTS

### *4.3.1 PostgreSQL/*PostGIS *to Neo4j bridging*

The purpose of first component in the overall system is to combine the simplified geometry of a trajectory with SELF structure and to convert the simplified geometry into nodes and edges with the semantics stored as properties. The simplified geometry data from PostgreSQL/PostGIS is mapped into Neo4j graph database using Java Database Connectivity (JDBC).

This component is implemented in three steps:

1. Simplifying a trajectory based on Synchronous Euclidean Distance (SED)
2. Generating nodes and edges from simplified geometry
3. Associating the nodes and edges with semantics from SELF structure

#### *4.3.1.1 Simplifying the trajectory based on Synchronous Euclidean Distance*

SELF structure for dynamic lines stores the semantics associated with individual points on the original trajectory along the simplified version. As shown in the Fig. 4.3, each point on the original trajectory is projected based on the SED to the simplified version. The sample trajectory shown in Fig. 4.3 has 10 points in the original version, while the simplified version retains 7 points.

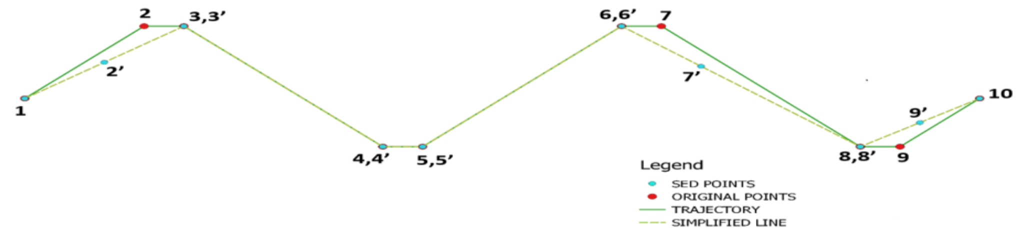


FIGURE 4.3: EXAMPLE ORIGINAL AND SIMPLIFIED TRAJECTORIES

TABLE 4.1: ATTRIBUTE TABLE FOR THE TRAJECTORY POINTS SHOWN IN FIGURE.4.3

| ID | SPEED (in KNOTS X 10) | LOCATION | TIME | HEADING (degrees) |
|---|---|---|---|---|
| 1 | 0 | POINT (482980 4101964) | 2017-05-23 01:00:00.000 | 311 |
| 2 | 233.261 | POINT (483010 4101994) | 2017-05-23 01:00:01.000 | 200 |
| 3 | 233.261 | POINT (483020 4101994) | 2017-05-23 01:00:02.000 | 111 |
| 4 | 233.261 | POINT (483070 4101944) | 2017-05-23 01:00:03.000 | 200 |
| 5 | 233.261 | POINT (483080 4101944) | 2017-05-23 01:00:04.000 | 311 |
| 6 | 233.261 | POINT (483130 4101994) | 2017-05-23 01:00:05.000 | 200 |
| 7 | 233.261 | POINT (483140 4101994) | 2017-05-23 01:00:06.000 | 111 |
| 8 | 77.7538 | POINT (483190 4101944) | 2017-05-23 01:00:08.000 | 200 |
| 9 | 77.7538 | POINT (483200 4101944) | 2017-05-23 01:00:10.000 | 311 |
| 10 | 77.7538 | POINT (483220 4101964) | 2017-05-23 01:00:12.000 | 0 |

For each vertex on the original trajectory, the corresponding SED point is tagged with the semantics given in Table. 4.1 [*Tamilmani and Stefanakis* 2017]. The entire SELF structure is represented as follows:

```
[ SPOINT (482980 4101964), EPOINT (483220 4101964), 322.843,
       {(0.000,0,311,0.00, 2017-05-23 01:00:00.000),
    (30.594,233.261,200,42.426, 2017-05-23 01:00:01.000),
    (61.188,233.261,111,52.426, 2017-05-23 01:00:02.000),
   (91.782,233.261,200,123.137, 2017-05-23 01:00:03.000),
  (122.376,233.261,311,133.137, 2017-05-23 01:00:04.000),
  (152.971,233.261,200,203.848, 2017-05-23 01:00:05.000),
  (166.523,233.261,111,213.848, 2017-05-23 01:00:06.000),
  (193.628,77.7538,200,284.559, 2017-05-23 01:00:08.000),
  (220.734,77.7538,311,294.559, 2017-05-23 01:00:10.000),
    (247.839,77.7538,0,322.843, 2017-05-23 01:00:12.000)}]
```

#### *4.3.1.2 Generating nodes and edges from simplified geometry*

Once the trajectory is simplified each point on the simplified geometry is compared with the points on the original trajectory to decide the node type. The points which were eliminated during the simplification are labelled as “SIDE” nodes and the retained points are labelled as “INTERMEDIATE” nodes, while the starting and ending points of the trajectory are labelled as “ORIGIN” and “DESTINATION” nodes. The “SIDE” nodes do not contain latitude and longitude, whereas all other nodes contain the spatial coordinates as these nodes represent the points which have been retained by the simplification algorithm.

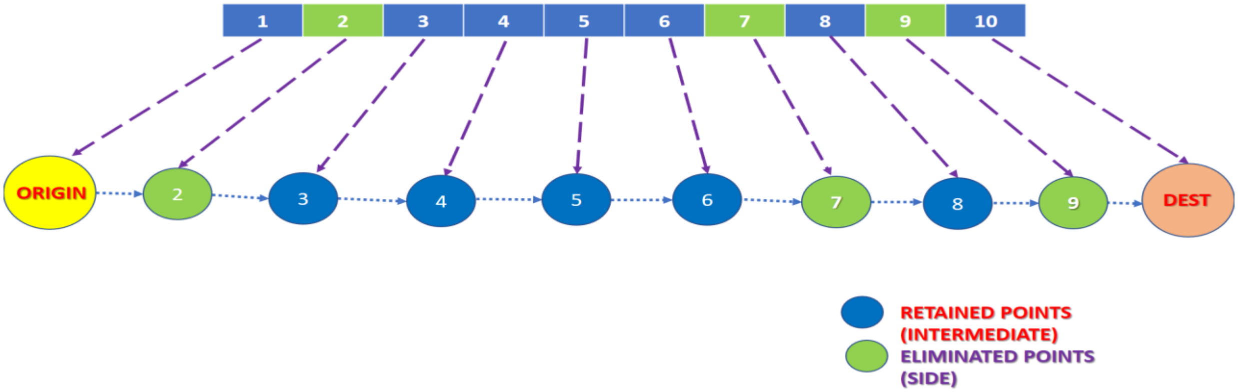


FIGURE 4.4: LABELLING THE NODES BASED ON THE CORRESPONDING RETAINED OR ELIMINATED POINTS

In our example, the points 2, 7 and 9 were lost during the simplification process (Fig. 4.4). So, these nodes are labelled as “SIDE”. The remaining nodes representing points 3, 4, 5, 6 and 8 are labelled as “INTERMEDIATE”. The first and last nodes are labelled as “ORIGIN” and “DESTINATION”.

#### *4.3.1.3 Associating the nodes and edges with semantics from SELF structure*

Once the nodes have been labelled, the SELF structure has to be parsed along with the simplified geometry to add the associated semantics as properties to the nodes. Algorithm 4.1 associates the nodes and edges with semantics from SELF structure. The developed algorithm takes the simplified geometry and SELF structure as input to add the properties to the nodes.

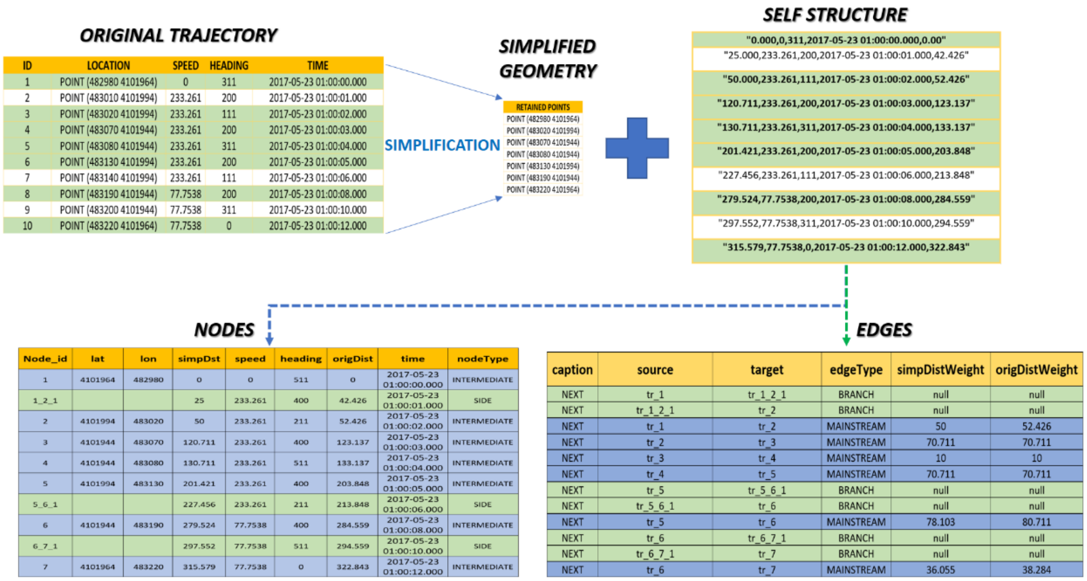
The accumulated length at every point on the simplified trajectory is calculated and stored in an array. Each simplified distance in SELF structure is searched through the accumulated length array. If a match is found then that node is labelled as "INTERMEDIATE" and attributed with the corresponding latitude, longitude, speed, heading, time and distance travelled as extracted from SELF structure. If the accumulated length does not match with simplified distance in SELF structure then that node is labelled as "SIDE" and attributed with the corresponding speed, heading, time and distance travelled from SELF structure.

Two separate CSV (Comma-Separated Values) files have been generated for uploading the trajectory data into Neo4j. One of them (all_nodes.csv) is for loading the nodes and the other (all_edges.csv) is for connecting these nodes with edges. Any two consecutive intermediate nodes are connected by an edge which is weighted with the original and simplified distance between those nodes (Fig. 4.5).

**Algorithm 4.1: Associating the nodes and edges with semantics from SELF structure**

```
Input:
        1.  Points in simplified trajectory as an array
        2.  SELF structure built on the simplified trajectory

Output:
        1.  CSV file containing nodes and edges
Steps:
        1.  Generate a CSV file for storing the nodes and edges
        2.  Initialize Node_ID to 1
        3.  FOR EACH point (P) in the geometry of simplified trajectory
        4.  Calculate the distance (d) to it's next point
        5.  Initialize CURRENT_COUNT to 1
        6.  FOR EACH simplified distance($d_s$) IN the SELF structure
                  i.  Find the match to the calculated distance (d)
                 ii.  IF $d=d_s$ THEN
                          1.  Name the node as "INTERMEDIATE"
                          2.  Assign the current Node_ID
                          3.  Add lat, lon, speed, heading, time and distance travelled
                              to the node
                iii.  ELSE
                          1.  Name the node as "SIDE"
                          2.  Format the Node_ID in the form of
                              "(Node_ID)_(Node_ID+1)_CURRENT_COUNT"
                          3.  Increase CURRENT_COUNT by 1
                          4.  Add speed, heading, time and distance travelled to the
                              node
        7.  END
        8.  Add the node to output CSV file
        9.  Increase Node_ID by 1
        10. END
        11. FOR EACH node(n) IN the CSV file
                  i.  IF label(n) = label(n+1) AND label(n) = "INTERMEDIATE"
                          1.  Connect the nodes n and n+1
                          2.  Add the original distance weight as
                              [distance_travelled(n+1)-distance_travelled(n)]
                          3.  Add the simplified distance weight as straight-line
                              distance between the nodes n and n+1
                 ii.  ELSE
                          1.  Connect the nodes n and n+1 without weights
        12. END
        13. Add the edge to output CSV file
        14. RETURN CSV file containing nodes and edges
```



| ID | LOCATION | SPEED | HEADING | TIME |
|---|---|---|---|---|
| 1 | POINT (482980 4101964) | 0 | 311 | 2017-05-23 01:00:00.000 |
| 2 | POINT (483010 4101994) | 233.261 | 200 | 2017-05-23 01:00:01.000 |
| 3 | POINT (483020 4101994) | 233.261 | 111 | 2017-05-23 01:00:02.000 |
| 4 | POINT (483070 4101944) | 233.261 | 200 | 2017-05-23 01:00:03.000 |
| 5 | POINT (483080 4101944) | 233.261 | 311 | 2017-05-23 01:00:04.000 |
| 6 | POINT (483130 4101994) | 233.261 | 200 | 2017-05-23 01:00:05.000 |
| 7 | POINT (483140 4101994) | 233.261 | 111 | 2017-05-23 01:00:06.000 |
| 8 | POINT (483190 4101944) | 77.7538 | 200 | 2017-05-23 01:00:08.000 |
| 9 | POINT (483200 4101944) | 77.7538 | 311 | 2017-05-23 01:00:10.000 |
| 10 | POINT (483220 4101964) | 77.7538 | 0 | 2017-05-23 01:00:12.000 |

| RETAINED POINTS |
|---|
| POINT (482980 4101964) |
| POINT (483020 4101994) |
| POINT (483070 4101944) |
| POINT (483080 4101944) |
| POINT (483130 4101994) |
| POINT (483190 4101944) |
| POINT (483220 4101964) |

| |
|---|
| "0.000,0,311,2017-05-23 01:00:00.000,0.00" |
| "25.000,233.261,200,2017-05-23 01:00:01.000,42.426" |
| "50.000,233.261,111,2017-05-23 01:00:02.000,52.426" |
| "120.711,233.261,200,2017-05-23 01:00:03.000,123.137" |
| "130.711,233.261,311,2017-05-23 01:00:04.000,133.137" |
| "201.421,233.261,200,2017-05-23 01:00:05.000,203.848" |
| "227.456,233.261,111,2017-05-23 01:00:06.000,213.848" |
| "279.524,77.7538,200,2017-05-23 01:00:08.000,284.559" |
| "297.552,77.7538,311,2017-05-23 01:00:10.000,294.559" |
| "315.579,77.7538,0,2017-05-23 01:00:12.000,322.843" |

| Node_id | lat | lon | simpDst | speed | heading | origDist | time | nodeType |
|---|---|---|---|---|---|---|---|---|
| 1 | 4101964 | 482980 | 0 | 0 | 511 | 0 | 2017-05-23 01:00:00.000 | INTERMEDIATE |
| 1_2_1 | | | 25 | 233.261 | 400 | 42.426 | 2017-05-23 01:00:01.000 | SIDE |
| 2 | 4101994 | 483020 | 50 | 233.261 | 211 | 52.426 | 2017-05-23 01:00:02.000 | INTERMEDIATE |
| 3 | 4101944 | 483070 | 120.711 | 233.261 | 400 | 123.137 | 2017-05-23 01:00:03.000 | INTERMEDIATE |
| 4 | 4101944 | 483080 | 130.711 | 233.261 | 511 | 133.137 | 2017-05-23 01:00:04.000 | INTERMEDIATE |
| 5 | 4101994 | 483130 | 201.421 | 233.261 | 400 | 203.848 | 2017-05-23 01:00:05.000 | INTERMEDIATE |
| 5_6_1 | | | 227.456 | 233.261 | 211 | 213.848 | 2017-05-23 01:00:06.000 | SIDE |
| 6 | 4101944 | 483190 | 279.524 | 77.7538 | 400 | 284.559 | 2017-05-23 01:00:08.000 | INTERMEDIATE |
| 6_7_1 | | | 297.552 | 77.7538 | 511 | 294.559 | 2017-05-23 01:00:10.000 | SIDE |
| 7 | 4101964 | 483220 | 315.579 | 77.7538 | 0 | 322.843 | 2017-05-23 01:00:12.000 | INTERMEDIATE |

| caption | source | target | edgeType | simpDistWeight | origDistWeight |
|---|---|---|---|---|---|
| NEXT | tr_1 | tr_1_2_1 | BRANCH | null | null |
| NEXT | tr_1_2_1 | tr_2 | BRANCH | null | null |
| NEXT | tr_1 | tr_2 | MAINSTREAM | 50 | 52.426 |
| NEXT | tr_2 | tr_3 | MAINSTREAM | 70.711 | 70.711 |
| NEXT | tr_3 | tr_4 | MAINSTREAM | 10 | 10 |
| NEXT | tr_4 | tr_5 | MAINSTREAM | 70.711 | 70.711 |
| NEXT | tr_5 | tr_5_6_1 | BRANCH | null | null |
| NEXT | tr_5_6_1 | tr_6 | BRANCH | null | null |
| NEXT | tr_5 | tr_6 | MAINSTREAM | 78.103 | 80.711 |
| NEXT | tr_6 | tr_6_7_1 | BRANCH | null | null |
| NEXT | tr_6_7_1 | tr_7 | BRANCH | null | null |
| NEXT | tr_6 | tr_7 | MAINSTREAM | 36.055 | 38.284 |

FIGURE 4.5: NODES AND EDGES AFTER COMBINING SELF STRUCTURE WITH SIMPLIFIED GEOMETRY

The generated nodes are then uploaded to Neo4j graph using the following cypher query. Nodes are connected by edges after uploading edges CSV file to Neo4j graph using the following cypher query:

```
LOAD CSV WITH HEADERS FROM
'file:///all_nodes.csv' AS csv
CREATE (t: trgraph {trid:csv.tr_id, ptid:csv.pt_id, lat:csv.lat, lon:csv.lon,
simpDist:csv.simpDst, speed:csv.speed, heading:csv.heading,
origDist:csv.origDist, time:csv.time, nodeType:csv.nodeType});
```

```
LOAD CSV WITH HEADERS FROM
'file:///all_edges.csv' as csv
MATCH ( t:trgraph  { ptid : csv.source} ), (t1 : trgraph {ptid : csv.target} )
WHERE  t.ptid <> t1.ptid AND t.trid=t1.trid
create (t)-[:NEXT{trid:csv.tr_id,
caption:csv.caption, source:csv.source,target:csv.target, edgeType:csv.edgeType,
between:csv.between,noofpoints:csv.noofpoints,simpDistWgt:csv.simpDistWeight,
origDistWeight:csv.origDistWeight}]->(t1)
```

The generated graph represents the enriched simplified trajectories using nodes and edges. Fig. 4.6 shown a sample trajectory graph in Neo4j.

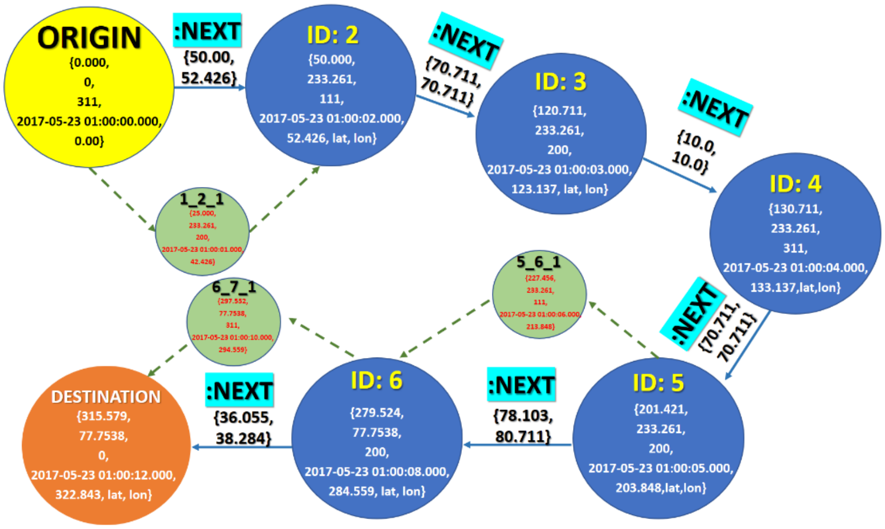


FIGURE 4.6: GENERATED GRAPH DATA SHOWING THE NODES AND EDGES

INTERMEDIATE nodes are labelled with the order of the corresponding point in the simplified geometry. The SIDE nodes are identified by an ID which follows the namingconvention:(INTERMEDIATE_NODE_ID)_(NEXT_INTERMEDIATE_NODE_ID)_(COUNTFor instance, the SIDE node with ID "5_6_1" denotes that is the first node lost during simplification between the intermediate nodes 5 and 6.

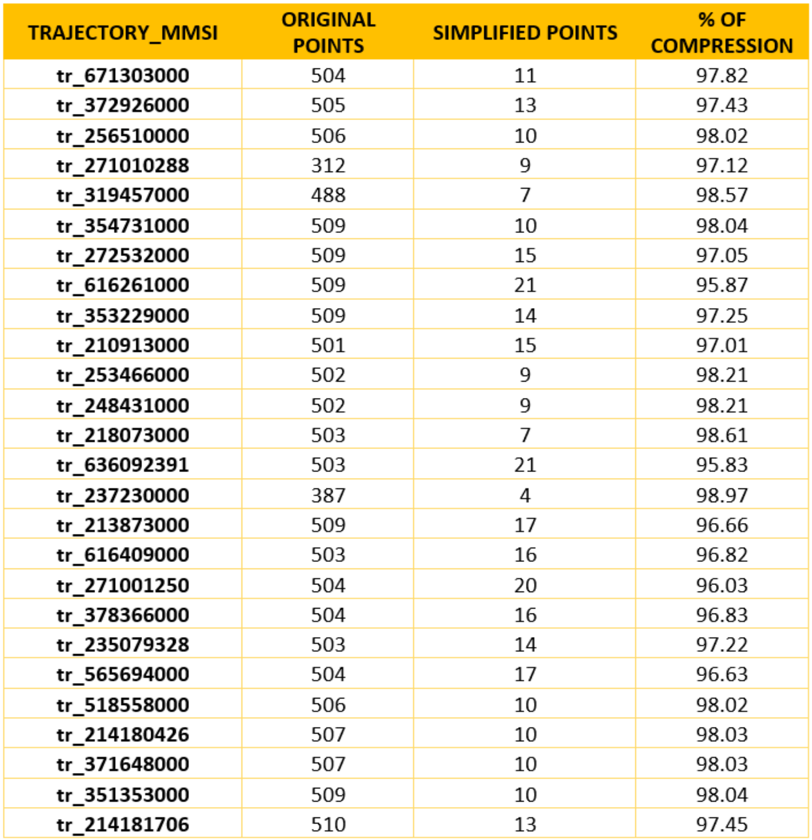
### 4.3.2 Spatial analysis using cypher

The uploaded trajectory data into Neo4j can be analyzed using cypher query language and Neo4j spatial procedures. Here is a list of examples queries that Neo4j can support:

1. Finding the shortest and longest trajectory based on:
    - Number of nodes
    - The original distance between origin and destination
    - The time difference between origin and destination
2. Identifying the overall collision points between trajectories:
    - Identifying the collision points at particular time interval

The SED based simplification was carried out over a dataset of vessel trajectories for August 2013 in the Aegean Sea as collected by the MarineTraffic Automatic Identification System (AIS) [*MarineTraffic* 2017] and the number of points retained after simplifying (with a threshold = 10km) is shown in Table.4.2. Individual trajectories are identified by "MMSI" which is unique for the moving vessel. From Fig. 4.7, it is evident that the simplification step has reduced the number of points in the original trajectories. Each simplified trajectory was associated with the SELF structure, modelled into the graph database, and used for further analysis. Trajectories with different number of mobility data points have been chosen for analysis in order for the set of features to be representative for a wide range of spatio-temporal characteristics.

TABLE 4.2: NUMBER OF POINTS IN THE ORIGINAL AND SIMPLIFIED TRAJECTORY



| TRAJECTORY_MMSI | ORIGINAL POINTS | SIMPLIFIED POINTS | % OF COMPRESSION |
|---|---|---|---|
| **tr_671303000** | 504 | 11 | 97.82 |
| **tr_372926000** | 505 | 13 | 97.43 |
| **tr_256510000** | 506 | 10 | 98.02 |
| **tr_271010288** | 312 | 9 | 97.12 |
| **tr_319457000** | 488 | 7 | 98.57 |
| **tr_354731000** | 509 | 10 | 98.04 |
| **tr_272532000** | 509 | 15 | 97.05 |
| **tr_616261000** | 509 | 21 | 95.87 |
| **tr_353229000** | 509 | 14 | 97.25 |
| **tr_210913000** | 501 | 15 | 97.01 |
| **tr_253466000** | 502 | 9 | 98.21 |
| **tr_248431000** | 502 | 9 | 98.21 |
| **tr_218073000** | 503 | 7 | 98.61 |
| **tr_636092391** | 503 | 21 | 95.83 |
| **tr_237230000** | 387 | 4 | 98.97 |
| **tr_213873000** | 509 | 17 | 96.66 |
| **tr_616409000** | 503 | 16 | 96.82 |
| **tr_271001250** | 504 | 20 | 96.03 |
| **tr_378366000** | 504 | 16 | 96.83 |
| **tr_235079328** | 503 | 14 | 97.22 |
| **tr_565694000** | 504 | 17 | 96.63 |
| **tr_518558000** | 506 | 10 | 98.02 |
| **tr_214180426** | 507 | 10 | 98.03 |
| **tr_371648000** | 507 | 10 | 98.03 |
| **tr_351353000** | 509 | 10 | 98.04 |
| **tr_214181706** | 510 | 13 | 97.45 |

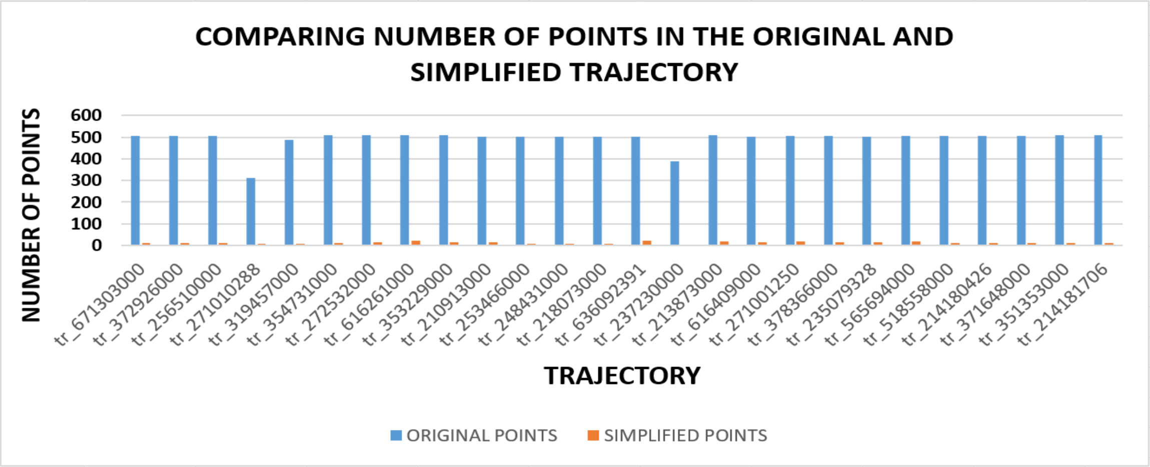


FIGURE 4.7: COMPARING NUMBER OF POINTS IN THE ORIGINAL AND SIMPLIFIED TRAJECTORY

#### 4.3.2.1 Finding shortest and longest trajectory

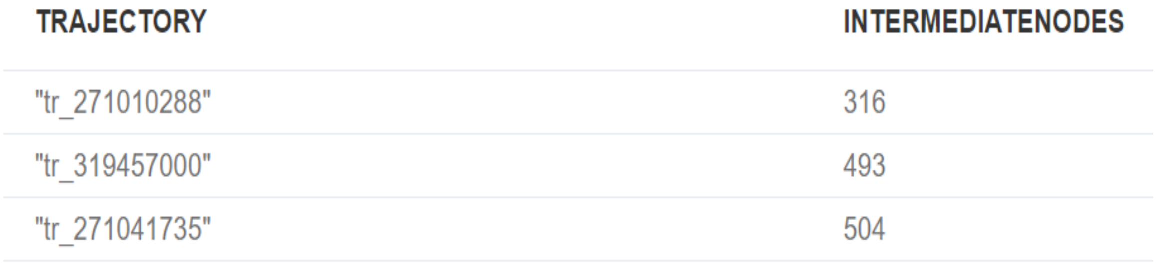
The path in the graph is defined as the sequence of nodes which are connected by weighted or non-weighted edges. In this model edges are weighted with both the original and simplified distance between consecutive simplified points. Three criteria have been considered in finding the longest and shortest trajectory. These are described next.

##### 4.3.2.1.1 Shortest/longest trajectory based on the number of nodes connected between origin and destination

The sequence of nodes which are connected between the origin and destination is counted to determine the length of the trajectory. This connectivity measurement is carried out on all the individual trajectories in the modelled dat. Then the trajectory with the highest number of intermediate nodes in the database is chosen as the longest and vice versa. The following cypher query counts the number of nodes in each trajectory and identifies three shortest trajectories based on the count in ascending order. The results are shown in Table 4.3.

```
FOREACH (simplified_trajectory IN graph)
MATCH (origin: simplified_trajectory)- [c: NEXT]->(destination: simplified_trajectory)
WHERE origin.id = destination.id
RETURN origin.name, COUNT(c)
ORDER BY COUNT(c) ASC LIMIT BY 3;
```

TABLE 4. 3: RESULTS OF LONGEST TRAJECTORIES BASED ON THE NUMBER OF NODES

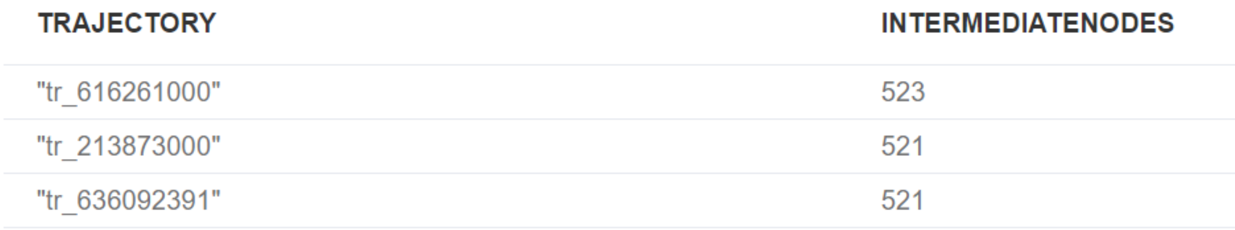


| TRAJECTORY | INTERMEDIATENODES |
|---|---|
| "tr_271010288" | 316 |
| "tr_319457000" | 493 |
| "tr_271041735" | 504 |

All queries have been executed on the data uploaded in section 4.3.1. The cypher query below counts the number of nodes in each trajectory and identifies the three longest trajectories by ordering the count in descending order. The results are shown in Table 4.4.

```
FOREACH (simplified_trajectory IN graph)
MATCH (origin: simplified_trajectory)-[c: NEXT]->(destination: simplified_trajectory)
WHERE origin.id = destination.id
RETURN origin.name, COUNT(c)
ORDER BY COUNT(c) DESC LIMIT BY 3;
```

TABLE 4.4: RESULTS OF LONGEST TRAJECTORIES BASED ON NUMBER OF NODES



| TRAJECTORY | INTERMEDIATENODES |
|---|---|
| "tr_616261000" | 523 |
| "tr_213873000" | 521 |
| "tr_636092391" | 521 |

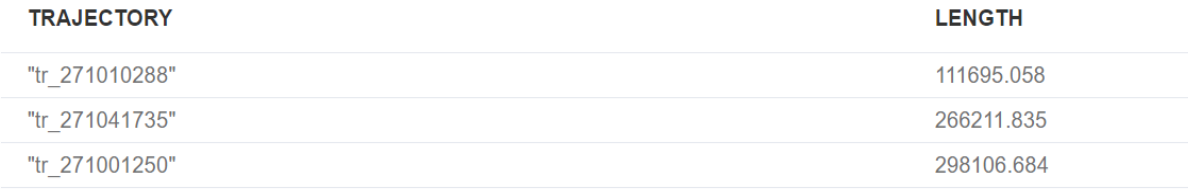
#### 4.3.2.1.2 Shortest/longest trajectory based on original distance between origin and destination

The accumulated distance between the sequence of nodes which are connected between the origin and destination determines the length of the trajectory. This geometry measurement is done on all the individual trajectories in the modelled data. Then, the trajectory with the longest distance is chosen as the longest and vice versa.

The below query sums up the original distance weight in all the edges for each trajectory to determine the original length of the trajectory. Then, it finds three shortest trajectories by ordering the calculated distance in ascending order. The results are shown in Table 4.5 match with the shortest trajectories identified using QGIS (Fig. 4.8).

```
FOREACH (simplified_trajectory IN graph)
MATCH (origin: simplified_trajectory)-[c: NEXT]->(destination: simplified_trajectory)
WHERE origin.id = destination.id AND origin.type = ‘intermediate’
RETURN origin.name, sum(toFloat(c.origDistWeight))
ORDER BY sum(toFloat(c.origDistWeight)) ASC
LIMIT 3;
```

TABLE 4.5: RESULTS OF SHORTEST TRAJECTORIES BASED ON ORIGINAL LENGTH



| TRAJECTORY | LENGTH |
|---|---|
| "tr_271010288" | 111695.058 |
| "tr_271041735" | 266211.835 |
| "tr_271001250" | 298106.684 |

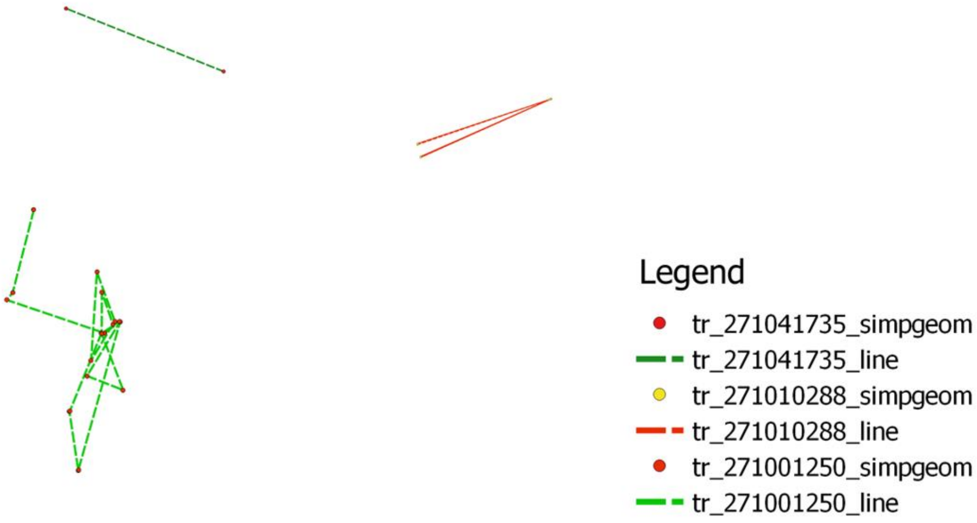
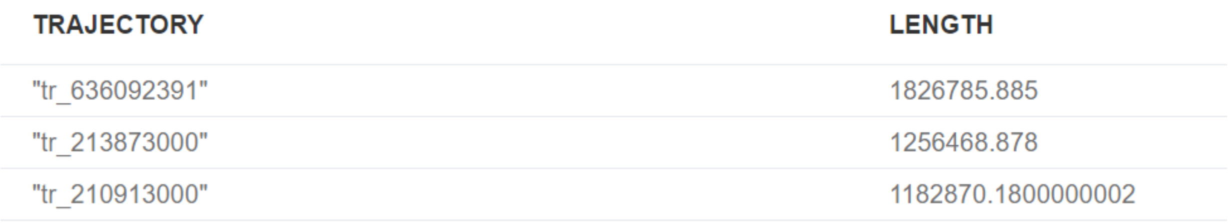


FIGURE 4.8: SHORTEST TRAJECTORIES IDENTIFIED IN QGIS BASED ON LENGTH

The below query extracts three longest trajectories by ordering the calculated distance in descending order. The results shown in Table 4.5 match with the longest trajectories identified using QGIS (Fig. 4.9).

*FOREACH (simplified_trajectory IN graph)*
*MATCH (origin: simplified_trajectory)-[c: NEXT]->(destination: simplified_trajectory)*
*WHERE origin.id = destination.id AND origin.type = 'intermediate'*
*RETURN origin.name, sum(toFloat(c.origDistWeight))*
*ORDER BY sum(toFloat(c.origDistWeight)) ASC*
*LIMIT 3;*

TABLE 4.6: RESULTS OF LONGEST TRAJECTORIES BASED ON THE LENGTH



| TRAJECTORY | LENGTH |
|---|---|
| "tr_636092391" | 1826785.885 |
| "tr_213873000" | 1256468.878 |
| "tr_210913000" | 1182870.1800000002 |

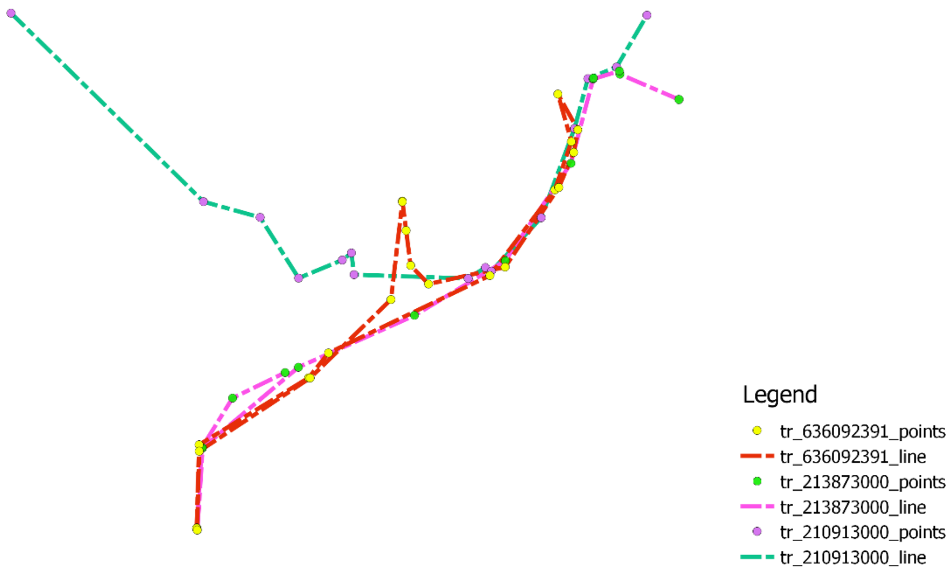


FIGURE 4.9: LONGEST TRAJECTORIES IDENTIFIED IN QGIS BASED ON LENGTH

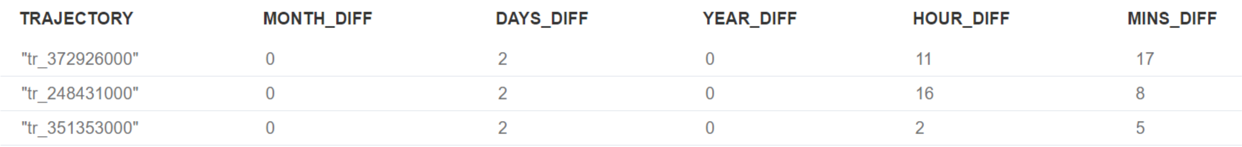
### 4.3.2.1.3 Shortest/longest trajectories based on time difference between origin and destination:

The time difference between the sequence of nodes which are connected between the origin and destination determines the total time duration of the trip. This temporal measurement is done on all the individual trajectories in the modelled data. Then, the trajectory with the higher time difference is chosen as the longest and vice versa.

The following query calculates the time difference between origin and destination for each trajectory to determine the total travel time of a trajectory. The time difference is then sorted in ascending order to find the three shortest trajectories. The results are shown in Table 4.7.

```
FOREACH (simplified_trajectory IN graph)
MATCH (origin: simplified_trajectory)-[c: NEXT]->(destination: simplified_trajectory)
WHERE origin.id = destination.id
RETURN origin.name, (destination.time – origin.time)
ORDER BY (destination.time – origin.time) ASC
LIMIT 3;
```

TABLE 4.7: RESULTS OF SHORTEST TRAJECTORIES BASED ON TIME DIFFERENCE

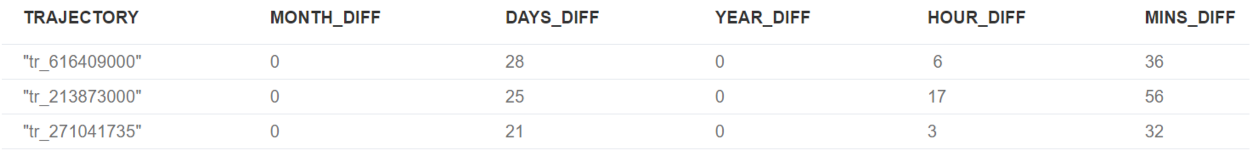


| TRAJECTORY | MONTH_DIFF | DAYS_DIFF | YEAR_DIFF | HOUR_DIFF | MINS_DIFF |
|---|---|---|---|---|---|
| "tr_372926000" | 0 | 2 | 0 | 11 | 17 |
| "tr_248431000" | 0 | 2 | 0 | 16 | 8 |
| "tr_351353000" | 0 | 2 | 0 | 2 | 5 |

The time difference is sorted in descending order to find the three longest trajectories. The results are shown in Table 4.8.

```
FOREACH (simplified_trajectory IN graph)
MATCH (origin: simplified_trajectory)-[c: NEXT]->(destination: simplified_trajectory)
WHERE origin.id = destination.id
RETURN origin.name, (destination.time – origin.time)
ORDER BY (destination.time – origin.time) DESC
LIMIT 3;
```

TABLE 4.8: RESULTS OF LONGEST TRAJECTORIES BASED ON TIME DIFFERENCE



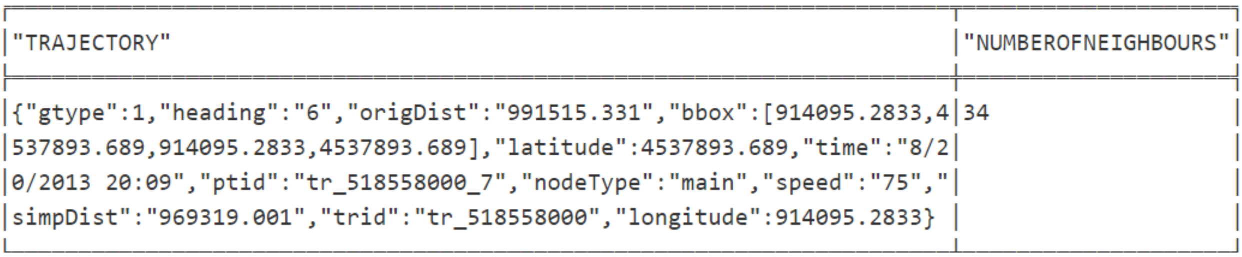
| TRAJECTORY | MONTH_DIFF | DAYS_DIFF | YEAR_DIFF | HOUR_DIFF | MINS_DIFF |
|---|---|---|---|---|---|
| "tr_616409000" | 0 | 28 | 0 | 6 | 36 |
| "tr_213873000" | 0 | 25 | 0 | 17 | 56 |
| "tr_271041735" | 0 | 21 | 0 | 3 | 32 |

### 4.3.2.2 Identifying the collision point

The number of points within a particular distance from a reference location helps in defining how close a point is to the rest of the points in the trajectory. Basically, this is similar to buffer analysis capability provided by most GIS software packages. In Cypher, using Neo4j Spatial plugin a similar kind of analysis can be carried out. The function *SPATIAL.CLOSEST* finds all geometry nodes in the graph within the distance to the given coordinate [*Neo4j Spatial Plugin* 2017]. The following cypher query finds the corresponding number of nodes within particular distance for each node and identifies the point which has maximum number of neighbours (Table. 4.9).

*MATCH (tr:TrajectoryNodes) WITH tr*
*CALL SPATIAL.CLOSEST('spatial_graph'',{latitude: tr.latitude, longitude: tr.longitude},10000)*
*YIELD node WHERE node.trid <> tr.trid*
*RETURN tr, COUNT(node) ORDER BY COUNT(node) DESC LIMIT BY 1;*

TABLE 4.9: THE POINT WHICH HAS MOST NUMBER OF NEIGHBORHOOD POINTS AROUND IT WITHIN 10 KILOMETERS



```
╒═════════════════════════════════════════════════════════════════════╤════════════════════╕
│"TRAJECTORY"                                                         │"NUMBEROFNEIGHBOURS"│
╞═════════════════════════════════════════════════════════════════════╪════════════════════╡
│{"gtype":1,"heading":"6","origDist":"991515.331","bbox":[914095.2833,4│34                  │
│537893.689,914095.2833,4537893.689],"latitude":4537893.689,"time":"8/2│                    │
│0/2013 20:09","ptid":"tr_518558000_7","nodeType":"main","speed":"75","│                    │
│simpDist":"969319.001","trid":"tr_518558000","longitude":914095.2833} │                    │
└─────────────────────────────────────────────────────────────────────┴────────────────────┘
```

Fig. 4.10 shows the plot of 10-kilometer buffer drawn around the query point using Quantum GIS.

The results from cypher query has been compared against the buffer operator in QGIS. The number of points falling within the buffer region (34) is same as the number of neighbours identified by Neo4j (Fig. 4.10).

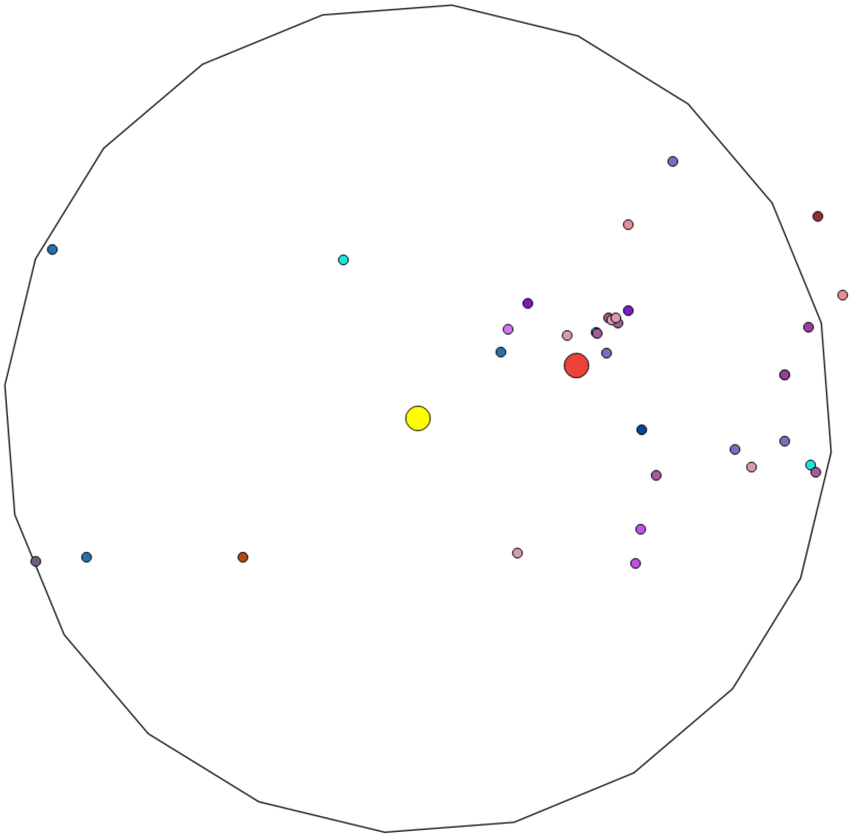
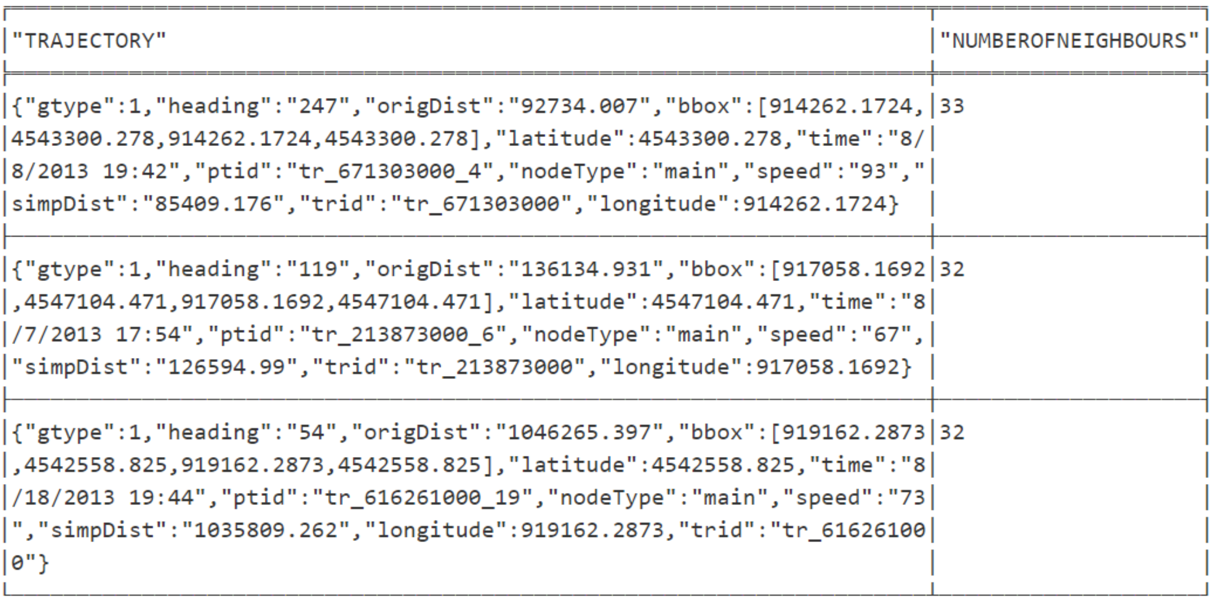


FIGURE 4.10: 10 KILOMETER BUFFER AROUND THE QUERY IN QGIS

By enforcing the temporal constraint, the above query can be extended to find the collision point for a given time period of a day. The following sample will identify the collision point between 4pm and 8pm (Table. 4.10).

```
MATCH (tr: TrajectoryNodes) WITH tr
CALL SPATIAL.CLOSEST(‘spatial_graph’’,{latitude: tr.latitude, longitude:
tr.longitude},10000)
YIELD node WHERE node.trid <> tr.trid  and hour>4pm and hour<8pm
RETURN tr, COUNT(node) ORDER BY COUNT(node) DESC LIMIT BY 1;
```

TABLE 4.10: THE TOP THREE COLLISION POINTS BETWEEN 4PM AND 8PM



```
╒═══════════════════════════════════════════════════════════════════════╤═════════════════════╕
│"TRAJECTORY"                                                           │"NUMBEROFNEIGHBOURS"│
╞═══════════════════════════════════════════════════════════════════════╪═════════════════════╡
│{"gtype":1,"heading":"247","origDist":"92734.007","bbox":[914262.1724,│33                   │
│4543300.278,914262.1724,4543300.278],"latitude":4543300.278,"time":"8/│                     │
│8/2013 19:42","ptid":"tr_671303000_4","nodeType":"main","speed":"93","│                     │
│simpDist":"85409.176","trid":"tr_671303000","longitude":914262.1724}  │                     │
├───────────────────────────────────────────────────────────────────────┼─────────────────────┤
│{"gtype":1,"heading":"119","origDist":"136134.931","bbox":[917058.1692│32                   │
│,4547104.471,917058.1692,4547104.471],"latitude":4547104.471,"time":"8│                     │
│/7/2013 17:54","ptid":"tr_213873000_6","nodeType":"main","speed":"67",│                     │
│"simpDist":"126594.99","trid":"tr_213873000","longitude":917058.1692} │                     │
├───────────────────────────────────────────────────────────────────────┼─────────────────────┤
│{"gtype":1,"heading":"54","origDist":"1046265.397","bbox":[919162.2873│32                   │
│,4542558.825,919162.2873,4542558.825],"latitude":4542558.825,"time":"8│                     │
│/18/2013 19:44","ptid":"tr_616261000_19","nodeType":"main","speed":"73│                     │
│","simpDist":"1035809.262","longitude":919162.2873,"trid":"tr_61626100│                     │
│0"}                                                                    │                     │
└───────────────────────────────────────────────────────────────────────┴─────────────────────┘
```

## 4.4 Visualization tool for semantic interpolation

A web based graph data visualization system has been developed for performing semantic interpolation from the proposed model. The dynamic system allows the user to choose the trajectory on which the semantic retrieval will be performed. The graph visualization capabilities have been added using the JavaScript framework “alchemy.js”. Alchemy is a graph visualization tool for developing web applications. It is easily customizable and includes the capabilities like clustering, filtering and embedding graphs [*Alchemy* 2017].

The visualization architecture is shown in Fig. 4.11.

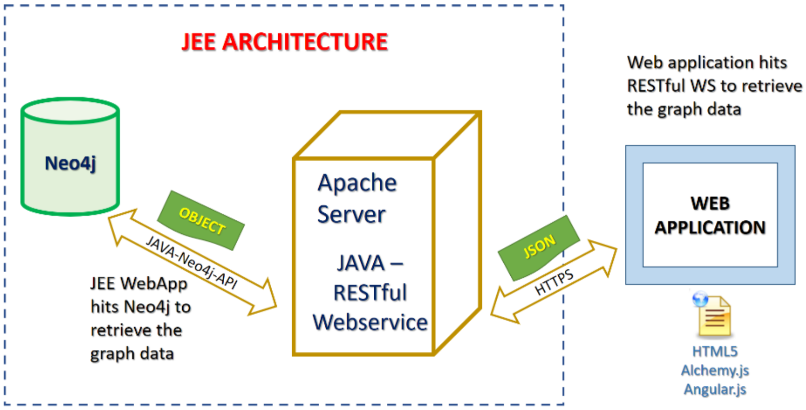
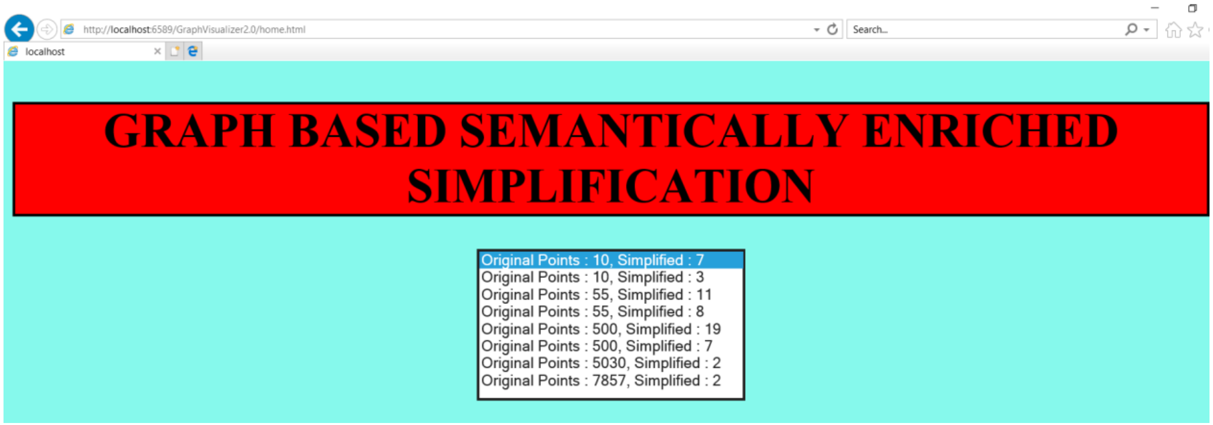

JEE ARCHITECTURE
Neo4j
OBJECT
JAVA-Neo4j-API
JEE WebApp hits Neo4j to retrieve the graph data
Apache Server
JAVA – RESTful Webservice
JSON
HTTPS
Web application hits RESTful WS to retrieve the graph data
WEB APPLICATION
HTML5
Alchemy.js
Angular.js

FIGURE 4.11: VISUALIZATION SYSTEM ARCHITECTURE

An HTML powered web application allows the user to select a single trajectory from a collection of trajectories. The browser then sends the corresponding https request to RESTful Webservice through Angular client [*Angular* 2017]. The request is then parsed by Apache server before it hits Neo4j database for retrieving the corresponding graph as an object. Java-Neo4j-API lets the java program communicate with Neo4j database. The response is a JSON object that is then parsed by Alchemy to display the graph data on the browser.  The following snapshots summarize the functionality of the developed system. The user can choose a single trajectory from the dropdown list of trajectories (Fig. 4.12).

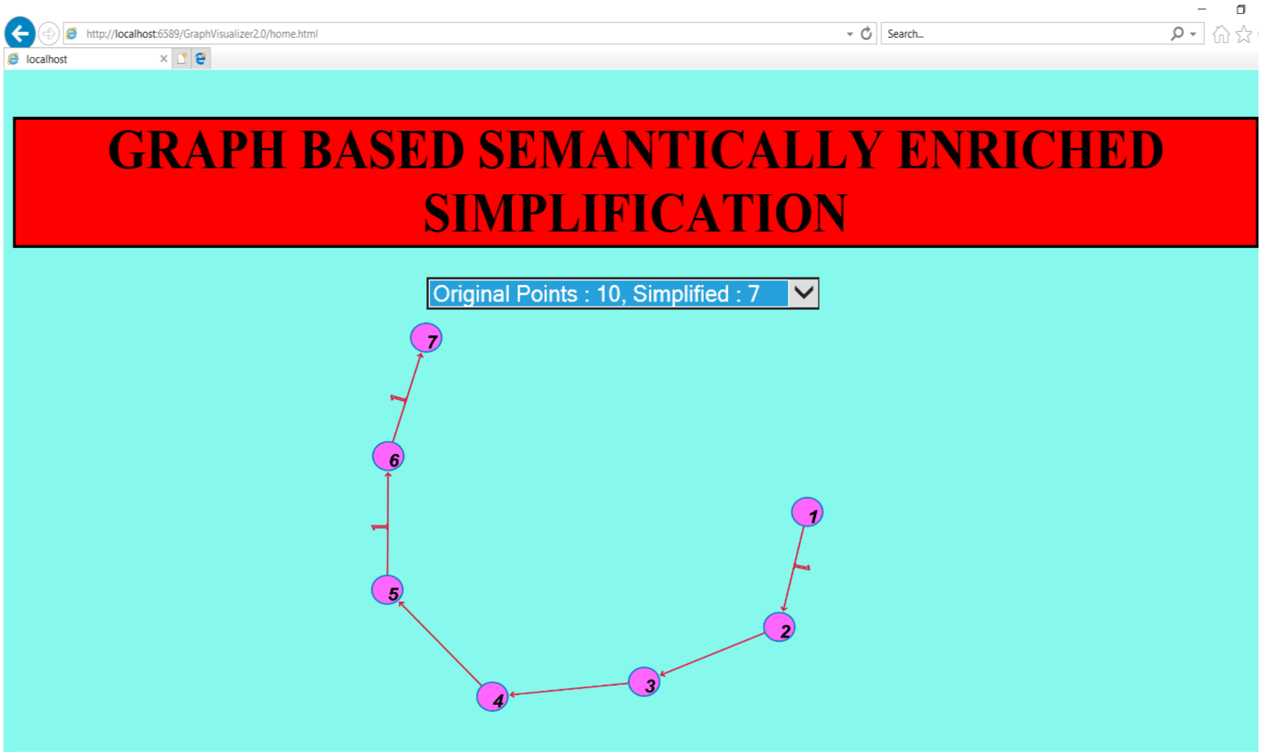


GRAPH BASED SEMANTICALLY ENRICHED SIMPLIFICATION

Original Points : 10, Simplified : 7
Original Points : 10, Simplified : 3
Original Points : 55, Simplified : 11
Original Points : 55, Simplified : 8
Original Points : 500, Simplified : 19
Original Points : 500, Simplified : 7
Original Points : 5030, Simplified : 2
Original Points : 7857, Simplified : 2

FIGURE 4.12: DROPDOWN LIST SHOWING ALL THE EXISTING TRAJECTORIES

After a trajectory is selected by the user, the corresponding graph data can be visualized on the browser screen (Fig. 4.13).



GRAPH BASED SEMANTICALLY ENRICHED SIMPLIFICATION

Original Points : 10, Simplified : 7

FIGURE 4.13: SELECTED SIMPLIFIED TRAJECTORY ON THE BROWSER SCREEN

Once the database responds with proper data, the object received by Apache server is sent to alchemy.js as a JSON (JavaScript Object Notation) object. Alchemy framework enables the browser to parse the JSON object. Numeric value on the edges denotes the number of points eliminated during the simplification between those edge nodes.

In this example between node 6 and 7 there was one point which is lost during the simplification of the trajectory. If the user clicks on that number, the corresponding node (“SIDE”) which has been lost will also be displayed (Fig. 4.14).

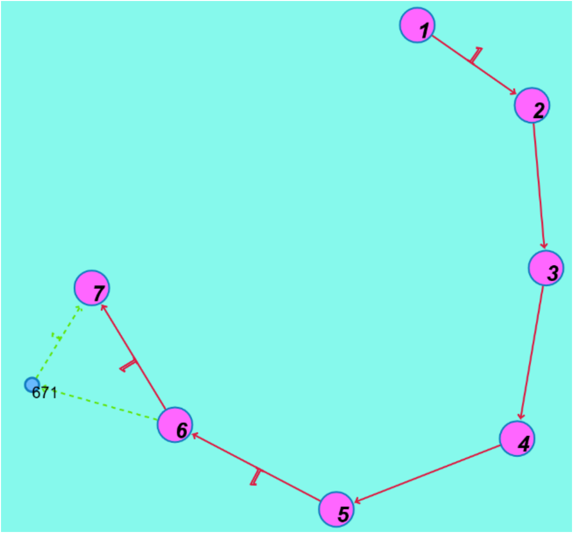
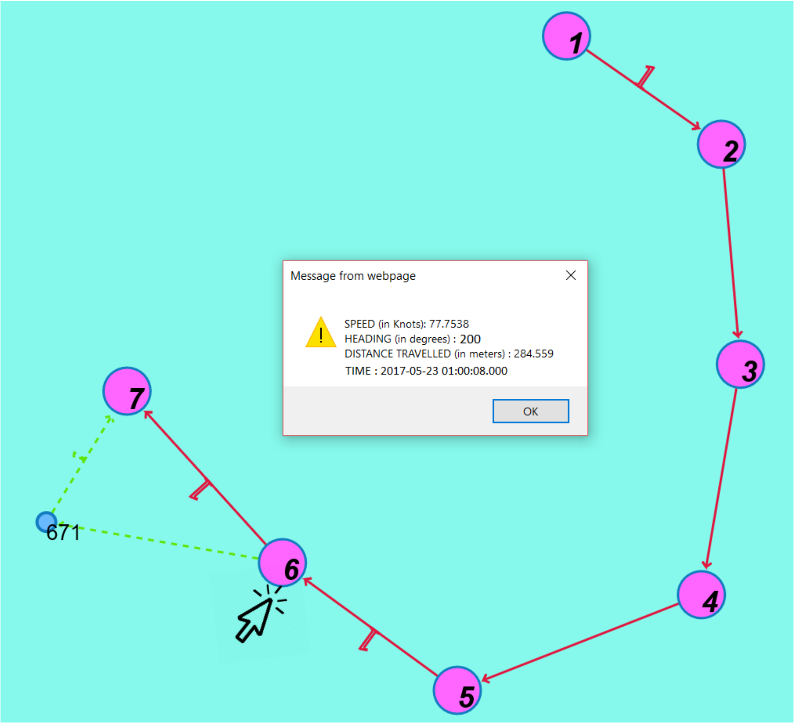


FIGURE 4.14: GRAPH SHOWING THE SIDE NODE WHICH IS LOST DURING THE SIMPLIFICATION

The nodes in pink circles connected by red colored edges represent "INTERMEDIATE" points, while the nodes in green connected by dotted green edges represent "SIDE" points. **T**he user can click on any node to retrieve its corresponding semantics, such as speed, heading, distance travelled and time of crossing (Fig. 4.15).



SPEED (in Knots): 77.7538
HEADING (in degrees) : 200
DISTANCE TRAVELLED (in meters) : 284.559
TIME : 2017-05-23 01:00:08.000

FIGURE 4.15: ALERT BOX WITH THE SEMANTICS FOR THE CLICKED POINT

## 4.5 Conclusions

This paper proposed a graph data model to represent simplified trajectories that preserve both the spatial and temporal characteristics of their original versions. The model has been implemented in Neo4j using Java programming language.  Simplified trajectories can be analyzed using Cypher query language and Neo4j spatial procedures. The developed visualization tool helps the user to perform semantic interpolation at different levels of simplification.

Currently, Neo4j spatial plugin has limited set of spatial procedures. The system can be made powerful in trajectory data management after more spatial procedures are integrated into Neo4j graph database [*Stefanakis* 2017]. Both the graph model and developed visualization framework can be applied to other application domains such as bus transit and metro systems.

### Acknowledgements

This research has received funding from the Natural Sciences and Engineering Research Council of Canada (NSERC), Discovery Grants Program.

# REFERENCES

Alchemy JavaScript., http://graphalchemist.github.io/Alchemy/#/, Accessed on : 12th September 2017

Angular JavaScript., https://angularjs.org, Accessed on : 12th September 2017

Alvares, L.O., et al., 2007. A model for enriching trajectories with semantic geographical information. In: *The Proceedings of the 15th annual ACM international symposium on advances in geographic information systems*, 7–9 November, Seattle, WA. Article No. 22.

Cromley, R.G., 1991. Hierarchical methods of line simplification. *Cartography and Geographic Information Science*, 18 (2), 125–131. doi:10.1559/152304091783805563

Gurfraind.A, Genkin.M, A graph database framework for covert network analysis: An application to the Islamic State network in Europe. *World Development,* 2016 September, Pages 1-11 doi: 10.1016/j.socnet.2016.10.004

Hadi Hajari, Farshad Hakimpour., 2014 A Spatial Data Model For Moving Object Databases, International Journal of Database Management Systems ( IJDMS ) Vol.6, No.1, February 2014, doi : 10.5121/ijdms.2013.6101 1

K. Buchin, M. Buchin, and J. Gudmundsson. Detecting single file movement. In *GIS '08: Proceedings of the 16thACM SIGSPATIAL international conference on Advances in geographic information systems,* pages 1-10, New York, NY, USA, 2008. ACM.

Keates, J.S., 1989. *Cartographic design and production*. 2nd ed. Essex: Longman.

MarineTraffic, 2017. Live-ships map: AIS [online]. Available from: http://www.marinetraffic.com/ais/ [Last visited, June 27, 2017]

Meratnia, N. and de By, R.A., 2004. Spatiotemporal compression techniques for moving point objects. In: *Proceedings of the international conference on extending database technology* (EDBT). Berlin: Springer, 765–782. LNCS 2992.

Neo4j Spatial Plugin, http://neo4j-contrib.github.io/spatial/0.24-neo4j-3.1/index.html, Accessed on: 17th August 2017

Neo4j Graph Database, https://neo4j.com/, Accessed on : 1st August 2017

Parent, C. et al., 2013. Semantic Trajectories Modeling and Analysis. *ACM Computing Surveys, Vol. 45, No.4, Article 42.* doi: http://dx.doi.org/10.1145/2501654.2501656

Partl.C, Gratzl.S, Streit.M , Wassermann.A.M, Pfister.H, Schmalstieg.D, and Lex.A , 2016 Pathfinder: Visual Analysis of Paths in Graphs *Eurographics Conference on Visualization (EuroVis)* 2016 Volume 35 (2016), Number 3

Przemysław Idziaszek, Wojciech Mueller, Janina Rudowicz-Nawrocka,Michał Gruszczy´ nski, Sebastian Kujawa, Karolina Górna, Kinga Balcerzak. 2016 Visualisation of Relational Database Structure by Graph Database *CMST 22(4)* 217-224 doi:10.12921/cmst.2016.0000014

QiuLei Guo and Hassan A. Karimi, 2016. A Topology-Inferred Graph-Based Heuristic Algorithm for Map Simplification. In: *Transactions in GIS,* doi:10.1111/tgis.12188

Richter, K.F., Schmid, F., and Laube, P., 2012. Semantic trajectory compression: representing urban movement in a nutshell. *Journal of Spatial Information Science,* (4), 3–30.

Robinson, Joel L. Morrison, Phillip C. Muehrcke, A. Jon Kimerling, Stephen C. Guptill, Elements of Cartography, 6th Edition ISBN: 978-0-471-55579-7

Shahriari, N., and V. Tao, 2002. Minimizing Positional Errors in Line Simplification Us-ing Adaptive Tolerance Values. In: *Symposium on Geospatial Theory, Processing and Applica-tion,* 4(3), 213-223.

Stefanakis, E., 2017. Graph Databases – Recent development in Neo4j may help accommodate the Geospatial Community. *GoGeomatics. Magazine of GoGeomatics Canada*. January 2017.

Spaccapietra, S., et al., 2008. A conceptual view on trajectories. *Data & Knowledge Engineering,* 65 (1), 126–146. Retrieved from : http://www.sciencedirect.com/science/article/pii/S0169023X07002078

Stefanakis, E., 2012. Trajectory generalization under space constraints. In: *The proceedings of the 7th international conference on geographic information science (GIScience 2012),* 18–21 September, Columbus, OH.

Stefanakis, E., 2015. SELF: Semantically enriched Line simpliFication. In: *International Journal of Geographical Information Science,* Vol. 29, Iss. 10, 2015, Pages 1826-1844 doi: 10.1080/13658816.2015.1053092

Tamilmani R, Stefanakis E, 2017. Enriched geometric simplification of linear features. *Geomatica Vol. 71, No.1, 2017, pp. 3 to 19*. doi: dx.doi.org/10.5623/cig2017-101

Tienaah, T., Stefanakis, E., and Coleman, D., 2015.S Contextual Douglas-Peucker simplification. *Geomatica Journal*, 69 (3), 327-338, https://doi.org/10.5623/cig2015-306

Weibel, R., 1996. A typology of constraints to line simplification. In: *Proceedings of 7th international symposium on spatial data handling*, 12–16 August, Delft. IGU, 533–546.

Weibel, R., 1997. Generalization of spatial data: principles and selected algorithms. In: M. Kreveld, et al., eds. *Algorithmic foundations of geographic information systems*. Berlin: Springer, 99–152.

Wu, et alShil – Ting and Mercedes Rocio Gonzales Marquez 2003. *Proceedings of the XVI Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI’03)* 1530-1834/03 doi :10.1109/SIBGRA.2003.1240992

Yan, Z., et al. 2011. SeMiTri: A framework for semantic annotation of heterogeneous trajectories. *In: The proceedings of the 14th international conference on Extending Database Technology* (EDBT 2011). New York: ACM

# 5. Conclusion and Recommendations

The primary purpose of this research was to retain the geometric (length) and semantic attributes associated with individual locations of linear features and trajectories in their simplified representation at various level of detail. Efficient modelling, analysis, and visualization methods were developed. The research has contributed to PostgreSQL/PostGIS, an open source spatial database system.

## 5.1 Summary of Research

The preliminary steps of this research were to: (i) investigate the existing simplification techniques; and (ii) to test the efficiency of SELF data structure in regard to semantic interpolation at different levels of simplification.

Chapter 2 focused on implementing the SELF (Semantically Enriched Line simpliFication) data structure to preserve the geometric characteristics associated to the original linear features. The data structure has been implemented in PostgreSQL 9.4 with PostGIS extension using PL/pgSQL to support static and non-functional polylines and tested with both synthetic and real world features.

The objective of Chapter 3 was to implement and test the SELF data structure for semantically enriched simplification of trajectories. The implemented method combines a Synchronous Euclidean Distance (SED) based simplification algorithm and SELF (Semantically Enriched Line simpliFication) data structure to preserve the semantics associated with the original trajectories (spatio temporal lines). This resulted in an enriched library of PL/pgSQL functions to support the simplification of both static and dynamic linear features.

Chapter 4 proposed a graph data model to represent and analyze simplified trajectories that preserve both the spatial and temporal characteristics of the original versions. Also, a visualization framework for trajectories in graph form has been developed.

## 5.2 Achievements of Research

The purpose of this research is to retain the semantic and geometric attributes associated with individual locations of original linear features and trajectories in their simplified versions. This has been accomplished by enriching the representation of the simplified lines with an array of values corresponding to multiple locations along the original lines.  To this end, a graph model to represent the simplified geometry of trajectories along with their semantics has been proposed for the trajectory data analysis and visualization purpose.

- SELF structure for static lines applies two kinds of compression: point level and segment level. The segment level compression eliminates entire segments (continuous points) which has the segment level compression ratio within the user-defined threshold, while point level compression discards only certain points which are within the point level threshold. The results of the experiments indicate that the different topological complexity of the datasets play a major role in distance interpolation error.
- The comparison results between SED-DP simplification and DP simplification indicated that SED-DP simplification always retains more number of points which are more significant in forming the trajectory than other points as they better convey the trajectory characteristics for a particular context.

- SELF structure for trajectories applies two kinds of semantic based reduction: speed based and heading based. Both compression techniques produce the same amount error in the interpolated semantics. However, the results of the experiments indicate that the different spatio-temporal characteristics of the datasets play a major role in the semantic interpolation error.
- The proposed graph model has been implemented in Neo4j using Java programming language. Thus, simplified trajectories can be analyzed using Cypher query language and Neo4j spatial procedures. The developed visualization tool helps the user to perform semantics retrieval at different levels of simplification.

## 5.3 Recommendations for Future Work

This research was primarily started with alleviating the problem of semantic loss during the process of simplification. But during the course of this research, additional research possibilities are identified, and are as follows:

- Future recommendations include recoding this library to other programming languages (such as Python) so that it can be embedded into other commercial or open source GIS software packages.
- More emphasis on evaluating the time complexity of the implemented algorithms with various compression levels being applied to the SELF structure will make the running time optimal.
- Currently, Neo4j spatial plugin has limited set of spatial procedures. This system can be made powerful in trajectory data management by integrating more spatial procedures into Neo4j graph [*Stefanakis* 2017].

- Extending the graph model for performing trajectory similarity measures would be another possible research area.
- Lastly, the extension of the visualization framework will facilitate the adoption of the SELF structure in various application domains with need for semantically enhanced multiscale representation of linear features and trajectories. Both the graph model and developed visualization framework can be applied to application domains such as bus transit and metro systems.

## 5.4 Conclusions

Overall, the primary purpose of this research to implement and test SELF data structure for linear features and trajectories was accomplished to alleviate the problem in losing the geometric and semantic attributes associated with the intermediate points on the original geometries. The experimental results have shown that segment level compression produces the error higher than point level compression. But, segment level compression algorithm has less time complexity than point level compression. With the experimental results from static linear features the SELF structure has been extended to support spatio temporal lines. That resulted in an enriched library of PL/pgSQL function to support the simplification of both static and dynamic lines.

Problem with the traditional simplification algorithms identified as, they utilize the distance offset as a criterion to eliminate the redundant points. Temporal dimension in trajectories have been considered in retaining the points to convey both the spatial and temporal characteristics of the trajectory. SED based trajectory simplification technique to consider spatio-temporal data in trajectory generalization has implemented. The comparison results between SED-DP simplification and DP simplification indicated that

SED-DP simplification always retains more number of points which are more significant in forming the trajectory than other points as they better convey the trajectory characteristics for a particular context.

Visualization of the massive trajectory dataset becomes difficult to handle as the millions of raw data points make the processing complex. The proposed graph model for combining the simplified geometry of trajectories and SELF data structure has been significant in performing useful trajectory data analysis on the modelled data using Cypher query language and Neo4j Spatial procedures. The visualization tool developed on top of Neo4j provide a useful functionality that can be extended to support different application domains.

In conclusion, the research provides the way of annotating the simplified geometry with the semantics by means of SELF structure that can retain the original semantics as an array of values followed by the graph model and visualization tool for performing useful trajectory data analysis.

## REFERENCES

Stefanakis, E., 2015. SELF: Semantically enriched Line simpliFication. In: *International Journal of Geographical Information Science,* Vol. 29, Iss. 10, 2015, Pages 1826-1844 doi: 10.1080/13658816.2015.1053092

Stefanakis, E., 2017. Graph Databases – Recent development in Neo4j may help accommodate the Geospatial Community. *GoGeomatics. Magazine of GoGeomatics Canada*. January 201

# Appendix I

## 1. SELF structure for static lines – PL/pgSQL function (Chapter 2)

```
/* Function Definition – Takes the input as original
geometry, simplification threshold and compression ratios*/

CREATE OR REPLACE FUNCTION SELF_ADV_CB(line geometry, threshold double
precision,thr_ratio numeric,comp_ratio double precision)
RETURNS SELFAdv AS $$
DECLARE
/* Defining the variables and data types*/

self_adv SELFAdv;
no_lines int;
no_simp_points int;
simp_segments geometry [];
func_lines int;
each_simp_line int;
StartingPoint geometry;
EndingPoint geometry;
ActualLength numeric;
pp_point geometry;
pp_point_nxt geometry;
array_count int;
no_of_points int;
count int;
prv_org_dist numeric;
prv_sim_dist numeric;
SELF_LEN  text[];
ratio numeric;
orig_line_dist numeric;
simp_line_dist numeric;
orig_dist numeric;
sim_dist numeric;
slp_diff_array double precision[];
comp_ratio_array double precision[];
BEGIN
each_simp_line = 1;
array_count = 1;
count = 1;
prv_org_dist = 0.0;
prv_sim_dist = 0.0;
no_lines = array_length(SELF_NS(line),1);
no_simp_points = ST_NPOINTS(ST_LINEMERGE(ST_SIMPLIFY(line,threshold)));
/* Checking for line type (single/multi segments) based on
the number of points on the lines*/
      IF no_simp_points=2 THEN
            self_adv = SELF_ADV(line, threshold,comp_ratio);

      ELSE

/* Simplify the geometry based on Douglas Peucker algorithm
*/
```

```
      StartingPoint=ST_StartPoint(ST_LineMerge(ST_Simplify(line,thresho
ld)));

      EndingPoint=ST_EndPoint(ST_LineMerge(ST_Simplify(line,threshold))
);

/* Getting starting and ending points of the line*/
      ActualLength=ST_Length(line);
            self_adv.StartingPoint =  StartingPoint;
            self_adv.EndPoint = EndingPoint;
            self_adv.ActualLength = round(ActualLength,3);
            simp_segments = SELF_ADV_CASE3(line, threshold);
            func_lines = array_length(simp_segments,1);
/* Applying point and segment level threshold*/
      WHILE each_simp_line <= func_lines
                LOOP
            comp_ratio_array =
SELF_COMP_RATIO(simp_segments[each_simp_line]);
                slp_diff_array =
SELF_SLP_DIFF(simp_segments[each_simp_line]);
                IF each_simp_line = func_lines THEN

                    orig_line_dist =
ST_LENGTH(simp_segments[each_simp_line]);
                simp_line_dist =
ST_DISTANCE(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])));
                    ratio = (orig_line_dist-
simp_line_dist)/(orig_line_dist);
                    ratio = ratio * 100;
                    IF ratio >= thr_ratio THEN
                  no_of_points =
ST_NPOINTS(simp_segments[each_simp_line]);
/* Finding orthogonal projection of each point on the
original line into simplified*/
                WHILE array_count < (no_of_points-1)
                LOOP
                      IF array_count=1 THEN
                           pp_point =
SELF_PP_POINT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])
),ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_PointN(ST_LINEMERGE(simp_segments[each_simp_line]),array_count));
                           pp_point_nxt =
SELF_PP_POINT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])
),ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_PointN(ST_LINEMERGE(simp_segments[each_simp_line]),array_count+1));

/* Calculating simplified and original distance */

orig_dist =
ST_DISTANCE(ST_PointN(ST_LINEMERGE(simp_segments[each_simp_line]),array
_count),ST_PointN(ST_LINEMERGE(simp_segments[each_simp_line]),array_cou
nt+1)) + prv_org_dist;
                           IF slp_diff_array[array_count] != 0.0 and
comp_ratio_array[array_count] >= comp_ratio  THEN
```

```
                                 IF
SELF_CHK_PT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),pp_point_nxt)
!= 3.0 THEN
                                       IF
SELF_CHK_PT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),pp_point_nxt)
= 0.0 THEN
                                            sim_dist =
ST_DISTANCE(pp_point,pp_point_nxt) + prv_sim_dist ;
                                          SELF_LEN[count] =
round(sim_dist,3) || ',' || round(orig_dist,3);
                                          count = count+1;
                                                   ELSIF
SELF_CHK_PT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),pp_point_nxt)
= 1.0 THEN
                                                        sim_dist =
ST_DISTANCE(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line]))) +
prv_sim_dist ;
                                          SELF_LEN[count] =
round(sim_dist,3) || ',' || round(orig_dist,3);
                                          count = count+1;
                                                   ELSIF
SELF_CHK_PT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),pp_point_nxt)
= 2.0 THEN
                                                        sim_dist = 0.0
+ prv_sim_dist ;
                                          SELF_LEN[count] =
round(sim_dist,3) || ',' || round(orig_dist,3);
                                          count = count+1;
                                       END IF;
                                 END IF;
                           END IF;
                      ELSE
                           pp_point =
SELF_PP_POINT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])
),ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_PointN(ST_LINEMERGE(simp_segments[each_simp_line]),array_count));
                           pp_point_nxt =
SELF_PP_POINT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])
),ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_PointN(ST_LINEMERGE(simp_segments[each_simp_line]),array_count+1));
                           orig_dist =
ST_DISTANCE(ST_PointN(ST_LINEMERGE(simp_segments[each_simp_line]),array
_count),ST_PointN(ST_LINEMERGE(simp_segments[each_simp_line]),array_cou
nt+1))+orig_dist;
```

/* Check if the projected point is on the simplified line or not */

```
IF slp_diff_array[array_count] != 0.0 and comp_ratio_array[array_count]
>= comp_ratio  THEN
                              IF
```

```
SELF_CHK_PT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),pp_point_nxt)
!= 3.0 THEN
                                       IF
SELF_CHK_PT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),pp_point_nxt)
= 0.0 THEN
                                            sim_dist =
ST_DISTANCE(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
pp_point_nxt) + prv_sim_dist ;
                                          SELF_LEN[count] =
round(sim_dist,3) || ',' || round(orig_dist,3);
                                          count = count+1;
                                                   ELSIF
SELF_CHK_PT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),pp_point_nxt)
= 1.0 THEN
                                                        sim_dist =
ST_DISTANCE(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line]))) +
prv_sim_dist ;
                                          SELF_LEN[count] =
round(sim_dist,3) || ',' || round(orig_dist,3);
                                          count = count+1;
                                                   ELSIF
SELF_CHK_PT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),pp_point_nxt)
= 2.0 THEN
                                                        sim_dist = 0.0
+ prv_sim_dist ;
                                          SELF_LEN[count] =
round(sim_dist,3) || ',' || round(orig_dist,3);
                                          count = count+1;
                                       END IF;
                              END IF;
                           END IF;
                      END IF;
                      array_count = array_count+1;
                END LOOP;
                    END IF;

                prv_org_dist = ST_LENGTH(simp_segments[each_simp_line])
+  prv_org_dist;
```

**/* Calculating the simplified and original accumulated length*/**

```
                prv_sim_dist =
ST_DISTANCE(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line]))) +
prv_sim_dist;
                array_count = 1;
                    sim_dist = 0.0;
                    orig_dist = 0.0;
                each_simp_line = each_simp_line+1;
            ELSE
                orig_line_dist =
ST_LENGTH(simp_segments[each_simp_line]);
                simp_line_dist =
ST_DISTANCE(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])));
                    ratio = (orig_line_dist-
simp_line_dist)/(orig_line_dist);
                    ratio = ratio * 100;
                    IF ratio >= thr_ratio THEN
                  no_of_points =
ST_NPOINTS(simp_segments[each_simp_line]);
                WHILE array_count < no_of_points
                LOOP
                      IF array_count=1 THEN
                           pp_point =
SELF_PP_POINT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])
),ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_PointN(ST_LINEMERGE(simp_segments[each_simp_line]),array_count));
                           pp_point_nxt =
SELF_PP_POINT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])
),ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_PointN(ST_LINEMERGE(simp_segments[each_simp_line]),array_count+1));
                           orig_dist =
ST_DISTANCE(ST_PointN(ST_LINEMERGE(simp_segments[each_simp_line]),array
_count),ST_PointN(ST_LINEMERGE(simp_segments[each_simp_line]),array_cou
nt+1)) + prv_org_dist;
                           IF slp_diff_array[array_count] != 0.0  and
comp_ratio_array[array_count] >= comp_ratio THEN
                              IF
SELF_CHK_PT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),pp_point_nxt)
!= 3.0 THEN
                                       IF
SELF_CHK_PT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),pp_point_nxt)
= 0.0 THEN
                                            sim_dist =
ST_DISTANCE(pp_point,pp_point_nxt) + prv_sim_dist ;
                                          SELF_LEN[count] =
round(sim_dist,3) || ',' || round(orig_dist,3);
                                          count = count+1;

ELSIF
SELF_CHK_PT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),pp_point_nxt)
= 1.0 THEN
                                                        sim_dist =
ST_DISTANCE(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])))+ prv_sim_dist
;
                                          SELF_LEN[count] =
round(sim_dist,3) || ',' || round(orig_dist,3);
                                          count = count+1;
                                                   ELSIF
SELF_CHK_PT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),pp_point_nxt)
= 2.0 THEN
                                                        sim_dist = 0.0
```

```
+ prv_sim_dist ;
                                          SELF_LEN[count] = 
round(sim_dist,3) || ',' || round(orig_dist,3);
                                          count = count+1;
                                       END IF;
                               END IF;
                            END IF;
                         ELSE
                           pp_point = 
SELF_PP_POINT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])
),ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_PointN(ST_LINEMERGE(simp_segments[each_simp_line]),array_count));
                           pp_point_nxt = 
SELF_PP_POINT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])
),ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_PointN(ST_LINEMERGE(simp_segments[each_simp_line]),array_count+1));
                           orig_dist = 
ST_DISTANCE(ST_PointN(ST_LINEMERGE(simp_segments[each_simp_line]),array
_count),ST_PointN(ST_LINEMERGE(simp_segments[each_simp_line]),array_cou
nt+1))+orig_dist;
                           IF slp_diff_array[array_count] != 0.0 and 
comp_ratio_array[array_count] >= comp_ratio  THEN
                              IF 
SELF_CHK_PT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),pp_point_nxt) 
!= 3.0 THEN

/* Ends the iteration if the cursor reaches last point of 
the line */
IF 
SELF_CHK_PT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),pp_point_nxt) 
= 0.0 THEN
                                            sim_dist = 
ST_DISTANCE(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
pp_point_nxt) + prv_sim_dist ;
                                          SELF_LEN[count] = 
round(sim_dist,3) || ',' || round(orig_dist,3);
                                          count = count+1;
                                                   ELSIF 
SELF_CHK_PT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),pp_point_nxt) 
= 1.0 THEN
                                                        sim_dist = 
ST_DISTANCE(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])))+ prv_sim_dist
;
                                          SELF_LEN[count] = 
round(sim_dist,3) || ',' || round(orig_dist,3);
                                          count = count+1;
                                                   ELSIF 
SELF_CHK_PT(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])),
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line])),pp_point_nxt) 
= 2.0 THEN
                                                        sim_dist = 0.0 
+ prv_sim_dist ;
```

```
                                          SELF_LEN[count] = 
round(sim_dist,3) || ',' || round(orig_dist,3);
                                          count = count+1;
                                       END IF;
                              END IF;
                          END IF;
                      END IF;
                      array_count = array_count+1;
                END LOOP;
                    END IF;

                prv_org_dist = ST_LENGTH(simp_segments[each_simp_line]) 
+  prv_org_dist;
                prv_sim_dist = 
ST_DISTANCE(ST_StartPoint(ST_LineMerge(simp_segments[each_simp_line])), 
ST_EndPoint(ST_LineMerge(simp_segments[each_simp_line]))) + 
prv_sim_dist;
                array_count = 1;
                    sim_dist = 0.0;
                    orig_dist = 0.0;
                each_simp_line = each_simp_line+1;

            END IF;
                END LOOP;
            self_adv.AccumulatedLength = SELF_LEN;
      END IF;
RETURN self_adv;
END; $$
LANGUAGE 'plpgsql';
```

**2. SELF structure for Trajectories – PL/pgSQL function (Chapter 3)**

```
/* Function Definition – Takes input as original trajectory, 
simplification threshold, speed and heading based 
compression based threshold values */
CREATE OR REPLACE FUNCTION self_dyn_str_ml_sp(
    _tbl regclass,
    threshold double precision, spd_comp double precision, hdg_comp 
double precision)
  RETURNS selfadv AS
$BODY$
DECLARE
/* Variables and data type definition to be used in the 
algorithm */
curs1 refcursor;
rowcount integer;
sed_rowcount integer;
rowvar record;
avg_vlcty numeric;
sed_ms numeric[];
points geometry[];
segments geometry[];
noofpoints integer;
pointscount integer;
```

```
noofsegments integer;
segmentscount integer;
seg_len numeric;
self_adv SELFAdv;
time_diff numeric;
simp_geom geometry;
orig_geom geometry;
startingtime timestamp;
avg_vlcty_array numeric[];
prv_len numeric;
acc_len numeric[];
SELF_ARRAY  text[];
ind_pts_sgs integer;
ind_pts_ptr integer;
prv_speed double precision;
prv_hdg double precision;
spd_ratio double precision;
hdg_ratio double precision;
BEGIN
rowcount = 0;
sed_rowcount = 0;
prv_len = 0;
prv_speed = 0.0;
prv_hdg = 0.0;
/* Calculating average velocity of the trip and simplifying
the geometry based on Synchronous Euclidean Distance */
avg_vlcty_array = SELF_AVG_VLCY_ML_SP(_tbl,threshold);
orig_geom  = SELF_ORIG_GEOM(_tbl, threshold);
simp_geom =
ST_MAKELINE(SIMP_LINE(SED_SIMPFY(_tbl,threshold,threshold),orig_geom
));
acc_len = SELF_ACC_LEN_DYN(orig_geom);
segments = SELF_SED_ADV_CASE3(_tbl, threshold);
self_adv.StartingPoint = ST_StartPoint(ST_LineMerge(simp_geom));
self_adv.EndPoint = ST_EndPoint(ST_LineMerge(simp_geom));
self_adv.ActualLength = ST_LENGTH(orig_geom);
noofsegments = array_length(segments,1);
segmentscount = 1;
/* Reading the individual points on the trajectory using
cursors */
OPEN curs1 FOR EXECUTE format('SELECT * FROM %s', _tbl);
WHILE segmentscount <= noofsegments
LOOP
/* Identifying number of segments in the simplified line */
IF segmentscount = 1 THEN
       ind_pts_ptr = 1;
       ind_pts_sgs = COUNT_POINTS(segments[segmentscount]);
       RAISE NOTICE 'Number of points : %',ind_pts_sgs;
      WHILE ind_pts_ptr <= ind_pts_sgs
      LOOP
      FETCH curs1 INTO rowvar;
        EXIT WHEN NOT FOUND;
            IF prv_speed > 0.0  THEN
```

```
                  spd_ratio = (prv_speed-
rowvar.speed)*100.0/(prv_speed);
            ELSE
                  spd_ratio = 100.0;
            END IF;
            IF spd_ratio < 0.0 THEN
                  spd_ratio = -1 * spd_ratio;
            END IF;
            IF prv_hdg > 0.0 THEN
                  hdg_ratio = (prv_hdg-
rowvar.heading)*100.0/(prv_hdg);
            ELSE
                  hdg_ratio = 100.0;
            END IF;
/* Adding the semantics to the SELF structure */
            IF hdg_ratio < 0.0 THEN
                  hdg_ratio = -1 * hdg_ratio;
            END IF;
            IF ind_pts_ptr = 1 THEN
                  startingtime = CAST(rowvar.time AS TIMESTAMP);
                  sed_ms[sed_rowcount] = 0;
                  points[sed_rowcount] = rowvar.geom;
                  prv_speed = rowvar.speed;
                  prv_hdg = rowvar.heading;
                     SELF_ARRAY[rowcount] =
round(sed_ms[sed_rowcount],3) || ',' || rowvar.speed|| ',' ||
rowvar.heading || ',' ||  rowvar.time || ',' ||
acc_len[sed_rowcount];
                  rowcount = rowcount + 1;
            ELSIF ind_pts_ptr = (ind_pts_sgs) THEN
                  time_diff = EXTRACT(EPOCH FROM (CAST (rowvar.time
AS TIMESTAMP) - CAST(startingtime AS TIMESTAMP)));
                  IF ST_LENGTH(segments[segmentscount])>0 THEN
                  sed_ms[sed_rowcount] = (time_diff *
avg_vlcty_array[segmentscount-
1])*(ST_DISTANCE(ST_STARTPOINT(segments[segmentscount]),ST_ENDPOINT(
segments[segmentscount]))/ST_LENGTH(segments[segmentscount]));
                  ELSE
                  sed_ms[sed_rowcount] = (time_diff *
avg_vlcty_array[segmentscount-1]);
                  END IF;
                  points[sed_rowcount] = rowvar.geom;
                  startingtime = CAST(rowvar.time AS TIMESTAMP);
                       prv_len = prv_len + sed_ms[sed_rowcount];
                  IF (spd_ratio > spd_comp) AND (hdg_ratio >
hdg_comp) THEN
                       SELF_ARRAY[rowcount] =
round(sed_ms[sed_rowcount],3) || ',' || rowvar.speed|| ',' ||
rowvar.heading || ',' ||  rowvar.time || ',' ||
acc_len[sed_rowcount];
                  rowcount = rowcount + 1;
                  END IF;
            ELSE
                  time_diff = EXTRACT(EPOCH FROM (CAST (rowvar.time
AS TIMESTAMP) - CAST(startingtime AS TIMESTAMP)));
                  IF ST_LENGTH(segments[segmentscount])>0 THEN
                  sed_ms[sed_rowcount] = (time_diff *
avg_vlcty_array[segmentscount-
1])*(ST_DISTANCE(ST_STARTPOINT(segments[segmentscount]),ST_ENDPOINT(
segments[segmentscount]))/ST_LENGTH(segments[segmentscount]));
                  ELSE
                  sed_ms[sed_rowcount] = (time_diff *
avg_vlcty_array[segmentscount-1]);
                  END IF;
                  points[sed_rowcount] = rowvar.geom;
                  IF (spd_ratio > spd_comp) AND (hdg_ratio >
hdg_comp) THEN
                       SELF_ARRAY[rowcount] =
round(sed_ms[sed_rowcount],3) || ',' || rowvar.speed|| ',' ||
rowvar.heading || ',' ||  rowvar.time || ',' ||
acc_len[sed_rowcount];
                  rowcount = rowcount + 1;
                  END IF;
            END IF;
      sed_rowcount = sed_rowcount + 1;
      prv_speed = rowvar.speed;
      prv_hdg = rowvar.heading;
        ind_pts_ptr = ind_pts_ptr + 1;
        RAISE NOTICE 'rowcount : %',rowcount;
      END LOOP;
   ELSE
       ind_pts_ptr = 1;
       ind_pts_sgs = COUNT_POINTS(segments[segmentscount]);
       RAISE NOTICE 'Number of points from else block :
%',ind_pts_sgs;
      WHILE ind_pts_ptr < ind_pts_sgs
      LOOP
      FETCH curs1 INTO rowvar;
        EXIT WHEN NOT FOUND;

/* Applying Speed and Heading based compression values */
            IF prv_speed > 0.0  THEN
                  spd_ratio = (prv_speed-
rowvar.speed)*100.0/(prv_speed);
            ELSE
                  spd_ratio = 100.0;
            END IF;
            IF spd_ratio < 0.0 THEN
                  spd_ratio = -1 * spd_ratio;
            END IF;
            IF prv_hdg > 0.0 THEN
                  hdg_ratio = (prv_hdg-
rowvar.heading)*100.0/(prv_hdg);
            ELSE
                  hdg_ratio = 100.0;
            END IF;
            IF hdg_ratio < 0.0 THEN
                  hdg_ratio = -1 * hdg_ratio;
            END IF;

/* Calculating the accumulated distance at each point on the
original trajectory and its SED projection point on the
generalized trajectory  */
            IF ind_pts_ptr = (ind_pts_sgs-1) THEN
                  time_diff = EXTRACT(EPOCH FROM (CAST (rowvar.time
AS TIMESTAMP) - CAST(startingtime AS TIMESTAMP)));
                  IF ST_LENGTH(segments[segmentscount])>0 THEN
                  sed_ms[sed_rowcount] = (time_diff *
avg_vlcty_array[segmentscount-
1])*(ST_DISTANCE(ST_STARTPOINT(segments[segmentscount]),ST_ENDPOINT(
segments[segmentscount]))/ST_LENGTH(segments[segmentscount]));
                  ELSE
                  sed_ms[sed_rowcount] = (time_diff *
avg_vlcty_array[segmentscount-1]);
                  END IF;
                  points[sed_rowcount] = rowvar.geom;

                  ind_pts_ptr = ind_pts_ptr + 1;
                  startingtime =  CAST(rowvar.time AS TIMESTAMP);
                  prv_len = prv_len + sed_ms[sed_rowcount];
            IF (spd_ratio > spd_comp) AND (hdg_ratio > hdg_comp) THEN
                  SELF_ARRAY[rowcount] =
round(sed_ms[sed_rowcount],3) || ',' || rowvar.speed|| ',' ||
rowvar.heading || ',' ||  rowvar.time || ',' ||
acc_len[sed_rowcount];
                rowcount = rowcount + 1;
            END IF;
                  sed_rowcount = sed_rowcount +1;
            ELSE
                  time_diff = EXTRACT(EPOCH FROM (CAST (rowvar.time
AS TIMESTAMP) - CAST(startingtime AS TIMESTAMP)));


/* Semantic based compression levels */
IF ST_LENGTH(segments[segmentscount])>0 THEN
                  sed_ms[sed_rowcount] = (time_diff *
avg_vlcty_array[segmentscount-
1])*(ST_DISTANCE(ST_STARTPOINT(segments[segmentscount]),ST_ENDPOINT(
segments[segmentscount]))/ST_LENGTH(segments[segmentscount]));
                  ELSE
                  sed_ms[sed_rowcount] = (time_diff *
avg_vlcty_array[segmentscount-1]);
                  END IF;
                  points[sed_rowcount] = rowvar.geom;

                  ind_pts_ptr = ind_pts_ptr + 1;
            IF (spd_ratio > spd_comp) AND (hdg_ratio > hdg_comp) THEN
                     SELF_ARRAY[rowcount] =
round(sed_ms[sed_rowcount],3) || ',' || rowvar.speed|| ',' ||
rowvar.heading || ',' ||  rowvar.time || ',' ||
acc_len[sed_rowcount];
                  rowcount = rowcount + 1;
           END IF;
                 sed_rowcount = sed_rowcount +1;
           END IF;
           prv_speed = rowvar.speed;
           prv_hdg = rowvar.heading;
            RAISE NOTICE 'row count from else block: %',rowcount;
      END LOOP;
   END IF;
   ind_pts_ptr=0;
   segmentscount = segmentscount+1;
END LOOP;
IF spd_comp >= 0.0 THEN
SELF_ARRAY[rowcount] = round(prv_len,3) || ',' || prv_speed || ','
|| prv_hdg || ',' || acc_len[sed_rowcount-1];
END IF;
self_adv.AccumulatedLength = SELF_ARRAY;
RETURN self_adv;
END; $BODY$
  LANGUAGE 'plpgsql';
```

### 3. JAVA methods for combining simplified trajectory with SELF structure (Chapter 4)

```
/* Method for combining the simplified geometry with SELF
structure to generate the nodes for the proposed graph model
*/
public List<String> generatetrGraphNodes(List<CustomPoint> simpPoints, List<Double>
arrayOfDistance, List<SELFEXTENTED> selfArray, String comment){


                List<String> nodeString = new ArrayList<String>();
                Trajectory trajectory = new Trajectory();
                Integer simpPointsPointer = 0;
                trajectory.comment = comment;
                trajectory.nodes =  new ArrayList<Node>();
                trajectory.edges = new ArrayList<Edge>();
                int noofIntermediatePoints = 0;
                Integer previousId=0;
                for(int i=0;i<selfArray.size() ;i++){
                        SELFEdge e = new SELFEdge();
                        SELFNode n = new SELFNode();
/* Adding the properties to the first node (Starting point)
*/
                        if (i==0){
                                n.id = simpPointsPointer+1;
                                n.latitude =
simpPoints.get(simpPointsPointer.intValue()).getLatitude();
                                n.longitude =
simpPoints.get(simpPointsPointer.intValue()).getLongitude();
                                n.simpDist = selfArray.get(i).getSimpDist();
                                n.speed = selfArray.get(i).getSpeed();
                                n.heading = selfArray.get(i).getHeading();
                                n.origDist = selfArray.get(i).getOrigDist();
```

```
                             n.time =  selfArray.get(i).getTime();
                             n.nodeType = "main";
                             trajectory.nodes.add(n);
                             simpPointsPointer  = simpPointsPointer + 1;
                             previousId = n.id;

                             System.out.println(comment+"_"+n.id + "," + n.latitude +
"," + n.longitude + "," + n.simpDist + "," + n.speed + "," + n.heading + "," +
n.origDist + "," + n.time + "," +  n.nodeType);
                             nodeString.add(comment+"_"+n.id + "," + n.latitude + "," +
n.longitude + "," + n.simpDist + "," + n.speed + "," + n.heading + "," + n.origDist +
"," + n.time + "," +  n.nodeType);

                      }

/* Adding the properties to the intermediate nodes */
                      else
if(arrayOfDistance.get(simpPointsPointer.intValue()).equals(selfArray.get(i).getSimpDis
t())){

                             n.id = simpPointsPointer+1;
                             n.latitude =
simpPoints.get(simpPointsPointer.intValue()).getLatitude();
                             n.longitude =
simpPoints.get(simpPointsPointer.intValue()).getLongitude();
                             n.simpDist = selfArray.get(i).getSimpDist();
                             n.speed = selfArray.get(i).getSpeed();
                             n.heading = selfArray.get(i).getHeading();
                             n.origDist = selfArray.get(i).getOrigDist();
                             n.time =  selfArray.get(i).getTime();
                             n.nodeType = "main";
                             trajectory.nodes.add(n);
                             simpPointsPointer  = simpPointsPointer + 1;

                             System.out.println(comment+"_"+n.id + "," + n.latitude +
"," + n.longitude + "," + n.simpDist + "," + n.speed + "," + n.heading + "," +
n.origDist + "," + n.time + "," +  n.nodeType);
                             nodeString.add(comment+"_"+n.id + "," + n.latitude + "," +
n.longitude + "," + n.simpDist + "," + n.speed + "," + n.heading + "," + n.origDist +
"," + n.time + "," +  n.nodeType);


                             SELFEdge e1 = new SELFEdge();
                             e1.caption = "NEXT";
                             e1.source = previousId;
                             e1.target = simpPointsPointer;
                             if(e1.source != e1.target-1){
                                    e1.noOfIntermediatePoints = 0;
                                    e1.simpDistWeight =
       Double.parseDouble(String.format("%.3f",
arrayOfDistance.get(simpPointsPointer.intValue()-1) -
arrayOfDistance.get(simpPointsPointer.intValue()-2)));

                                    e1.origDistWeight =
       Double.parseDouble(String.format("%.3f", selfArray.get(i).getOrigDist() -
selfArray.get(i-noofIntermediatePoints-1).getOrigDist()));

                                    e1.between = (simpPointsPointer-1)+"-
"+simpPointsPointer;
                                    e1.edgeType = "BRANCH";
                                    trajectory.edges.add(e1);
                             }
                             e.caption = "NEXT";
                             e.source = (simpPointsPointer-1);
                             e.target = simpPointsPointer;
                             e.edgeType = "MAINSTREAM";
                             if(noofIntermediatePoints>0){

                                    e.noOfIntermediatePoints = noofIntermediatePoints;
                                    e.simpDistWeight =
       Double.parseDouble(String.format("%.3f",
arrayOfDistance.get(simpPointsPointer.intValue()-1) -
arrayOfDistance.get(simpPointsPointer.intValue()-2)));

                                    e.origDistWeight =
       Double.parseDouble(String.format("%.3f", selfArray.get(i).getOrigDist() -
selfArray.get(i-noofIntermediatePoints-1).getOrigDist()));


                             }
                             else{


                                    e.noOfIntermediatePoints = noofIntermediatePoints;
                                    e.simpDistWeight =
       Double.parseDouble(String.format("%.3f",
arrayOfDistance.get(simpPointsPointer.intValue()-1) -
arrayOfDistance.get(simpPointsPointer.intValue()-2)));

                                    e.origDistWeight =
       Double.parseDouble(String.format("%.3f", selfArray.get(i).getOrigDist() -
selfArray.get(i-noofIntermediatePoints-1).getOrigDist()));


                             }

                             trajectory.edges.add(e);

                             previousId = n.id;
                             noofIntermediatePoints = 0;
                      }
                      else{
                             noofIntermediatePoints = noofIntermediatePoints +1;

                             n.id = Integer.parseInt(String.valueOf(simpPointsPointer
+""+(simpPointsPointer+1)+""+ (noofIntermediatePoints)));
                             n.simpDist = selfArray.get(i).getSimpDist();
                             n.speed = selfArray.get(i).getSpeed();
                             n.heading = selfArray.get(i).getHeading();
                             n.origDist = selfArray.get(i).getOrigDist();
                             n.nodeType = "side";
                             n.between = simpPointsPointer+"-"+(simpPointsPointer+1);
                             n.time =  selfArray.get(i).getTime();
                             trajectory.nodes.add(n);

       System.out.println(comment+"_"+String.valueOf(simpPointsPointer
+"_"+(simpPointsPointer+1)+"_"+ (noofIntermediatePoints)) + "," + n.latitude + "," +
n.longitude + "," + n.simpDist + "," + n.speed + "," + n.heading + "," + n.origDist +
"," + n.time + "," +  n.nodeType);

       nodeString.add(comment+"_"+String.valueOf(simpPointsPointer
+"_"+(simpPointsPointer+1)+"_"+ (noofIntermediatePoints)) + "," + n.latitude + "," +
n.longitude + "," + n.simpDist + "," + n.speed + "," + n.heading + "," + n.origDist +
"," + n.time + "," +  n.nodeType);
                             e.caption = "NEXT";
                             e.source = previousId;
                             e.target = n.id;
                             e.edgeType = "BRANCH";
                             e.between = simpPointsPointer+"-"+(simpPointsPointer+1);
                             previousId = n.id;
                             trajectory.edges.add(e);

                     }

                     }

/* Building the JSON document for the nodes */
                     GsonBuilder builder = new GsonBuilder();
                     Gson gson = builder.create();
                     String gsonString = gson.toJson(trajectory);
                     return nodeString;
       }
/* Method for combining the simplified geometry with SELF
structure to generate the nodes for the proposed graph model
*/

       public List<String> generatetrGraphEdges(List<CustomPoint> simpPoints,
List<Double> arrayOfDistance, List<SELFEXTENTED> selfArray, String comment){


                     List<String> edgeString = new ArrayList<String>();
                     Trajectory trajectory = new Trajectory();
                     Integer simpPointsPointer = 0;
                     trajectory.comment = comment;
                     trajectory.nodes =  new ArrayList<Node>();
                     trajectory.edges = new ArrayList<Edge>();
                     int noofIntermediatePoints = 0;
                     Integer previousId=0;
                     String prvID = "";
/* Connecting starting node with the intermediate nodes */

                     for(int i=0;i<selfArray.size() ;i++){
                         SELFEdge e = new SELFEdge();
                         SELFNode n = new SELFNode();
                         if (i==0){
                             n.id = simpPointsPointer+1;
                             n.latitude =
simpPoints.get(simpPointsPointer.intValue()).getLatitude();
                             n.longitude =
simpPoints.get(simpPointsPointer.intValue()).getLongitude();
                             n.simpDist = selfArray.get(i).getSimpDist();
                             n.speed = selfArray.get(i).getSpeed();
                             n.heading = selfArray.get(i).getHeading();
                             n.origDist = selfArray.get(i).getOrigDist();
                             n.time =  selfArray.get(i).getTime();
                             n.nodeType = "main";
                             trajectory.nodes.add(n);
                             simpPointsPointer = simpPointsPointer + 1;
                             previousId = n.id;
                             prvID = String.valueOf(previousId);
```

```
                      }
                      else
if(arrayOfDistance.get(simpPointsPointer.intValue()).equals(selfArray.get(i).getSimpDis
t())){

                             n.id = simpPointsPointer+1;
                             n.latitude =
simpPoints.get(simpPointsPointer.intValue()).getLatitude();
                             n.longitude =
simpPoints.get(simpPointsPointer.intValue()).getLongitude();
                             n.simpDist = selfArray.get(i).getSimpDist();
                             n.speed = selfArray.get(i).getSpeed();
                             n.heading = selfArray.get(i).getHeading();
                             n.origDist = selfArray.get(i).getOrigDist();
                             n.time =  selfArray.get(i).getTime();
                             n.nodeType = "main";
                             trajectory.nodes.add(n);
                             simpPointsPointer  = simpPointsPointer + 1;

                             SELFEdge e1 = new SELFEdge();
                             e1.caption = "NEXT";
                             e1.source = previousId;
                             e1.target = simpPointsPointer;
                             if(e1.source != e1.target-1){
                                    e1.noOfIntermediatePoints = 0;

                                    e1.between = (simpPointsPointer-1)+"-
"+simpPointsPointer;
                                    e1.edgeType = "BRANCH";
                                    System.out.println(e1.caption + "," +
comment+"_"+e1.source +"," +comment+"_"+e1.target +","+e1.edgeType +","+e1.between +
"," + e1.noOfIntermediatePoints +","+e1.simpDistWeight +","+ e1.origDistWeight);
                                    edgeString.add(e1.caption + "," +
comment+"_"+prvID +"," +comment+"_"+e1.target +","+e1.edgeType +","+e1.between + "," +
e1.noOfIntermediatePoints +","+e1.simpDistWeight +","+ e1.origDistWeight);

                                    trajectory.edges.add(e1);
                             }

                             e.caption = "NEXT";
                             e.source = (simpPointsPointer-1);
                             e.target = simpPointsPointer;
                             e.edgeType = "MAINSTREAM";
                             if(noofIntermediatePoints>0){

                                    e.noOfIntermediatePoints = noofIntermediatePoints;
                                    e.simpDistWeight =
       Double.parseDouble(String.format("%.3f",
arrayOfDistance.get(simpPointsPointer.intValue()-1) -
arrayOfDistance.get(simpPointsPointer.intValue()-2)));

                                    e.origDistWeight =
       Double.parseDouble(String.format("%.3f", selfArray.get(i).getOrigDist() -
selfArray.get(i-noofIntermediatePoints-1).getOrigDist()));


                             }
                             else{

                                    e.noOfIntermediatePoints = noofIntermediatePoints;
```

```
                }
                System.out.println(e.caption + "," + comment+"_"+e.source
+"," +comment+"_"+e.target +","+e.edgeType +","+e.between + "," +
e.noOfIntermediatePoints +","+e.simpDistWeight +","+ e.origDistWeight);
                edgeString.add(e.caption + "," + comment+"_"+e.source +","
+comment+"_"+e.target +","+e.edgeType +","+e.between + "," + e.noOfIntermediatePoints
+","+e.simpDistWeight +","+ e.origDistWeight);
                trajectory.edges.add(e);

                previousId = n.id;
                prvID = String.valueOf(previousId);
                noofIntermediatePoints = 0;
            }
/*  Adding properties to the edges connecting intermediate nodes */
            else{
                noofIntermediatePoints = noofIntermediatePoints +1;

                n.id = Integer.parseInt(String.valueOf(simpPointsPointer
+""+(simpPointsPointer+1)+""+ (noofIntermediatePoints)));

                n.simpDist = selfArray.get(i).getSimpDist();
                n.speed = selfArray.get(i).getSpeed();
                n.heading = selfArray.get(i).getHeading();
                n.origDist = selfArray.get(i).getOrigDist();
                n.nodeType = "side";
                n.between = simpPointsPointer+"-"+(simpPointsPointer+1);
                n.time =  selfArray.get(i).getTime();
                trajectory.nodes.add(n);
                e.caption = "NEXT";
                e.source = previousId;
                e.target = n.id;
                e.edgeType = "BRANCH";
                e.between = simpPointsPointer+"-"+(simpPointsPointer+1);
                previousId = n.id;
                trajectory.edges.add(e);
                System.out.println(e.caption + "," + comment+"_"+e.source
+"," +comment+"_"+e.target +","+e.edgeType +","+e.between + "," +
e.noOfIntermediatePoints +","+e.simpDistWeight +","+ e.origDistWeight);
                edgeString.add(e.caption + "," + comment+"_"+prvID +","
+comment+"_"+ String.valueOf(simpPointsPointer +"_"+(simpPointsPointer+1)+"_"+
(noofIntermediatePoints)) +","+e.edgeType +","+e.between + "," +
e.noOfIntermediatePoints +","+e.simpDistWeight +","+ e.origDistWeight);
                prvID = String.valueOf(simpPointsPointer
+"_"+(simpPointsPointer+1)+"_"+ (noofIntermediatePoints));

            }

        }
    /* Building the JSON document for the edges */
        GsonBuilder builder = new GsonBuilder();
        Gson gson = builder.create();
        String gsonString = gson.toJson(trajectory);

        return edgeString;



    }
```

# Curriculum Vitae

## Rajesh Tamilmani

2016 – 2017, MScEng., Geodesy and Geomatics Engineering, University of New Brunswick, Canada

2014, Bachelor of Engineering in Geoinformatics, Anna University, Chennai, India

### Publications:

### Peer-Reviewed Journal Papers:

**1. Tamilmani R,** Stefanakis E, 2017. *Enriched geometric simplification of linear features. Geomatica Vol. 71, No.1, 2017, pp. 3 to 19*. doi: dx.doi.org/10.5623/cig2017-101

**2. Tamilmani R,** Stefanakis E, 2017. *Semantically enriched simplification of trajectories.* – (under review)

**3. Tamilmani R,** Stefanakis E, 2017. *Modelling and Analysis of Semantically Enriched Simplified Trajectories using Graph Databases* – (under review)

### Conference Papers/Presentations:

**1. Tamilmani, R.,** Stefanakis, E., "*ESRI Web AppBuilder for rediscovering the journey of an abandoned child*". The 2017 ESRI User Conference, November 14-15 2017, Halifax, Canada

**2. Tamilmani, R.,** Stefanakis, E., "*Enriched Geometric Simplification of Linear Features*". The 2017 Graduate Research Conference (GRC), University of New Brunswick, March 18 2017, Fredericton, Canada