# yoGERT GIS Toolbox

Capstone 4G06

Module Interface Specification for yoGERT

Team 19,
Smita Singh, Abeer Alyasiri, Niyatha Rangarajan,
Moksha Srinivasan, Nicholas Lobo, Longwei Ye

April 6, 2023

# 1 Revision History

| Date | Version | Notes |
|---|---|---|
| January 18, 2023 | 1.0 | **Smita**:Generating Activity Locations Module, Main Module, Data Transformation Module. **Abeer**:Network Graph, Shortest Route, Alternative Route, Route Generation, Mapping. **Moksha**: Preprocessing, Data Transformation. **Longwei** Module Decomposition, **Niyatha** Generating episodes |
| February 5th, 2023 | 1.1 | **Smita**: Updating Modules **Moksha**: Updating Preprocessing Modules, addressing issues 50 and 53 from Team 14's Review |
| March 15, 2023 | 1.2 | **Abeer**: Updating Network Graph, Shortest Route Trace, Alternative Route, and Mapping modules. Adding Shortest Route Episode and Point modules. |
| April 04, 2023 | 1.2 | **Abeer**: Updating Network Graph, Shortest Route Trace, Alternative Route, and Mapping modules. |

## 2 Symbols, Abbreviations and Acronyms

See SRS

# Contents

1 Revision History i

2 Symbols, Abbreviations and Acronyms ii

3 Introduction 1

4 Notation 1

5 Module Decomposition 2

6 MIS of Module Preprocessing 3
6.1 Module . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
6.2 Uses . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
6.3 Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
6.3.1 Exported Constants . . . . . . . . . . . . . . . . . . . . . . . . 3
6.3.2 Exported Access Programs . . . . . . . . . . . . . . . . . . . . 3
6.4 Semantics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
6.4.1 State Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
6.4.2 Environment Variables . . . . . . . . . . . . . . . . . . . . . . 3
6.4.3 Assumptions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
6.4.4 Access Routine Semantics . . . . . . . . . . . . . . . . . . . . . 3
6.4.5 Local Functions . . . . . . . . . . . . . . . . . . . . . . . . . . 4

7 MIS of Module Network Graph 5
7.1 Template Module . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
7.2 Uses . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
7.3 Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
7.3.1 Exported Constants . . . . . . . . . . . . . . . . . . . . . . . . 5
7.3.2 Exported Type . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
7.3.3 Exported Access Programs . . . . . . . . . . . . . . . . . . . . 5
7.4 Semantics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
7.4.1 State Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
7.4.2 Environment Variables . . . . . . . . . . . . . . . . . . . . . . 6
7.4.3 Assumptions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
7.4.4 Access Routine Semantics . . . . . . . . . . . . . . . . . . . . . 6
7.4.5 Local Functions . . . . . . . . . . . . . . . . . . . . . . . . . . 7

8 MIS of Generate Episodes 8
8.1 Template Module . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
8.2 Uses . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
8.3 Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
8.3.1 Exported Constants . . . . . . . . . . . . . . . . . . . . . . . . 8

8.3.2 Exported Type . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
8.3.3 Exported Access Programs . . . . . . . . . . . . . . . . . . . . . . 8
8.4 Semantics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
8.4.1 State Variables . . . . . . . . . . . . . . . . . . . . . . . . . . 8
8.4.2 Environment Variables . . . . . . . . . . . . . . . . . . . . . . . 9
8.4.3 Assumptions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
8.4.4 Access Routine Semantics . . . . . . . . . . . . . . . . . . . . . 9

**9 MIS of Data Transformation Module 11**
9.1 Module . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
9.2 Uses . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
9.3 Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
9.3.1 Exported Constants . . . . . . . . . . . . . . . . . . . . . . . . 11
9.3.2 Exported Access Programs . . . . . . . . . . . . . . . . . . . . . 12
9.4 Semantics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
9.4.1 State Variables . . . . . . . . . . . . . . . . . . . . . . . . . . 12
9.4.2 Environment Variables . . . . . . . . . . . . . . . . . . . . . . . 12
9.4.3 Assumptions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
9.4.4 Access Routine Semantics . . . . . . . . . . . . . . . . . . . . . 13
9.4.5 Local Functions . . . . . . . . . . . . . . . . . . . . . . . . . . 14

**10 MIS of Module Shortest Route Trace 15**
10.1 Template Module . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
10.2 Uses . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
10.3 Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
10.3.1 Exported Constants . . . . . . . . . . . . . . . . . . . . . . . . 15
10.3.2 Exported Type . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
10.3.3 Exported Access Programs . . . . . . . . . . . . . . . . . . . . . 15
10.4 Semantics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
10.4.1 State Variables . . . . . . . . . . . . . . . . . . . . . . . . . . 15
10.4.2 Environment Variables . . . . . . . . . . . . . . . . . . . . . . . 15
10.4.3 Assumptions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
10.4.4 Access Routine Semantics . . . . . . . . . . . . . . . . . . . . . 16
10.4.5 Local Functions . . . . . . . . . . . . . . . . . . . . . . . . . . 16

**11 MIS of Module Shortest Route Stop 17**
11.1 Template Module . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
11.2 Uses . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
11.3 Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
11.3.1 Exported Constants . . . . . . . . . . . . . . . . . . . . . . . . 17
11.3.2 Exported Type . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
11.3.3 Exported Access Programs . . . . . . . . . . . . . . . . . . . . . 17
11.4 Semantics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

11.4.1 State Variables . . . 17
11.4.2 Environment Variables . . . 17
11.4.3 Assumptions . . . 17
11.4.4 Access Routine Semantics . . . 18
11.4.5 Local Functions . . . 18

**12 MIS of Module Shortest Route Episode 19**
12.1 Template Module . . . 19
12.2 Uses . . . 19
12.3 Syntax . . . 19
12.3.1 Exported Constants . . . 19
12.3.2 Exported Type . . . 19
12.3.3 Exported Access Programs . . . 19
12.4 Semantics . . . 19
12.4.1 State Variables . . . 19
12.4.2 Environment Variables . . . 19
12.4.3 Assumptions . . . 20
12.4.4 Access Routine Semantics . . . 20
12.4.5 Local Functions . . . 20

**13 MIS of Module Alternative Route 21**
13.1 Template Module . . . 21
13.2 Uses . . . 21
13.3 Syntax . . . 21
13.3.1 Exported Constants . . . 21
13.3.2 Exported Type . . . 21
13.3.3 Exported Access Programs . . . 21
13.4 Semantics . . . 21
13.4.1 State Variables . . . 21
13.4.2 Environment Variables . . . 21
13.4.3 Assumptions . . . 22
13.4.4 Access Routine Semantics . . . 22
13.4.5 Local Functions . . . 22

**14 MIS of Activity Location 23**
14.1 Template Module . . . 23
14.2 Uses . . . 23
14.3 Syntax . . . 23
14.3.1 Exported Constants . . . 23
14.3.2 Exported Type . . . 23
14.3.3 Exported Access Programs . . . 23
14.4 Semantics . . . 23
14.4.1 State Variables . . . 23

14.4.2 Environment Variables . . . 23
14.4.3 Assumptions . . . 23
14.4.4 Access Routine Semantics . . . 24

**15 MIS of ~~Stop~~ Point 25**
15.1 Template Module . . . 25
15.2 Uses . . . 25
15.3 Syntax . . . 25
15.3.1 Exported Constants . . . 25
15.3.2 Exported Type . . . 25
15.3.3 Exported Access Programs . . . 25
15.4 Semantics . . . 25
15.4.1 State Variables . . . 25
15.4.2 Environment Variables . . . 25
15.4.3 Assumptions . . . 26
15.4.4 Access Routine Semantics . . . 26

**16 MIS of Generating Activity Locations 27**
16.1 Module . . . 27
16.2 Uses . . . 27
16.3 Syntax . . . 27
16.3.1 Exported Constants . . . 27
16.3.2 Exported Access Programs . . . 27
16.4 Semantics . . . 27
16.4.1 State Variables . . . 27
16.4.2 Environment Variables . . . 27
16.4.3 Assumptions . . . 27
16.4.4 Access Routine Semantics . . . 28
16.4.5 Local Functions . . . 28

**17 MIS of Module Mapping 29**
17.1 Template Module . . . 29
17.2 Uses . . . 29
17.3 Syntax . . . 29
17.3.1 Exported Constants . . . 29
17.3.2 Exported Type . . . 29
17.3.3 Exported Access Programs . . . 29
17.4 Semantics . . . 30
17.4.1 State Variables . . . 30
17.4.2 Environment Variables . . . 30
17.4.3 Assumptions . . . 30
17.4.4 Access Routine Semantics . . . 30
17.4.5 Local Functions . . . 31

18 Appendix 33

# 3 Introduction

The following document details the Module Interface Specifications for the yoGERT toolbox

Complementary documents include the System Requirement Specification and Module Guide. The full documentation and implementation can be found at https://github.com/NicLobo/Capstone-yoGERT.

# 4 Notation

The structure of the MIS for modules comes from Hoffman and Strooper (1995), with the addition that template modules have been adapted from Ghezzi et al. (2003). The mathematical notation comes from Chapter 3 of Hoffman and Strooper (1995). For instance, the symbol := is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1 | c_2 \Rightarrow r_2 | ... | c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by .

| **Data Type** | **Notation** | **Description** |
|---|---|---|
| character | char | a single symbol or digit |
| integer | $\mathbb{Z}$ | a number without a fractional component in (-∞, ∞) |
| natural number | $\mathbb{N}$ | a number without a fractional component in [1, ∞) |
| real | $\mathbb{R}$ | any number in (-∞, ∞) |

The specification of  uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition,  uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification.

# 5 Module Decomposition

The following table is taken directly from the Module Guide document for this project.

| **Level 1** | **Level 2** |
|---|---|
| Hardware-Hiding | |
| Behaviour-Hiding | Generate Episode<br>Route Generation<br>Fetch Activity Locations<br>Mapping<br>Main |
| Software Decision | Preprocessing Inputs<br>Network Graph<br>Shortest Route<br>Alternative Route<br>Data Transformation |

Table 1: Module Hierarchy

# 6 MIS of Module Preprocessing

## 6.1 Module

Preprocessing Inputs

## 6.2 Uses

## 6.3 Syntax

### 6.3.1 Exported Constants

### 6.3.2 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| validateCSV | CSV | $\mathbb{B}$, newCSV | InvalidInput<br>IOError |

## 6.4 Semantics

### 6.4.1 State Variables

None.

### 6.4.2 Environment Variables

IOmanager : it is the access point between the local memory and the application. It is used when saving files locally.

### 6.4.3 Assumptions

- Assume CSV file paths are valid when working within the same directory.

### 6.4.4 Access Routine Semantics

validateCSV(csvfile):

- transition: None
- output: $(\forall columns \in CSV, if\ validateCols(CSV) == True,$ $return\ True, newCSV == (validateRows(CSV), dmsToDD(CSV), formatTime(CSV))$
- exception: $if\ \neg(ValidateCols(CSV) \Rightarrow$ InvalidInput
- exception: $(\neg(\forall rows : tuple(\mathrm{R}, \mathrm{R}, \mathrm{str}) | file \in normalizeCSV(csvfile) : validateRows(rows))) \Rightarrow$ IOError)

#### 6.4.5 Local Functions

validateRows : removes any invalid gps and time points from the original CSV data
validateCols : Ensures that data has latitude, longitude, and time columns
dmsToDD: If lat/long data type is DMS, converts to DD
formatTime : Updates time format to work with rest of toolbox

# 7 MIS of Module Network Graph

## 7.1 Template Module

Network Graph

## 7.2 Uses

None.

## 7.3 Syntax

### 7.3.1 Exported Constants

DISTANCETOLERANCE = 200 - tolerance distance (m) added to the radius of the circle for the mapped area that encapsulates all the input GPS coordinates.
EARTHRADIUS = 6371000 - earth's radius (m).

### 7.3.2 Exported Type

NetworkGraph = ?

### 7.3.3 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| new NetworkGraph | ~~tuple of (latitude: $\mathbb{R}$, longitude: $\mathbb{R}$), tuple of (latitude: $\mathbb{R}$, longitude: $\mathbb{R}$), seq of tuple of (latitude: $\mathbb{R}$, longitude: $\mathbb{R}$)~~ String, ~~seq of~~ String $\in$ {drive, walk, bike}, $\mathbb{B}$, $\mathbb{B}$ | NetworkGraph | InvalidMode, EmptyFilePath |
| getNearestNode | tuple of (latitude: $\mathbb{R}$, longitude: $\mathbb{R}$) | $\mathbb{N}$ | OutOfBoundsCoord |
| getMode | | String $\in$ {drive, walk, bike} | |

## 7.4 Semantics

### 7.4.1 State Variables

*dist* : $\mathbb{R}$ - distance in m for the search radius of the network graph ~~using GPS coordinates~~.
*stCoord* : tuple of (latitude: $\mathbb{R}$, longitude: $\mathbb{R}$) - starting GPS coordinate of the inputted data by the user.

*endCoord* : tuple of (latitude: $\mathbb{R}$, longitude: $\mathbb{R}$) - ending GPS coordinate of the inputted data by the user.
*graph* : directed graph.
~~*nodes* : seq of $\mathbb{N}$ - network graph node.~~
~~*edges* : seq of tuple of ($\mathbb{N}$, $\mathbb{N}$, $\mathbb{R}$) - network graph edge defined by 2 nodes and weight extracted from.~~

### 7.4.2 Environment Variables

onlinenetworkdatabase : connection to online database to retrieve information layers of intersections, roads, and paths initialised within module to build the network graph.

### 7.4.3 Assumptions

- Assume the inputted GPS coordinates are valid as they are the output of another module.
- Assume the online network database is always accessible within 10 minutes.

### 7.4.4 Access Routine Semantics

new NetworkGraph(~~*startcoord, endcoord, stopcoords,*~~*filepath*, *networkmode*, *episodeAnalysis*, *alternativeAnalysis*):

- transition: The *stCoord* ~~*startcoord*~~ and *endCoord* are set by $extractStartEnd(filepath$. The *dist* is set by $finddistance(stCoord, endCoord, extractData(filepath))$. The *graph* is set to the extracted data from *onlinenetworkdatabase* upon querying the search radius.
- output: Creates a *NetworkGraph* object with the parameters ~~*startcoord, endcoord, stopcoords,*~~*filepath*, *networkmode*, *episodeAnalysis*, *alternativeAnalysis*
- exception: $(\neg(networkmode \in \{drive, walk, bike\}) \Rightarrow \text{InvalidMode}) \vee (filepath == "") \Rightarrow \text{EmptyFilePath}$

getNearestNode(*coord*):

- transition: None
- output: *graph*.nodes[i] where $i \in [0..|datainput| - 1]$
- exception: $(\neg(findhdistance(coord, stCoord) \leq \text{dist}) \Rightarrow \text{OutOfBoundsCoord}))$

### 7.4.5 Local Functions

finddistance : tuple of (latitude: $\mathbb{R}$, longitude: $\mathbb{R}$) × tuple of (latitude: $\mathbb{R}$, longitude: $\mathbb{R}$) × seq of tuple of (latitude: $\mathbb{R}$, longitude: $\mathbb{R}$) → $\mathbb{R}$

- Description: Computes largest distance using *findhdistance* between starting coordinate and another coordinate, either a destination coordinate or another ~~stop~~ coordinate.

findhdistance : tuple of (latitude: $\mathbb{R}$, longitude: $\mathbb{R}$) × tuple of (latitude: $\mathbb{R}$, longitude: $\mathbb{R}$) → $\mathbb{R}$

- Description: finds the distance between two GPS coordinates using Haversine formula.

extractData : String → collection of *Point* Type

- Description: Uses the inputted file path to read the file and extract the data as a collection.

extractStartEnd : String → collection of tuples of (latitude: $\mathbb{R}$, longitude: $\mathbb{R}$)

- Description: Uses the inputted file path to read the file and extract the start and end GPS coordinate.

# 8 MIS of Generate Episodes

## 8.1 Template Module

Generate Episodes

## 8.2 Uses

None.

## 8.3 Syntax

### 8.3.1 Exported Constants

N/A

### 8.3.2 Exported Type

Activityepisode = ?

### 8.3.3 Exported Access Programs

| **Name** | **In** | **Out** | **Exceptions** |
|---|---|---|---|
| new Activityepisode | A full path to a .csv file containing GPS points (Start: Lat, Long, Time, Stop: Lat, Long, Time) | A full path to a .csv file containing preprocessed GPS data | invalidCSV |

## 8.4 Semantics

### 8.4.1 State Variables

*points* : columns of csv containing ($\mathbb{R}$, $\mathbb{R}$, *datetime*) - latitude, longitude and timestamp for a point.
*finalpoints* : columns of csv containing ($\mathbb{R}$, $\mathbb{R}$, *datetime*) - latitude, longitude and timestamp for a point after data has been run with generateEpisodes(csvPath).
*mode* : enum of ($\mathbb{N}$, $\mathbb{N}$, $\mathbb{N}$) - modes for episodes, STOP = 0, WALK = 1, DRIVE = 10
*timetol* time tolerance used to filter stops by time in cleanStops(csvPath).
*disttol* distance tolerance used to filter stops by distance in cleanStops(csvPath).

### 8.4.2 Environment Variables

N/A

### 8.4.3 Assumptions

- Assume the values for latitude, longitude and time in the given file are valid.

### 8.4.4 Access Routine Semantics

new Activityepisode($csvPath$):

- transition: A full path to a .csv file containing GPS points (Start: Lat, Long, Time, Stop: Lat, Long, Time) to a full path to a .csv file containing preprocessed GPS data.
- output: finalpoints[i] where $i \in [0..|len(points)| - 1]$
- exception: $(\neg(csvPath) \Rightarrow \text{InvalidFile}))$

createTrace($csvPath, title$):

- transition: Assigns a unique ID to each GPS ping point of the inputted trace file and creates a new CSV file, called trace.csv, for the trace's geo-data in the inputted directory path.
- output: CSV file with points filtered based on stepsize

createSegments($csvPath, title$):

- transition: The given csv file is parsed and trimmed based using the average stepsize of the given data points.
- output: points[i] where $i \in [0..|len(points)|-1] \wedge i = i+(points[-1]-points[0])/len(points)$
- exception: $(\neg(csvPath) \Rightarrow \text{InvalidFile}))$

createVelocities($csvPath$):

- transition: The given csv file is used to create another csv file with the added column velocity
- output: points.append(velocity[i] where $(sqrt((i.lat - (i + 1).lat)^2 + (i.long - (i + 1).long)^2)/(i.time - (i + 1).time)$
- exception: $(\neg(csvPath) \Rightarrow \text{InvalidFile}))$

findStops($csvPath$):

- transition: The given csv file is used to create another csv file called stops.csv

- output: CSV file called ”stops.csv” in a newly created directory called ”stop” within the input directory path.
- exception: $(\neg(csvPath) \Rightarrow \text{InvalidFile}))$

cleanStops($csvPath$):

- transition: The given csv file is used to update the existing rows in the csv
- output: The function updates the ”stops.csv” file in the ”stop” directory by removing any rows that do not fit the time and distance tolerance
- exception: $(\neg(csvPath) \Rightarrow \text{InvalidFile}))$

createEpisode($csvPath$):

- transition: The given csv path is used to generate a folder of episode CSV’s
- output: The function updates the episode folder filling it with CSV’s create from the original CSV.
- exception: $(\neg(csvPath) \Rightarrow \text{InvalidFile}))$

summarymode($csvPath$):

- transition: The given csv path that has the episode folder
- output:Creates a CSV that records the most used method of travel from all episode CSVs’
- exception: $(\neg(csvPath) \Rightarrow \text{InvalidFile}))$

episodeGenerator($csvPath$):

- transition: The given csv path the has the gps trace CSV
- output:generates the folder structure and all files from given gps trace
- exception: $(\neg(csvPath) \Rightarrow \text{InvalidFile}))$

# 9 MIS of Data Transformation Module

## 9.1 Module

Data Transformation Module

## 9.2 Uses

None

## 9.3 Syntax

### 9.3.1 Exported Constants

N/A

### 9.3.2 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| GenInputtrace | A full path to a .csv containing trace lat, travel episode long, mode detected and trace id | list of Point objects, longitude: $\mathbb{R}$) | invalidCSV |
| GenInputepisode | A full path to a .csv containing travel episodes lat, travel episode long, mode detected and travel episode id | list of Point objects, longitude: $\mathbb{R}$) | invalidCSV |
| GenInputstop | A full path to a .csv containing stop episodes lat, travel episode long, , mode detected and stop episode id | list of Point objects, longitude: $\mathbb{R}$) | invalidCSV |
| convertActivityLocation | a list of stop point and activity location objects tuples of form [(Point stop, [activitylocation object1,activitylocation object2]), (Point stop2, [activitylocation object1,activitylocation object2])] | a list of stop point lat, stop point long and activity location objects attributes in tuple form of form [(stop.lat, stop.lon, [[activitylocation.name,activitylocation.lat,activitylocation.lon,activi | invalidCSV |
| convertActivityCSV | output file of $fetchactivityLocation$ | list of activity location objects | invalidCSV |

## 9.4 Semantics

### 9.4.1 State Variables

None

### 9.4.2 Environment Variables

None

### 9.4.3 Assumptions

We assume that the input files are generated from the GenerateEpisodes module.

### 9.4.4 Access Routine Semantics

GenInput1(tracefile):

- transition: N/A
- output: list of *Point* objects provided by genHelper
- exception: invalidCSV

GenInput2(episodefile):

- transition: N/A
- output: list of *Point* objects provided by genHelper
- exception: invalidCSV

GenInput3(stopfile):

- transition: N/A
- output: list of *Point* objects provided by genHelper
- exception: invalidCSV

convertActivityLocation(activitylocationlist):

- transition: N/A
- output: a list of stop point lat, stop point long and activity location objects attributes in tuple form of form [(stop.lat, stop.lon, [[activitylocation.name,activitylocation.lat,activitylocation.l
- exception: invalidCSV

convertActivityCSV(fetchActivitylocationfile):

- transition: N/A
- output: list of *activityLocation* objects given by convertListToActivityLocationObject
- exception: invalidCSV

### 9.4.5 Local Functions

convertListToActivityLocationObject : CSV line(String)

- Description : Converts a list of activity location attributes into an activity location object. Converts activityLocationList, an activity location attribute list of form [activitylocation.name, activitylocation.lat,activitylocation.lon,activitylocation.amenity] to an activity location object

genHelper : CSV line(String)

- Description : string of lat, travel episode long, mode detected and id and converts content returning *Point* object

# 10 MIS of Module Shortest Route Trace

## 10.1 Template Module

Shortest Route Trace

## 10.2 Uses

Network Graph

## 10.3 Syntax

### 10.3.1 Exported Constants

None.

### 10.3.2 Exported Type

ShortestRouteTrace = ?

### 10.3.3 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| new ShortestRouteTrace | NetworkGraph, ~~tuple of (latitude: $\mathbb{R}$, longitude: $\mathbb{R}$), tuple of (latitude: $\mathbb{R}$, longitude: $\mathbb{R}$)~~, String, String $\in \{time, distance\}$ | ShortestRouteTrace | InvalidWeight,<br>OutOfBoundsCoord<br>EmptyFilePath<br>NoPathFound |

## 10.4 Semantics

### 10.4.1 State Variables

$graph$ : $NetworkGraph$ - directed graph used to create the route from.<br>
$data$ : collection of $Point$ Type - GPS pings used to create route from.<br>
$nodes$ : collection of $\mathbb{N}$ - relevant graph nodes to construct the route from.<br>
$routes$ : collection of $\mathbb{N}$ - node connections to form a route.<br>
~~$startnode$ : $\mathbb{N}$ - graph node nearest the starting GPS coordinate.~~<br>
~~$endnode$ : $\mathbb{N}$ - graph node nearest the destination GPS coordinate.~~

### 10.4.2 Environment Variables

None.

### 10.4.3 Assumptions

- Assume the inputted GPS coordinates are valid as they are the output of another module.

### 10.4.4 Access Routine Semantics

new ShortestRouteTrace(*graph*, *filePath* ~~*startcoord, endcoord,*~~ *weighttype*):

- transition: Set state variables ~~startnode, endnode := findnode(graph, startcoord), findnode(graph, endcoord)~~ *graph, data, nodes, routes := graph, extractData(filepath), findNodes(graph, extractData(graph, filePath)), createRoutes(graph, weighttype, nodes).*

- output: ~~Creates~~ *ShortestRouteTrace* ~~using *DijkstraAlg(graph, weighttype, startnode, endnode)*~~

- exception: $(\neg(weighttype \in \{time, distance\}) \Rightarrow$ InvalidWeight$)\vee(\neg(extractData(filepath)[i] \in graph.nodes) \Rightarrow$ OutOfBoundsCoord$)\vee$ $(\neg$(an edge graph exists between two nodes) $\Rightarrow$ NoPathFound) $\vee(filepath == ””) \Rightarrow$ EmptyFilePath

### 10.4.5 Local Functions

findnodes : *NetworkGraph* $\times$ collection of *Point* type $\rightarrow$ collection of $\mathbb{N}$ ~~tuple of $(\mathbb{R}, \mathbb{R}) \rightarrow \mathbb{N}$~~

- Description : iterates over the inputted GPS data and finds a node for each GPS ping using the *NetworkGraph* access routine called *NetworkGraph.getNearestNode*. ~~(GPS coordinate) using the *NetworkGraph* access routine the function returns node number.~~

DijkstraAlg : *NetworkGraph* $\times$ String $\times$ $\mathbb{N} \times \mathbb{N} \rightarrow$ collection of $\mathbb{N}$

- Description : using Dijkstra's Shortest Path Algorithm given graph with weighted edges, source node, and destination node find shortest path as a seq of nodes.

CreateRoutes : *NetworkGraph, String,* collection of $\mathbb{N} \rightarrow$ collection of *Point* Type

- Description: iterates over the matched graph nodes to find the shortest path between every two nodes using *DijkstraAlg.*

extractData : *String* $\rightarrow$ collection of *Point* Type

- Description: Uses the inputted file path to read the file and extract the data as a collection.

# 11 MIS of Module Shortest Route Stop

## 11.1 Template Module

Shortest Route Stop

## 11.2 Uses

Network Graph

## 11.3 Syntax

### 11.3.1 Exported Constants

None.

### 11.3.2 Exported Type

ShortestRouteStop = ?

### 11.3.3 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| new ShortestRouteStop | NetworkGraph, String, String $\in \{time, distance\}$ | ShortestRouteStop | InvalidWeight,<br>OutOfBoundsCoord,<br>EmptyFilePath,<br>NoPathFound |

## 11.4 Semantics

### 11.4.1 State Variables

$graph$ : $NetworkGraph$ - directed graph used to create the route from.
$data$ : collection of $Point$ Type - Stop GPS pings used to create route from.
$nodes$ : collection of $\mathbb{N}$ - relevant graph nodes to construct the route from.
$routes$ : collection of $\mathbb{N}$ - node connections to form a route.

### 11.4.2 Environment Variables

None.

### 11.4.3 Assumptions

- Assume the inputted GPS coordinates are valid as they are the output of another module.

### 11.4.4 Access Routine Semantics

new ShortestRouteStop(*graph, filePath, weighttype*):

- transition: Set state variables *graph, data, nodes, routes := graph, extractStopData(filepath), findNodes(graph, extractStopData(graph, filePath)), createRoutes(graph, weighttype, nodes).*
- output: *ShortestRouteStop*
- exception: $(\neg(weighttype \in \{time, distance\}) \Rightarrow$ InvalidWeight$)\vee(\neg(extractData(filepath)[i] \in graph.nodes) \Rightarrow$ OutOfBoundsCoord$)\vee(\neg($an edge graph exists between two nodes$) \Rightarrow$ NoPathFound$) \vee (filepath ==$ ””$) \Rightarrow$ EmptyFilePath

### 11.4.5 Local Functions

findnodes : *NetworkGraph* × collection of *Point* type → collection of $\mathbb{N}$

- Description : iterates over the inputted GPS data and finds a node for each GPS ping using the *NetworkGraph* access routine called *NetworkGraph.getNearestNode*.

DijkstraAlg : *NetworkGraph* × String × $\mathbb{N} \times \mathbb{N}$ → collection of $\mathbb{N}$

- Description : using Dijkstra’s Shortest Path Algorithm given graph with weighted edges, source node, and destination node find shortest path as a collection of nodes.

CreateRoutes : *NetworkGraph, String*, collection of $\mathbb{N}$ → collection of *Point* Type

- Description: iterates over the matched graph nodes to find the shortest path between every two nodes using *DijkstraAlg*.

extractStopData : *String* → collection of *Point* Type

- Description: Uses the inputted file path to read the file and extract the stop GPS pings as a collection.

# 12 MIS of Module Shortest Route Episode

## 12.1 Template Module

Shortest Route Episode

## 12.2 Uses

Network Graph

## 12.3 Syntax

### 12.3.1 Exported Constants

None.

### 12.3.2 Exported Type

ShortestRouteEpisode = ?

### 12.3.3 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| new ShortestRouteEpisode | NetworkGraph, String, String $\in \{time, distance\}$,B,N | ShortestRouteEpisode | InvalidWeight,<br>OutOfBoundsCoord,<br>EmptyFilePath,<br>NoPathFound |

## 12.4 Semantics

### 12.4.1 State Variables

$graph$ : $NetworkGraph$ - directed graph used to create the route from.<br>
$data$ : collection of $Point$ Type - inputted GPS pings used to create route from.
$sampleddata$ : collection of $Point$ Type - sampled GPS pings used to create route from.
$nodes$ : collection of $\mathbb{N}$ - relevant graph nodes to construct the route from.
$routes$ : collection of $\mathbb{N}$ - node connections to form a route.

### 12.4.2 Environment Variables

None.

### 12.4.3 Assumptions

- Assume the inputted GPS coordinates are valid as they are the output of another module.

### 12.4.4 Access Routine Semantics

new ShortestRouteEpisode(*graph, filePath, weighttype, sampling, samplingdist*):

- transition: Set state variables *graph, data, sampleddata, nodes, routes := graph, extractData(filepath), sampleData(extractData(filepath), samplingdist), findNodes(graph, sampleddata), createRoutes(graph, weighttype, nodes).*
- output: *ShortestRouteEpisode*
- exception: $(\neg(weighttype \in \{time, distance\}) \Rightarrow \text{InvalidWeight}) \vee (\neg(extractData(filepath)[i] \in graph.nodes) \Rightarrow \text{OutOfBoundsCoord}) \vee (\neg(\text{an edge graph exists between two nodes}) \Rightarrow \text{NoPathFound}) \vee (filepath == "") \Rightarrow \text{EmptyFilePath}$

### 12.4.5 Local Functions

findnodes : *NetworkGraph* × collection of *Point* type → collection of $\mathbb{N}$

- Description : iterates over the samples GPS data and finds a node for each GPS ping using the *NetworkGraph* access routine called *NetworkGraph.getNearestNode*.

DijkstraAlg : *NetworkGraph* × String × $\mathbb{N} \times \mathbb{N}$ → collection of $\mathbb{N}$

- Description : using Dijkstra's Shortest Path Algorithm given graph with weighted edges, source node, and destination node find shortest path as a collection of nodes.

CreateRoutes : *NetworkGraph* × *String* × collection of $\mathbb{N}$ → collection of *Point* Type

- Description: iterates over the matched graph nodes to find the shortest path between every two nodes using *DijkstraAlg*.

extractData : *String* → collection of *Point* Type

- Description: Uses the inputted file path to read the file and extract the GPS pings as a collection.

sampleData : collection of *Point* × $\mathbb{N}$ → collection of *Point* Type

- Description: Samples the GPS pings to be used for routes.

# 13 MIS of Module Alternative Route

## 13.1 Template Module

Alternative Route

## 13.2 Uses

Network Graph, ShortestRouteTrace, ShortestRouteStop

## 13.3 Syntax

### 13.3.1 Exported Constants

None.

### 13.3.2 Exported Type

AlternativeRoute = ?

### 13.3.3 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| new AlternativeRoute | *String, String, String* ~~NetworkGraph, tuple of (latitude: $\mathbb{R}$, longitude: $\mathbb{R}$), tuple of (latitude: $\mathbb{R}$, longitude: $\mathbb{R}$)~~ | AlternativeRoute | OutOfBoundsCoord,<br>InvalidWeight,<br>EmptyFilePath,<br>NoPathFound |

## 13.4 Semantics

### 13.4.1 State Variables

$network : NetworkGraph$ - graph of the bike transportation network.
$path : ShortestRoute$ - shortest route representation for a bike's shortest travel path.
~~$startnode : \mathbb{N}$ - graph node nearest the starting GPS coordinate.~~
~~$endnode : \mathbb{N}$ - graph node nearest the destination GPS coordinate.~~

### 13.4.2 Environment Variables

onlinetransitdatabase : connection to online database to retrieve information layers of bus stops initialised within module to deactivate inaccessible edges on the network graph.

### 13.4.3 Assumptions

- Assume the inputted GPS coordinates are valid as they are the output of another module.

### 13.4.4 Access Routine Semantics

new AlternativeRoute(~~graph,~~ filepath, optimizer, stopsfilepath ~~startcoord, endcoord~~):

- transition: Set network, path := *NetworkGraph(filepath, "bike", False, True)*, findPath(network, filepath, optimizer, stopsfilepath. ~~startnode, endnode, graph := findnode(graph, startcoord), findnode(graph, endcoord) , *deactivateedges(graph)* at the beginning.~~

- output: ~~Creates~~ *AlternativeRoute* ~~using pathfinder(graph, startcoord, endcoord)~~.

- exception: $(\neg(weighttype \in \{time, distance\}) \Rightarrow \text{InvalidWeight})\vee(\neg(extractData(filepath)[i] \in graph.nodes) \Rightarrow \text{OutOfBoundsCoord})\vee(\neg(\text{an edge graph exists between two nodes}) \Rightarrow \text{NoPathFound})\vee(filepath == \text{""}) \Rightarrow \text{EmptyFilePath}$ ~~$(\neg(startnode \in \text{seq of graph.nodes}) \Rightarrow \text{OutOfBoundsCoord}) \vee (\neg(endnode \in seqof\,graph.nodes) \Rightarrow \text{OutOfBoundsCoord})$~~

### 13.4.5 Local Functions

findPath : $NetworkGraph \times String \times String \times String \rightarrow ShortestRoute$ Type

- Description: if input exists for stopfilepath then route will be found based on that data and return *ShortestRouteStop* else route will be found based on the filepath data and return *ShortestRouteTrace*..

~~findnode : $NetworkGraph \times$ tuple of $(\mathbb{R}, \mathbb{R}) \rightarrow \mathbb{N}$~~

- ~~Description : *NetworkGraph.getNearestNode*(GPS coordinate) using the *NetworkGraph* access routine the function returns node number.~~

~~pathfinder : $NetworkGraph \times$ String $\times \mathbb{N} \times \mathbb{N} \rightarrow seqof\mathbb{N}$~~
~~deactivateedges : $NetworkGraph \times onlinetransitdatabase \rightarrow NetworkGraph$~~

- ~~Description : deactivate edges inaccessible by bus according to the exported information.~~

# 14 MIS of Activity Location

## 14.1 Template Module

Activity Location

## 14.2 Uses

None.

## 14.3 Syntax

### 14.3.1 Exported Constants

None

### 14.3.2 Exported Type

ActivityLocation = ?

### 14.3.3 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| new ActivityLocation | name: String, lat: Latitude ($\mathbb{R}$), lon: Longitude ($\mathbb{R}$), amenity: String | Activity Location | |

## 14.4 Semantics

### 14.4.1 State Variables

*name* : String of Activity Location name
*lat* : latitude of Activity Location ($\mathbb{R}$)
*lon* : longitude of Activity Location ($\mathbb{R}$)
*amenity* : String of Activity Location description

### 14.4.2 Environment Variables

None.

### 14.4.3 Assumptions

- Assume the values for name, latitude, longitude and amenity provided are valid

### 14.4.4 Access Routine Semantics

None

# 15 MIS of ~~Stop~~ Point

## 15.1 Template Module

Point

## 15.2 Uses

None.

## 15.3 Syntax

### 15.3.1 Exported Constants

None

### 15.3.2 Exported Type

Point = ?

### 15.3.3 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| new Point | lat: Latitude ($\mathbb{R}$), lon: Longitude ($\mathbb{R}$), time($\mathbb{R}$), mode (Enum), ID($\mathbb{N}$), | Point | |

## 15.4 Semantics

### 15.4.1 State Variables

*lat* : latitude of Stop Point ($\mathbb{R}$)
*lon* : longitude of Stop Point ($\mathbb{R}$)
*time* time the point was pinged
*mode* : enum of ($\mathbb{N}$, $\mathbb{N}$, $\mathbb{N}$) - modes for episodes, STOP = 0, WALK = 1, DRIVE = 10
*ID* : ID number represent id of trace ($\mathbb{N}$)

### 15.4.2 Environment Variables

None.

### 15.4.3 Assumptions

- Assume the values for latitude, longitude, time, mode, and ID provided are valid

### 15.4.4 Access Routine Semantics

None

# 16 MIS of Generating Activity Locations

## 16.1 Module

Fetch Activity Locations

## 16.2 Uses

~~Uses no other modules~~ Activity Location, Point, Transformation

## 16.3 Syntax

### 16.3.1 Exported Constants

None.

### 16.3.2 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| FetchActivityLocations | input CSV file path(String), output file path(String), radiuse($\mathbb{R}$) | output CSV file generated | N/A |

## 16.4 Semantics

### 16.4.1 State Variables

Not applicable

### 16.4.2 Environment Variables

onlinenetworkdatabase: connection to online database to retrieve information layers of intersections, roads, and paths, and activity locations.

### 16.4.3 Assumptions

- Collection of stop locations longitudes and latitudes that are correct and valid
- Tolerance given is a number
- Onlineworkdatabase is accurate and accessible within 10 minutes

### 16.4.4 Access Routine Semantics

fetchStopAL(listOfStops)():

- transition: Not applicable
- output: output CSV file generated collection of tuples (Point, Activity Location)
- exception: N/A

### 16.4.5 Local Functions

fetchActivityLocations : list of Point Objects, tol

- Description: Computes activity location given the latitude and longitude of a specific stop location by fetching activity locations in the tol radius from onlineworkdatabase and ~~returns list of activity location names and latitudes and longitudes~~ returns list of ActivityLocation objects

# 17 MIS of Module Mapping

## 17.1 Template Module

Mapping

## 17.2 Uses

Network Graph, Shortest Route, Alternative Route, Fetch Activity Locations, Transformation, Point, Activity Location

## 17.3 Syntax

### 17.3.1 Exported Constants

None.

### 17.3.2 Exported Type

Display = ?

### 17.3.3 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| CreateRouteMap | *NetworkGraph*, object of type *ShortestRoute* or *AlternativeRoute*, *String* | HTML FILE | InvalidRoute ,<br>EmptyFilePath,<br>InvalidMappingFile |
| CreateActivityLocationMap | *String*, *String*, *String* | HTML FILE | EmptyFilePath,<br>InvalidMappingFile |
| CreateEpisodeMap | *String*, *String* | HTML FILE | EmptyFilePath,<br>InvalidMappingFile |
| ~~new Mapping~~ | ~~*NetworkGraph*, seq of *ShortestRoute*, seq of *AlternativeRoute*, seq of ℕ, seq of ℕ~~ | ~~HTML FILE~~ | - |
| ~~updateMapping~~ | ~~seq of *ShortestRoute*, seq of *AlternativeRoute*, seq of ℕ, seq of ℕ~~ | ~~HTML FILE~~ | - |

## 17.4 Semantics

### 17.4.1 State Variables

*map* : HTML FILE - displays mapped routes and points.

### 17.4.2 Environment Variables

IOmanager : it is the access point between the local memory and the application. It is used when saving and updating HTML files locally.

### 17.4.3 Assumptions

- Assume all the inputs are valid as they are outputs of previous modules.

### 17.4.4 Access Routine Semantics

CreateRouteMap(network, route, savepath):

- transition: None.
- output: Create a *Mapping* object using *mapBuilder(network, route)* and saves it locally with *IOmanager*
- exception: ¬(route of type *ShortestRoute* ∨ *AlternativeRoute*) → *InvalidRoute* ∨ (savepath == "") → *EmptyFilePath* ∨ (savepath is not an html type) → *InvalidMappingFile.*

CreateActivityLocationMap(activitylocationsfilepath, stopfilepath, savepath):

- transition: None.
- output: Create a *Mapping* object using *mapBuilderPoints(activitylocationsfilepath, stopfilepath)* and saves it locally with *IOmanager*
- exception: (savepath == "") → *EmptyFilePath* ∨ (savepath is not an html type) → *InvalidMappingFile.*

CreateEpisodeMap(episodefilepath, savepath):

- transition: None.
- output: Create a *Mapping* object using *mapBuilderPoints(episodefilepath)* and saves it locally with *IOmanager*
- exception: (savepath == "") → *EmptyFilePath* ∨ (savepath is not an html type) → *InvalidMappingFile.*

~~new Mapping(graph, shortest, alternative, activitylocations, episodes):~~

- transition: ~~None.~~
- output: ~~Create a *Mapping* object and saves it locally with *IOmanager*~~
- exception: ~~None~~

~~updateMapping(shortest, alternative, activitylocations, episodes):~~

- transition: ~~Add new elements to *Mapping* object and saves it locally with *IOmanager*~~
- output: ~~None.~~
- exception: ~~None~~

### 17.4.5 Local Functions

mapBuilder : *NetworkGraph* × *object* ⇒ *map object*

- Description: visualizes the network as the map basis and maps the routes on top of it. All stored within a mapping object of choice.

mapBuilderPoint : *String* × *String* ⇒ *map object*

- Description: visualizes two layers of inputted gps points on a mapping object of choice.

# References

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995. URL http://citeseer.ist.psu.edu/428727.html.

# 18 Appendix

Extra information if required