From Trajectories to Semantic Mobility Networks - Hands-On SBO survey dataset

UNIPI-InfoLab-TR-2015-02, Technical Report Series

Information Management Lab (InfoLab)

Department of Informatics

University of Piraeus

Stelios Sideridis
Nikos Pelekis
Yannis Theodoridis

March 2015

Table of Contents

From Diaries and GPS logs to Semantic Mobility Timelines

This document describes the SBO survey dataset, its transformation to semantic mobility timelines (NOTE: in the sequel, we may use the term semantic trajectory (episode) but we mean semantic mobility timeline (semantic trajectory, respectively)), as defined by (Pelekis et al. 2013). The report also includes a case study upon this unique dataset that also stands as a hands-on experience describing the framework of a data-type system for semantic trajectories with its associated query language, as envisioned in (Pelekis et al. 2013).

All the work has been done by extending the HERMES MOD environment (URL: http://infolab.cs.unipi.gr/hermes/).

The initial dataset is composed of two sources of data. The '*GPS data*', which are GPS records per user derived from their gps-enabled devices; and '*Diaries*' in which each user documented semantic information about his/her trips. A trip for a user is a movement from a point A to another point B for a specific purpose. This document focuses on a subset from the above sources that corresponds to 186 users who documented their trips.

The 'GPS data'

The total number of unprocessed (raw) gps records is 7698808, meaning an average of 41391.44 gps records per user. These gps records covering a period of 730 days, from 13-4-2006 to 14-4-2008.Each gps record is tagged with a number showing the trip in which the gps record belongs to. The gps records were already tagged with trip numbers. The total number of trips is 3474 meaning an average of 18.677 trips per user. Moreover every gps record has a gps validity letter ('A' for valid gps records and 'V' for not valid, validity had to do with the number of satellites being at least 3 for signal stability.

These gps records were processed to produce trajectories as follows:

We take into account only valid gps records so we have 7599702 gps records to process. Then a simple reconstruction algorithm was used to group gps records into trajectories (meaningful subsequences of gps records) per user. In detail, a new trajectory for the user is created when a spatial or a temporal threshold is exceeded, when examining successive gps records. Obviously these thresholds identify corresponding spatial or temporal gaps between successive gps records. The used parameters were 5000 meters for spatial threshold, and 4 hours for temporal threshold. This simple algorithm orders in time gps records for each user. Then iterates through these gps records and if space or time constraint between two consecutive gps records is met then a new trajectory is formed. Additionally, any consecutive gps records that have the same spatial position (or temporal, more rarely) are discarded.

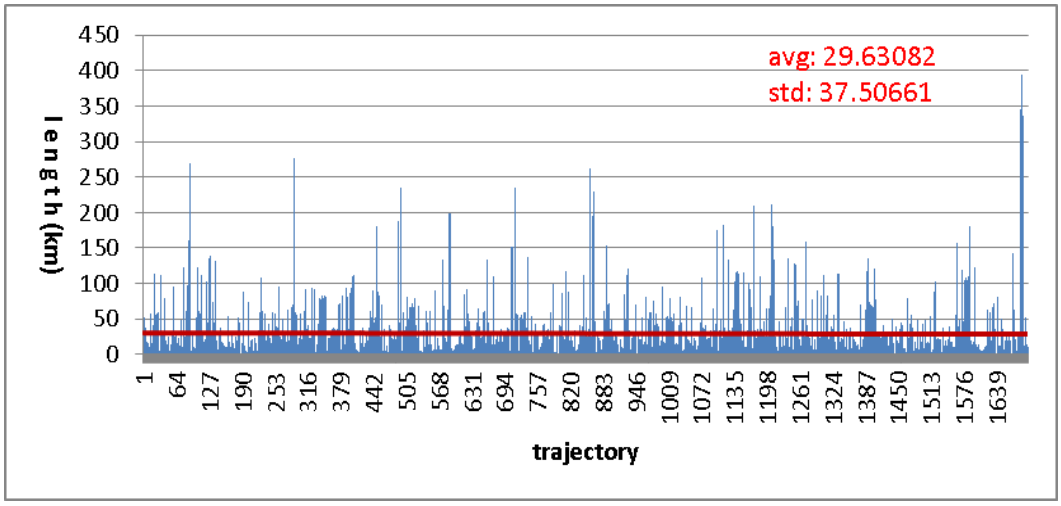
The above is executed through the following procedure code (its explanation follows in the next section):

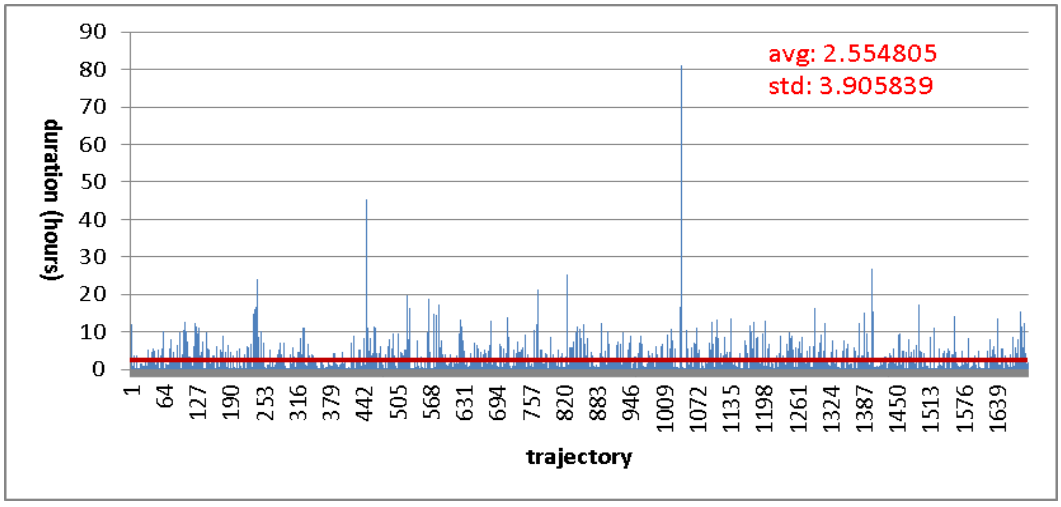
```
begin
        sem_reconstruct.reconstructtrajectories('belg_users_gps', 4326, 'sem_mpoint', 5000, 14400);
end;
```

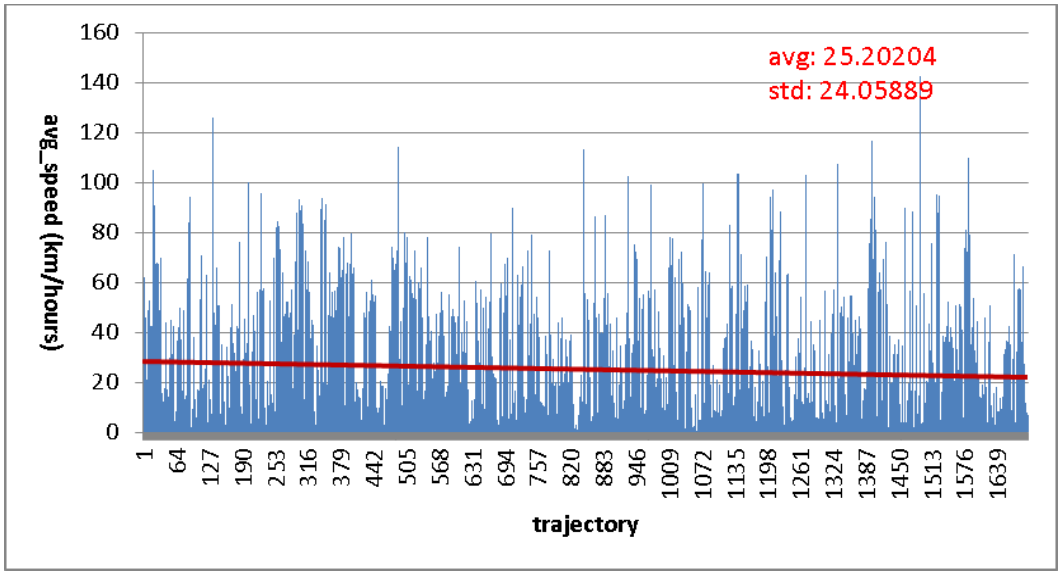
The algorithm outputted 1709 trajectories consisting of 3542205 gps records, meaning that we had many consecutive gps records that had the same spatial position. From those trajectories only 1698 were valid (i.e. those having more than one gps records) corresponding to 3542194 gps records. This gives an average of2086.0977 points per trajectory. Total number of segments is 3540496. The above trajectories give an average of 9.129 trajectories for each user. The total length of the 1698 trajectories is 50313133.007449 meters meaning an average of 29630.82038 meters per trajectory. The average speed for the 1698 trajectories is 7.000565 m/s.

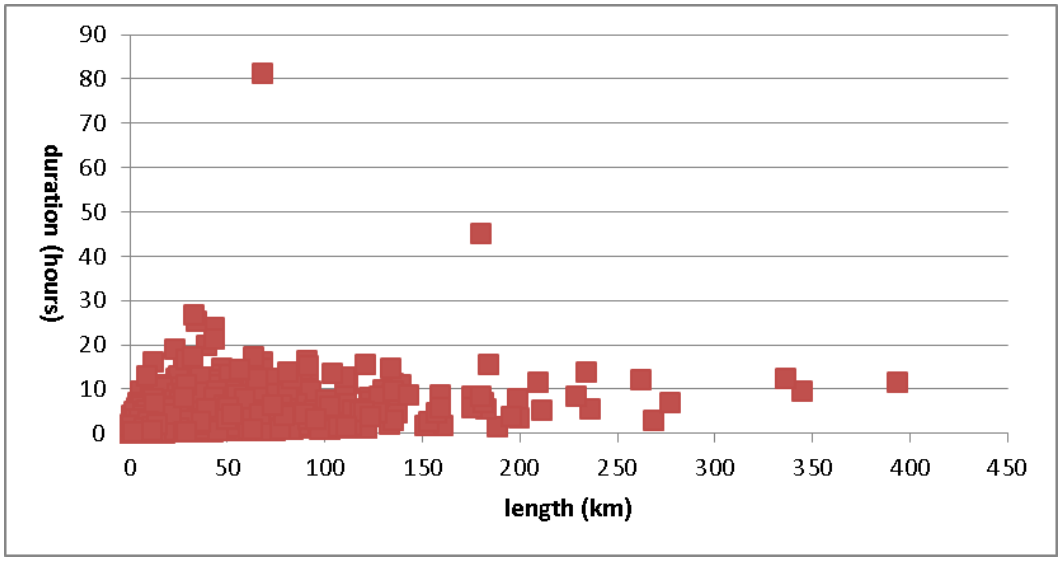
Those trajectories cover a rectangle space with Minimum Bounding Rectangle (MBR): [(2.598423, 49.926288), (5.878673, 52.351493)].

The following diagrams give a more graphical/statistical description of the produced raw trajectory database.






avg: 25.20204
std: 24.05889
avg_speed (km/hours)
trajectory
0
20
40
60
80
100
120
140
160
1
64
127
190
253
316
379
442
505
568
631
694
757
820
883
946
1009
1072
1135
1198
1261
1324
1387
1450
1513
1576
1639


duration (hours)
length (km)
0
10
20
30
40
50
60
70
80
90
0
50
100
150
200
250
300
350
400
450

The Semantic Trajectory Database

Before describing the 'Diaries' dataset and the way we processed it, let us first describe the Semantic Trajectory Database, and the required data types that were used in our schema. A semantic trajectory is considered as a time-ordered set of objects called episodes. Each episode is characterized as a MOVE when the person of the trajectory is moving from a place to another or as a STOP when the person is considered stationery. Additionally, each episode has semantic information about the person's activity during the episode. Also there is a link between each episode and the corresponding raw sub-trajectory (sequences of <x, y, t>). The word sub-trajectory emphasizes the fact that the whole person's trajectory is broken into sub-trajectories (episodes).

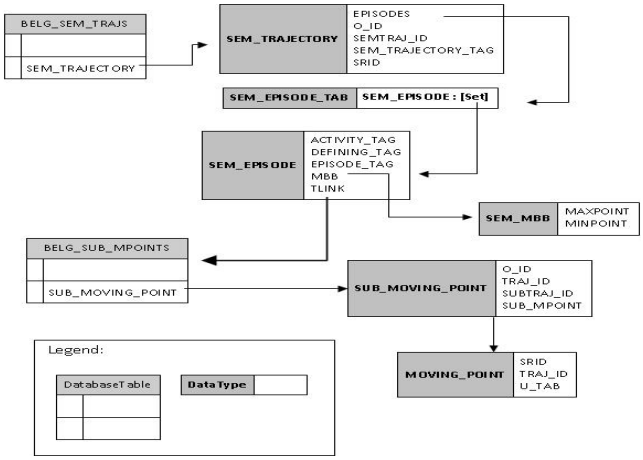


Figure 1: Semantic Trajectory Database Schema

In the above diagram we see two object tables 'BELG_SEM_TRAJS' and 'BELG_SUB_MPOINTS' along with the definitions of the objects, holding all the produced information. Every object has already a number of useful methods that will be continuously expanded.

- SUB_MOVING_POINT: has all the methods of MOVING_POINT object (i.e. the trajectory datatype in HERMES) plus a method *getsemmbb()* which returns its MBB as a SEM_MBB object.
- SEM_MBB has the following methods:
    - *sem_mbb(geomsdo_geometry, period tau_tll.d_period_sec)*: This function takes as input a geometry object (other than point or line) and a time period. It constructs a sem_mbb instance that is returned. Default constructor also has two parameters so be careful.
    - *area(srid):* returns the spatial area of the MBB
    - *duration()*: returns the temporal area of the MBB
    - *getrectangle(srid):* returns the MBR as an MDSYS.GEOMETRY object
    - *intersects(inmbbsem_mbb)*: This function takes as input another *sem_mbb* object. It checks whether the intersection between the two *sem_mbb* objects in all dimensions (i.e. x, y, t) is the empty set or not. If not, it returns true otherwise it returns false.
- SEM_EPISODE:
    - *duration()*: returns the temporal area of the episode

- *sim_episodes(e sem_episode, dbtable varchar2, indxprefix varchar2:=null, lamda number:=0.5, weight number_array:=(0.333,0.333,0.333))*: This function takes as input another sem_episode object, a dataset table for calculating required global values, an optional index prefix if such an index (i.e. STB-tree) has been built, an optional value for the λ (i.e. lamda) parameter and an optional 3 number array for the w (i.e. weight) parameter. It returns a number that defines the distance between this episode object and the input episode e.

- SEM_TRAJECTORY
  - *num_of_stops():* returns the number of STOP episodes it contains
  - *num_of_moves():* returns the number of MOVE episodes it contains
  - *num_of_episodes(tag varchar2, distinct varchar2)*: This function takes as input a "*tag*" string (less than 1000 chars) of the form "tag1+tag2+....+tagn" (i.e. implying a concatenated set of (sub-)strings), and a "*distinct*" string that can be either "yes" or "no" (i.e. implying a boolean flag). It returns the number of episodes (distinct or not, depending on the use of the flag) of the semantic trajectory that includes tags LIKE the given ones. In this case, “LIKE” implies pattern-matching per input tag#.
  - *getMBB():* returns the MBB of the semantic trajectory
  - *sem_stops():* returns the STOP episodes as a nested table object
  - *sem_moves():*returns the MOVE episodes as a nested table object
  - *episodes_with(tag varchar2) return sem_episode_tab*: This function takes as input a "*tag*" string (less than 1000 chars) of the form "tag1+tag2+....+tagn" (i.e. implying a concatenated set of (sub-)strings). It returns a nested table of those episodes that have tags LIKE the given ones. In this case, “LIKE” implies pattern-matching per input tag#. In case where an episode matches multiple times with some input tags, thisis returned only once. A null collection is returned when none episode is found.
  - c*onfined_in(geomsdo_geometry, period tau_tll.d_period_sec, tag varchar2)*: This function takes as input a spatial geometry, a temporal period and a "*tag*" string (less than 1000 chars) of the form "tag1+tag2+....+tagn" (i.e. implying a concatenated set of (sub-)strings). It returns a *sem_trajectory* object, whose episodes are overlapping: a) spatially with the *“geom”* parameter, b) temporally with *“period”* parameter and textually with the *“tag”* parameter. If one or more of the three paramaters is null, then the function assumes that no confinement is requested in the corresponding dimension, as such it continues with the rest. In other words, if (for instance) the *“geom”* is null then the function uses it as the user gave the MBR of the whole semantic trajectory. Similarly, if the *“period”* is null then the function uses it as the user gave the period of the whole lifespan of the semantic trajectory. If *“tag”* is null, it is like the user requests for all possible tag matchings (i.e. like giving the “%” wild character as input) If none such episode is found, then a semantic trajectory is returned with an empty nested table of episodes.
  - *sim_trajectories (trsem_trajectory, dbtable varchar2,indxprefix varchar2:=null, lamda number:=0.5, weight number_array:=(0.333,0.333,0.333)):* This function takes as input another *sem_trajectory* object, a dataset table for calculating required global values, an optional index prefix if such an index (i.e. STB-tree) has been built, an optional value for

the λ (i.e. lamda) parameter and an optional 3 number array for the *w* (i.e. weight) parameter. It returns a number that defines the spatio-textual distance between this semantic trajectory object and the input semantic trajectory.

Note that with a slight modification each episode may hold multiple values on its semantic fields (e.g. episode_tag and activity_tag, or even a sequence of probabilities). With this Semantic Trajectory Database (STD) in hand, we can pose many useful queries and apply many algorithms that contain any combination of spatial, temporal and semantic predicates. For the moment we have implemented the following variations of range queries:

- *stb_range_episodes(episodeType, geometry)*: returns episodes of type episodeType (e.g. STOP or MOVE) that intersect the given geometry.
- *stb_range_episodes(episodeType, temporalPeriod)*: returns episodes of type episodeType (e.g. STOP or MOVE) that intersect the given temporal period.
- *stb_range_episodes(episodeType, geometry, temporalPeriod)*: returns episodes of type episodeType (STOP or MOVE) that intersect a geometry and a temporal period.
- *stb_range_episodes(fromGeometry, toGeometry, temporalPeriod)*: returns MOVE episodes that started (ended) from (to) the fromGeometry (toGeometry) geometry, respectively, inside the given temporal period.
- *stb_range_episodes(from_stopsem_episode, to_stopsem_episode, via_movesem_episode, stbtreeprefix varchar2) return sem_episode_tab*: This method returns MOVE episodes. It takes as input arguments three episodes. The returned MOVE episodes obey to the input parameters, that is, they begin from the *from_stop* episode, they end to the *to_stop* episode and they have similar attributes as the *via_move* episode. Input episodes can be null, meaning that the corresponding constraint is not applied. For example, if the *from_stop* episode is null, then this method would return all move episodes from the dataset that ended to the *to_stop* episode, after having moved according to the via_move episode. Moreover, each input episode can have its text or its spatio-temporal attributes set to null, meaning again that no corresponding constraint is applied. For example,the *from_stop* episode parameter can have its spatiotemporal attribute (sem_mbb) set to null and the *to_stop* episode can have all or some of its text attributes (i.e. defining_tag, episode_tag or activity_tag) set to null, while the *via_move* is null. In this case, the method returns MOVE episodes that began from the *from_stop* episode, where only text constraints are applied (e.g. Home, eating), went to the *to_stop* episode, where only spatiotemporal and some text constraints are applied (e.g.areaX, Work), using any in-between existing MOVE episode in the dataset.

- *stb_patterns (inputepisodes, inputchars, stbtreeprefix) return integer_nt*: This method returns integers corresponding to semantic trajectory identifiers that follow a spatio-temporal-textual pattern. The returned semantic trajectories follow the movement pattern described by the arguments *inputepisodes* and *inputchars*. The argument *inputepisodes* holds the sequence of episodes in time order that constitute the pattern to be examined (e.g. episode1, episode2, episode5). Each episode in the sequence may have its textual attributes or its spatiotemporal attributes set or not (null), meaning that an attribute must be taken or not into account when current episode is examined. Additionally, between two consecutive episodes in the input sequence, others episodes that need not to be examined may exist. This is realized with the use of the *inputchars* argument, which is an array of characters each of which can be either '>' or '*'.

This array is synchronized with the array of episodes (i.e. *inputepisodes*), as it implies whether two given episodes should be consecutive or not (i.e. others are in-between them). More specifically, when the '>' character is found, then the two episodes must be consecutive in the returned trajectory, while when the '*' character is present, then between the two episodes other episodes may exist.

The above range queries can be run efficiently in our STD as we have defined and implemented a specialized index for semantic trajectories named STB-tree (Pelekis and Theodoridis, 2013).

Semantic reconstruction in Hermes MOD

In order to support semantic reconstruction in Hermes, we have developed a special library (package), called *sem_reconstruct* (for semantic reconstruction), which supports this goal and is composed of the following procedures and functions:

- *reconstructtrajectories(sourcetblgps, srid, targettblmpoints, spacegapmet, timegapsec):* This procedure takes as input a table with gps points which should have at least columns about user id, longitude, latitude and timestamp and reconstructs trajectories (that will be stored in *targettblmpoints* table with *srid* as its spatial reference system) by producing a new trajectory whenever a space gap and a time gap pair of thresholds between succesive gps points are exceeded.
- *belgdiariestosemtrajs:* This procedure is made particularly for SBO-survey dataset. It takes input from the imported excel files and produces semantic trajectories and the corresponding sub trajectories.
- stopfinderinputfile(o_id, traj_id, mpoint, subtraj_id): This procedure takes as input a trajectory *mpoint* with its *object_id*, *trajectory_id* and an optional *sub-trjectory_id*. It outputs a .dat file in the IO directory (where user must have read-write rights) in the form that is required by T-Optics Stop detection algorithm. It can be called multiple times inside a pl/sql loop block, to output many trajectories (as will be exemplified in the subsequent section) .
- stopfinder(dir, conf): This procedure calls the T-Optics (i.e. stopfinder) algorithm (Zimmerman et al. 2009).
- *rawtrajs2semtrajs(inputtblstopseqs, inputtblmpoints, outputtblsubmpoints, outputtblsemmpoints):* This procedure takes as input a table of Stops found from T-optics algorithm, a table name of raw trajectories on which T-Optics run and two output tables for sub-trajectories and semantic trajectories, respectively. It transforms raw trajectories to semantic trajectories based on the T-Optics findings.
- *stops2semtrajs(inputtblstopseqs, inputtblsemmpoints, outputtblsubmpoints, outputtblsemmpoints):* This procedure takes as input a table of Stops found from T-optics algorithm and a table name of semantic trajectories upon the MOVE episodes of which the T-Optics run. There are also two output tables for sub-trajectories and semantic trajectories, respectively. It modifies the given semantic trajectories and returns again semantic trajectories based on the T-Optics findings on the MOVE episodes of the initial ones.

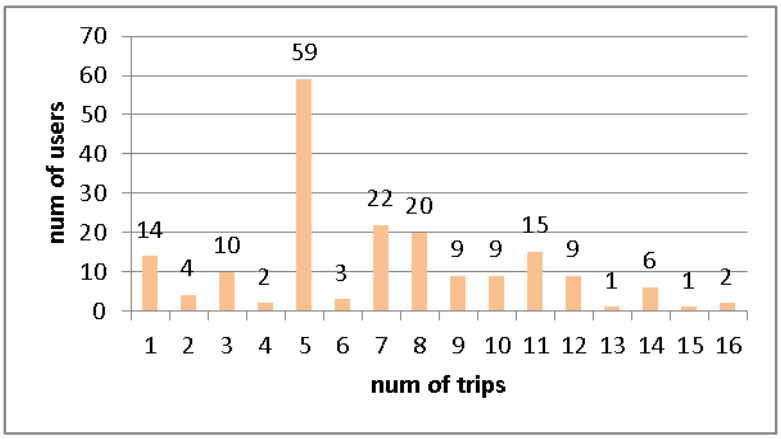
- *pois_probability (mbb, srid, is4visualize, poitable, out bestpoitag):* Finds (and visualizes) POIs, each of which is annotated with a probability of being the POI that a Stop activity took place. The returned POIs are within some given MBB. The function also returns the *bestpoitag* parameter with the tag that corresponds to the POI having the biggest probability.
- *nn_pois(mbb, k, is4visualize):* Find the K -Nearest-Neighbor (K-NN) Points-Of-Interest (POI) w.r.t. the centroid of the MBR of an episode.
- *annotate_episodes(semtrajs, poitable):* This procedure takes a table of semantic trajectories and a table of POIs as input and then updates tags of episodes for each semantic trajectory.

The 'Diaries'

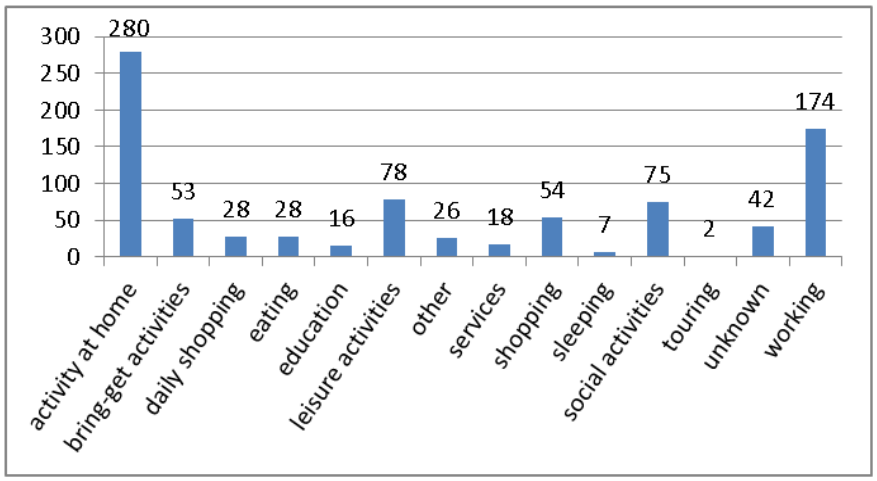
Diaries are in the form of trips with semantic information for each trip. This information is spread into a number of excel files. The dataset contains 881 trips for the 186 users. This gives an average of 4.73655 trips per user or 2.93 trips per user per day. If we group those trips per user and per day then we get 300 groups of trips. Moreover each trip has a connection with the trip number found in 'GPS data' source for each user, thus we can match gps records with trips in the diaries. Additionally, for every trip there is semantic information about origin and destination, the activity at destination and the used transportation mean.

Those diaries were processed to produce semantic trajectories as follows:

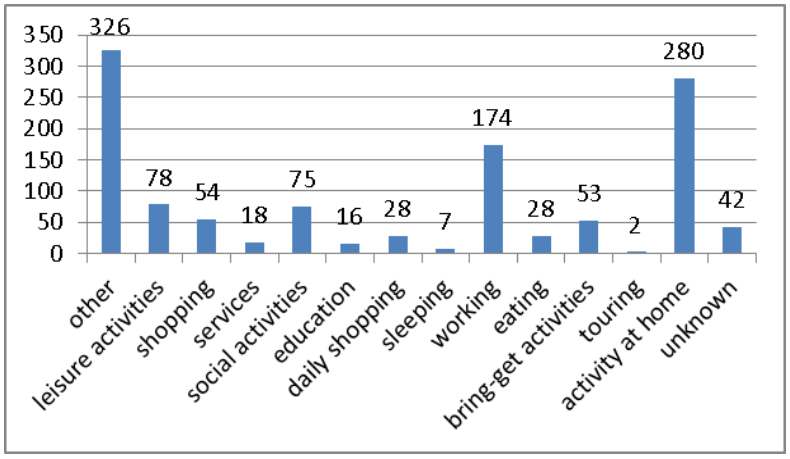
Procedure 'diaries to semantic trajectories' combine initially, information from the above sources and for each user iterates through his trips. A diary trip is considered a MOVE episode, so STOP episodes must be inserted before and after that trip. Semantic trajectories are created each time a new user or a new day is met (also when there is discontinuity between trip numbers). The above procedure results in creating 300 semantic trajectories, which is 1.6129 semantic trajectories in average per user. Additionally, by combining information from the 'GPS data' source, for every episode it builds a corresponding trajectory (a sub-trajectory if you like). This can be done, as we know for each trip (MOVE episode) the beginning and ending gps record ids. In the following figure the distribution of the number of users w.r.t. number of trips is shown.

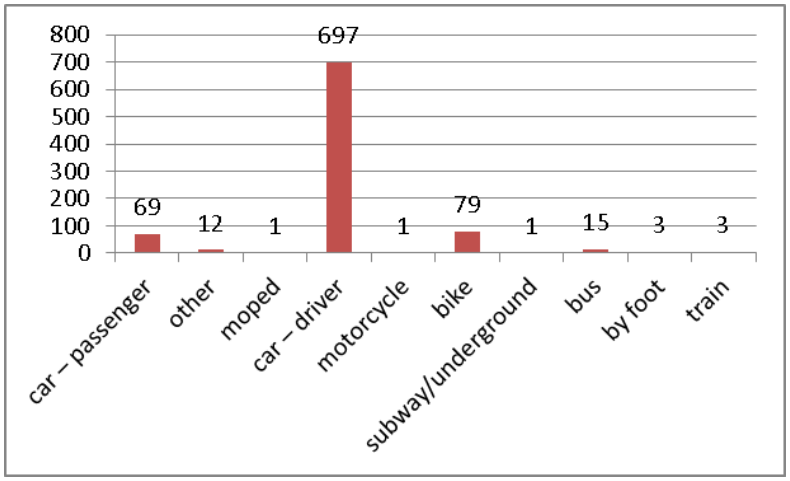


The total number of episodes is 2062 (881 MOVE and 1181 STOP episodes), an average of 6.87333 episodes for each semantic trajectory. There exist 14 categories for STOP episode activities (due to different purposes at destination) and 10 for MOVE episode activities (due to different transportation means). Below we see how the number of trips per activity is distributed.

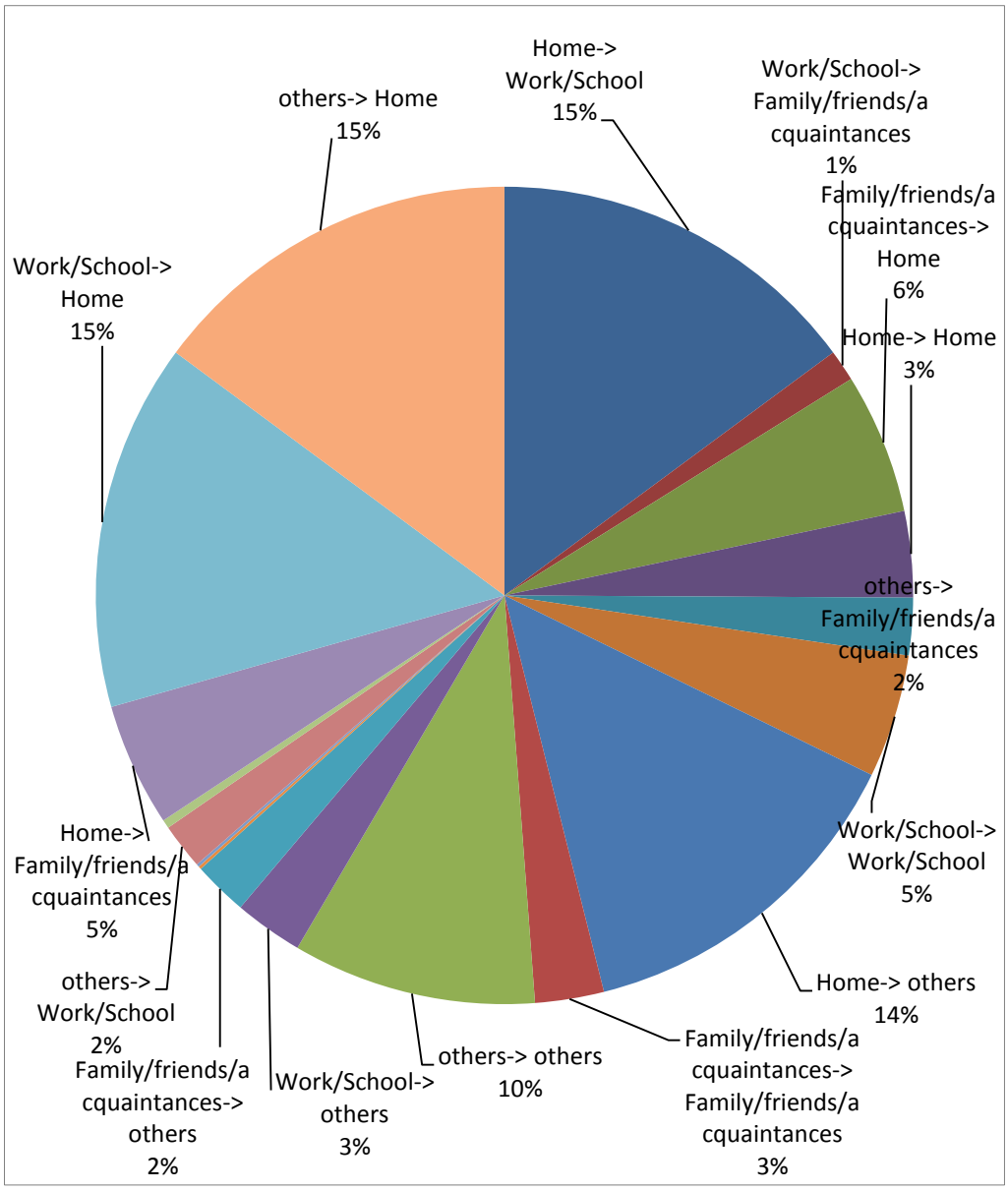


For STOP episodes the distribution of activities (STOP purposes) and the distribution of transportation mean for MOVE episodes, are shown in the following two figures.





The following chart shows the distribution between (origin, destination) types of MOVE episodes.



Each episode has a corresponding sub-trajectory (either STOP or MOVE episode). Those sub-trajectories, 2062 in number, have a total of 1050492 segments (509.4529 in average per sub-trajectory) which are separated in 196675 for STOP episodes and 853817 for MOVE episodes. That gives an average of 166.5326 segments for STOP episodes and of 969.1452 segments for MOVE episodes. Below is a table with statistics about STOP and MOVE episodes.

	STOP	MOVE
Distance Covered (meters)	2224655.2036	14445367.99321
Average	1883.70466	16396.558
Duration (seconds)	29576280	1016626
Average	25043.4208	1153.9455
Average Speed (m/s)	0.22406	12.9139

The above described semantic trajectory database is based solely on users' diaries. To put differently, the destination of a trip corresponds to an ActivitySTOP, as there is a declared purpose for this. The next step is to include incident STOP episodes into the semantic trajectories found from the above procedure. Towards this direction, we have employed the T-Optics algorithm (Zimmermann et al. 2009). T-Optics is a "STOP-finder" algorithm that takes as input one raw trajectory and outputs sequences of successive points from that trajectory that form areas where the object is considered stationery. Thus in our setting, when T-Optics is applied on a raw trajectory it discovers sub-trajectoriesthat correspond to STOP episodes. Giving as input to T-Optics all trajectories that correspond to MOVE episodes from the semantic trajectories, we discover intermediate STOP areas. In other words, we break each MOVE episode to more than one MOVE episodes, if the T-Optics algorithm identifies intermediate STOP areas. The number of semantic trajectories in the database is still 300 but now these consist of 4634 episodes in total (2530 STOPs and 2104 MOVEs). Similarly the corresponding sub-trajectories have a total of 1050492 segments (226.69 in average per sub-trajectory),which are separated in 226747 for STOP episodes and 823745 for MOVE episodes.

The above procedure takes place in a series of steps. First, for each MOVE episode we output corresponding trajectories to a T-Optics format, by calling the following procedure:

```
declare
begin
for cur in ( select b.object_id,b.traj_id,b.mpoint
            from stopfinder_mpoints b) loop
        sem_reconstruct.stopfinderinputfile(cur.object_id,cur.traj_id,cur.mpoint,null);
end loop;
end;
```

Then we call the T-Optics algorithm on the directory (*mydir*) where we outputed the previous files:

```
declare
mydir varchar2(2000);
begin
        select directory_path into mydir from dba_directories where directory_name='IO';
        sem_reconstruct.stopfinder(mydir,'config.ini');--procedure
end;
```

where the file *config.ini* is the configuration file needed by the algorithm.

config.ini

```
1   eps=100
2   minTime=20
3   columns=X,Y,T
4   numberColumns=3
5   delimiter=
6   dataset=
7   header=yes
8   cartesian=no
9   output=
10  outputStop=/stop.csv
11  orderedList=
12  alpha=50
```

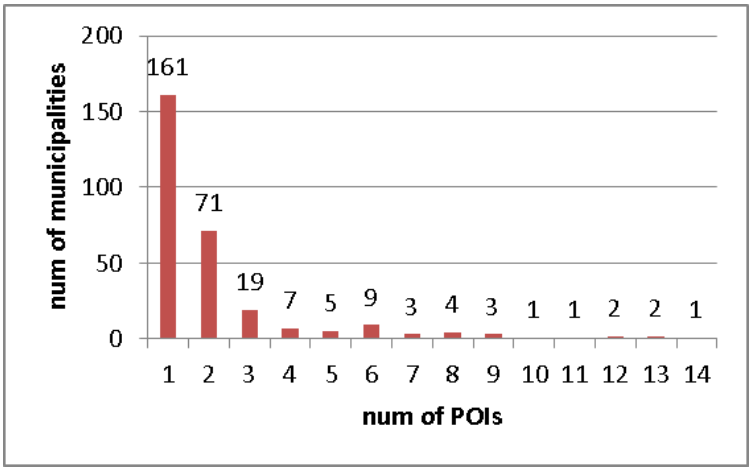
Figure 2: Configuration file for T-Optics stop detection algorithm

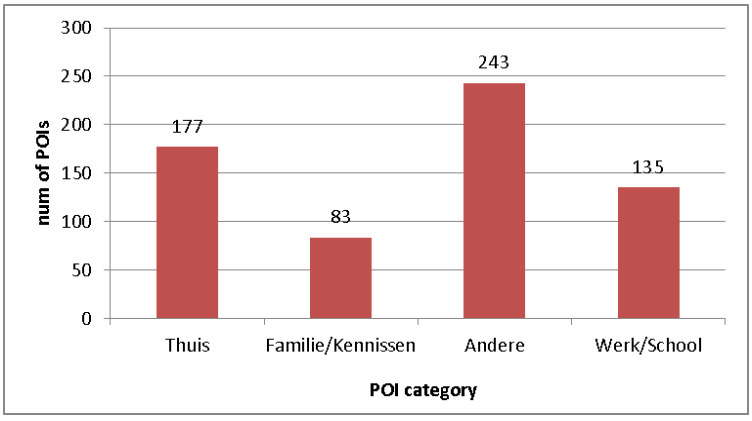
The algorithm outputs a file per trajectory consisting of points belonging to stops (a trajectory may have multiple stops). These files are imported in the HERMES MOD. For every point of a stop the trajectory identifier and the stop identifier must be known. Finally, we call the following procedure to transform the original semantic trajectories to new semantic trajectories based on T-Optics results:

```
declare
begin
        sem_reconstruct.stops2semtrajs('stops_found', 'in_trajs','out_submpoints','out_semtrajs')
end;
```

	STOP	MOVE
Distance Covered (meters)	2337682.39476	14332340.8021
Average	923.9851	6811.949
Duration (seconds)	29698205	894701
Average	11738.4209	425.2381
Average Speed (m/s)	1.7534	10.93021

The number of POIs (Point Of Interest) in the 'Diaries' source is 638 and are categorized in 4 categories (Home,Work/School,Family/friends/acquaintances,others). These POIs are spread across 289 municipalities (NOTE that for us, a municipality is simply the MBR of all points inside a municipality, as we do not have such a spatial database). The following figures show the distribution of POIs.





The Semantic Trajectory Cube

Having designed such a database for semantic trajectories we move one step further to introduce a data warehouse for semantic trajectories. As we saw the main entities of a semantic trajectory are their stop and move episodes. Quite naturally those two entities become our facts in the new data warehouse scheme.

The new data warehouse schema (see Figure 2) is a constellation scheme consisting of five dimensions and two fact tables. The dimensions are space, time, user's profile, stop-semantics and move-semantics. Note that this relational representation of the DW corresponds to an instantiation of a Semantic Mobility Network (Graph), as this is defined by (Pelekis et al. 2013).

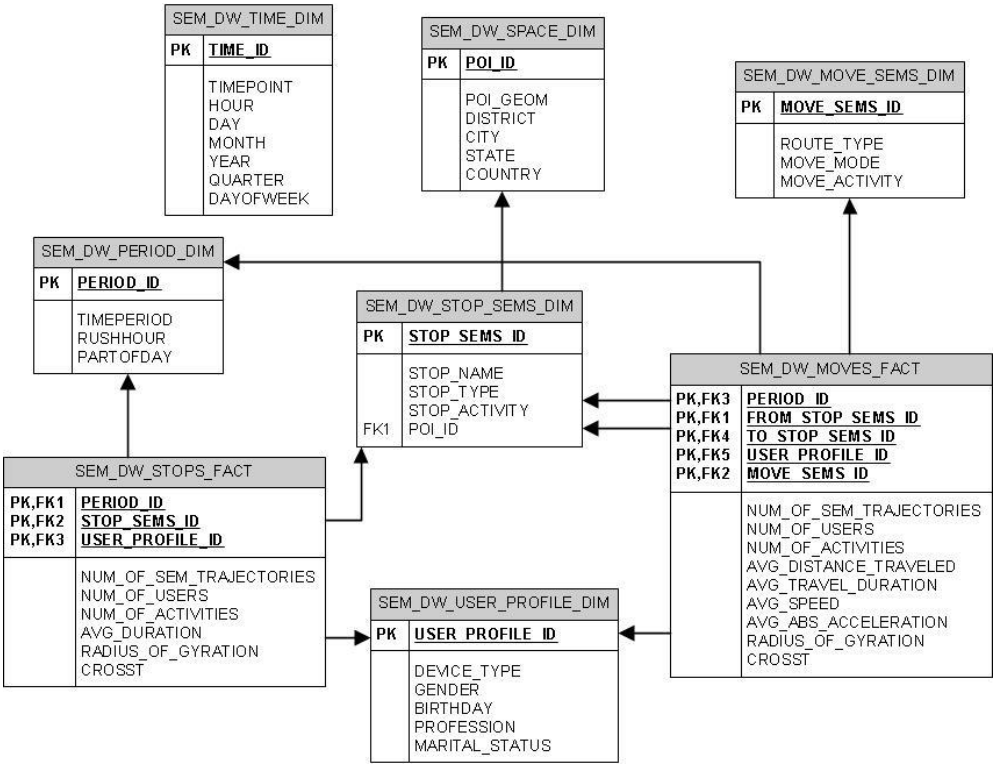


Figure 3: The Semantic Trajectory Data Warehouse schema

Measures in the above scheme are basic measures that can be easily extended. Such a DW mayenable the following kinds of analysis:

- who made a stop? when and where? what did she do during her stop?
- who made a movement? When and from/to where? How did she move and what did she do during her motion?

All these have been implemented in HERMES MOD where we deal with issues concerning the ETL process as well as other issues related to OLAP operations in the cube. Let us explain the above schema in more detail. First we define two base cells (*bc*) one for each fact table. For *stops-fact-table* we define bc_{stop}composed of three dimensions (time, user-profile and stop-semantics) and for *moves-fact-table* we define bc_{moves} composed of five dimensions (time, user-profile, move-semantics, from-stop-semantics and to-stop-semantics). These two base cells are the basis for splitting our multidimensional space of

semantic trajectories and form the base cuboid of our lattice. As an example, we discuss three basic measures named *num-of-sem-trajectories* which is calculated by counting all distinct semantic trajectories that found inside a base cell*, num-of-users* which is the number of different objects that are found inside the cell and *num-of-activities* which is the number of distinct activities of the moving objects in the base cell.

Let's describe the produced (so far) API of HERMES with which the user can manage the above DW scheme:

- `sdw.createSDW(sourceTablePrefix)` where parameter is a prefix for the necessary database objects that will be created. In different words, this procedure creates a graph (with the described relational representation), which is one of the graphs in the lattice of our semantic mobility cube, according to (Pelekis et al. 2013).

Similarly, with procedure `sdw.dropSDW(sourceTablePrefix)` user can drop the created data warehouse scheme.

To load dimensions, the user invokes procedure:

- `sdw.loaddimensions(sourceTablePrefix, poitable, rawtable, intervalsecs)` where the user must provide the DW prefix, the source for POIs for splitting the space dimension, the source for the minimum and maximum timestamp of the data (*rawtable*) and the interval between two times for splitting the temporal dimension.

Then when all dimension tables are filled in, the user can invoke ETL procedures to load data into the fact tables, from a semantic trajectory database. The ETL procedures are divided in two categories, one for loading every cell of the fact table for all semantic trajectories in STD and another for loading every semantic trajectory into the cells of the fact tables. Moreover for efficiency reasons these ETL procedures make use of the STB-TREE index already defined in STD. Thus for the first category user can call procedure:

- `sdw.cellstopsload(sdwTablePrefix, stbtreenodes, stbtreeleafs)` where parameters define the DW tables, the STB-TREE nodes table and STB-TREE leaves table.

which load the *stops-fact-table*. Loading the *moves-fact-table* can be loaded by calling the procedure:

- `sdw.cellmovesload(sdwTablePrefix, stbtreenodes, stbtreeleafs)` where parameters define the DW tables, the STB-TREE nodes table and STB-TREE leaves table.

If the user wants to use the second approach for the ETL process, that is to load every semantic trajectory in the cells of the fact tables, then user can call the following procedure to load *stops-fact-table*:

- `sdw.semtrajstopsload(sdwTablePrefix, semtrajs)` where the first parameter defines the DW scheme and the second the table where semantic trajectories are stored.

Similarly to load the *moves-fact-table* user can call procedure:

- `sdw.semtrajmovesload(sdwTablePrefix, semtrajs)` where the first parameter defines the DW scheme and the second the table where semantic trajectories are stored.

After successful loading of the fact tables, the user must invoke two other procedures which compute some auxiliary measures that will be used later on the OLAP operations. These are:

- `sdw.updateauxiliarystops(sdwTablePrefix)` for the *stops-fact-table,* where the parameter defines the DW scheme.

and

- `sdw.updateauxiliarymoves(sdwTablePrefix)` for the *moves-fact-table,* where the parameter defines the DW scheme.

On the classification/prediction of episodes' tags

In order to support classification/prediction tasks taken into advantage of the synchronized nature of this unique dataset, we may transform each episode to a multi-dimensional vector (as such producing a feature space), with dimensions derived properties from the episode such as: (1) the distance covered by the moving object in the episode, (2) episode's duration, (3) episode's top speed, (4) episode's average speed, (5) speed variation, (6) road type, (7) starting POI type, (8) end POI type, (9) episode's area (10) episode's radius of gyration, etc. The classification label of such a vector may be the tags annotating that episode (e.g. STOP, MOVE, CAR, WALK etc. The list of features can be easily extended with new features that may aid the discrimination process of the classifier.

The classification model is built from episodes features for the available semantic trajectories (training set). For the moment, features are calculated by the following procedure:

- `std.calcfeatures(outputtblfeatures, intblsemtrajs)` where the first parameter is the table to hold episodes properties and second parameter gives the input table of the available semantic trajectories..

Exploring SBO survey dataset

This section includes a case study upon the above described unique dataset that also stands as a hands-on experience upon the data-type system for semantic trajectories with its associated query language (Pelekis et al. 2013), the API of which we presented in the previous sections.

Querying the Semantic Trajectory Database

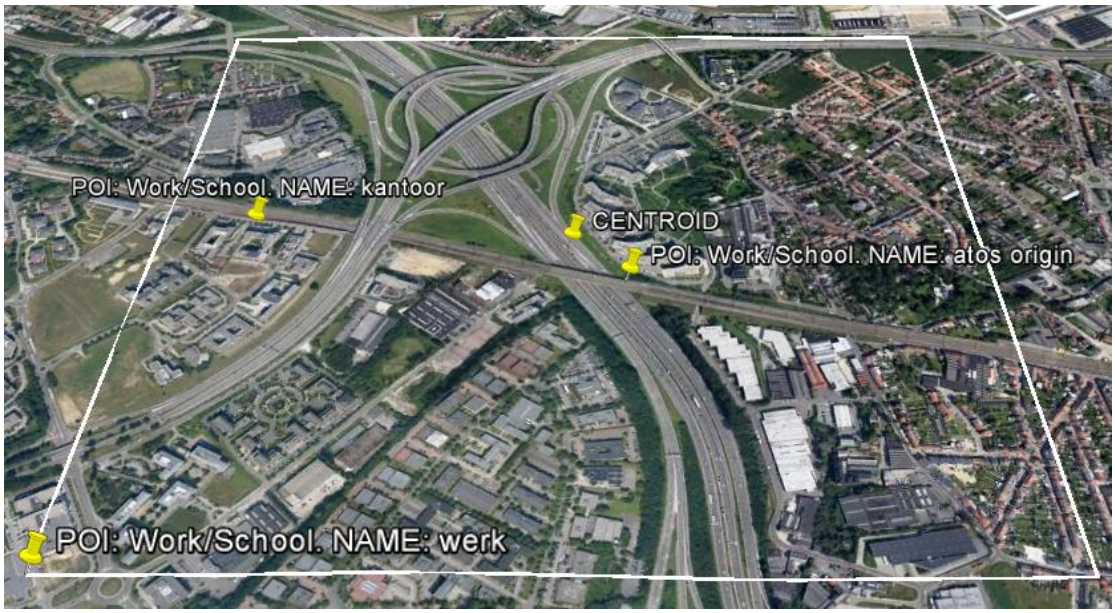
1. Find NN POI inside an episode

Description:

Find the K-Nearest-Neighbor (K-NN) Points-Of-Interest (POI) w.r.t. the centroid of the MBR of an episode. The returned POI (visualized in the subsequent figure) should be inside the MBR. Depending on this topological filter, the function may return less than (if any) K neighbors.

Code:

```
DECLARE
  sembb sem_mbb := sem_mbb (sem_st_point (5.503502, 50.953662,
tau_tll.d_timepoint_sec (2006, 5, 1, 5, 00, 00) ), sem_st_point (5.727093,
51.026928, tau_tll.d_timepoint_sec (2007, 6, 1, 5, 00, 00) ) );
BEGIN
  sem_reconstruct.nn_pois (sembb, 10, 'TRUE');
  COMMIT;
END;
```



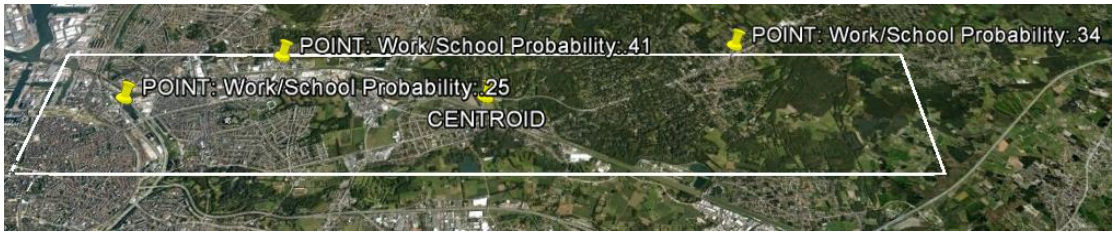
2. Probability of POIS within episode MBB

Description:

This function finds the POI that exist inside the MBR of an episode and for each of them it returns the probability of being the POI where the moving object performed an activity. The probability is inverse proportional to the distance of the POI from the centroid of the episode's MBR.

Code:

```
DECLARE
  sembb sem_mbb := sem_mbb (hermes.sem_st_point (4.415422, 51.218345,
tau_tll.d_timepoint_sec (2007, 9, 4, 6, 27, 17 ) ), hermes.sem_st_point
(4.634043, 51.24343, tau_tll.d_timepoint_sec (2007, 9, 5, 5, 44, 37 ) ) );
  outtag varchar2(50);
BEGIN
  sem_reconstruct.pois_probability (sembb, 4326, 'TRUE', 'belg_pois',
outtag);
  COMMIT;
END;
```



3. Count STOP and MOVES of semantic trajectories

Description:

This function counts the number of Stops and Moves for all semantic trajectories in the table.

Code:

```
SELECT o_id,
  semtraj_id,
  VALUE (t).num_of_stops () num_of_stops,
  VALUE (t).num_of_moves () num_of_moves,
  (VALUE (t).num_of_stops () + VALUE (t).num_of_moves () ) AS num_of_episodes
FROM belg_sem_trajs t
ORDER BY 4 DESC;
```

Query Result

SQL | Fetched 50 rows in 0,282 seconds

	O_ID	SEMTRAJ_ID	NUM_OF_STOPS	NUM_OF_MOVES	NUM_OF_EPISODES
1	127728	1	11	10	21
2	128116	1	10	9	19
3	121150	1	10	9	19
4	126836	1	9	8	17
5	15058	1	8	7	15
6	211191	1	8	7	15

4. Visualize a semantic trajectory

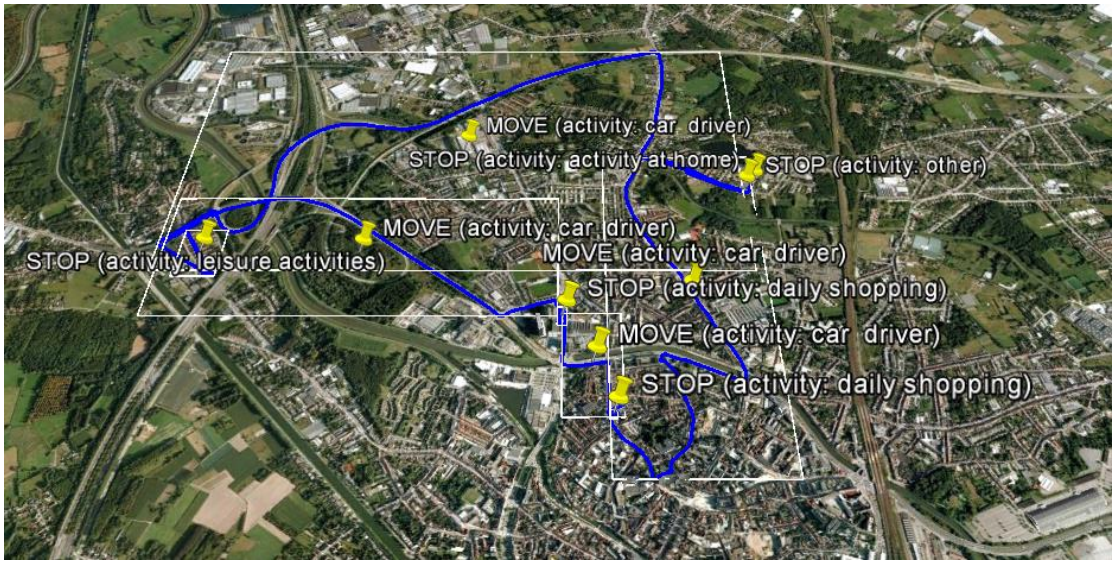
Description:

This operation visualizes a given semantic trajectory, i.e. the MBRs of its episodes and the corresponding raw sub-trajectories.

Code:

```
DECLARE
   semtraj       sem_trajectory;
   void          INTEGER         := 123043;
   vsemtrajid    INTEGER         := 1;
BEGIN
   SELECT VALUE (s)
     INTO semtraj
     FROM belg_sem_trajs s
    WHERE o_id = void AND semtraj_id = vsemtrajid;

   visualizer.semtrajectory2kml (semtraj, 'TRUE', 'TRUE', 'TRUE');
END;
```



5. Sum of MOVE durations per transportation mode

Description:

This query finds the summation of the durations of the MOVE episodes per transportation mode, for a given semantic trajectory.

Code:

```
SELECT    SUM (value(s).duration(). m_Value ) duration,
          defining_tag, activity_tag, COUNT (defining_tag) num_of_episodes
     FROM TABLE (SELECT t.episodes_with ('MOVE')
                   FROM belg_sem_trajs t
                  WHERE t.o_id = 5238 AND t.semtraj_id = 1) s
GROUP BY defining_tag, activity_tag;
```

	DURATION	DEFINING_TAG	ACTIVITY_TAG	NUM_OF_EPISODES
1	3860	MOVE	bus	2

6. MOVEs duration with that of some MOVEs having specific tags

Description:

Identify those MOVE episodes whose duration is less than the average duration of MOVE episodes whose activity is 'working'. This function demonstrates how one can filter episodes with multiple tags.

Code:

```
SELECT VALUE (ext_s).DURATION ().m_value DURATION,
  defining_tag,
  activity_tag
FROM TABLE
  (SELECT t.episodes_with ('MOVE')
  FROM belg_sem_trajs t
  WHERE t.o_id      = 5238
  AND t.semtraj_id = 2
  ) ext_s
WHERE (VALUE (ext_s).DURATION ().m_value) <
  (SELECT AVG (VALUE (s).DURATION ().m_value)
  FROM TABLE
    (SELECT t.episodes_with ('working')
    FROM belg_sem_trajs t
    WHERE t.o_id                    = 5238
    AND t.semtraj_id                = 2
    AND t.episodes_with ('MOVE') IS NOT NULL
    ) s
  );
```

	DURATION	DEFINING_TAG	ACTIVITY_TAG
1	1022	MOVE	subway/underground
2	1016	MOVE	by foot
3	2620	MOVE	bus
4	3211	MOVE	car - passenger

7. Confine a semantic trajectory in temporal dimension as well as by filtering its textual component

Description:

a. Restrict a given semantic trajectory inside a temporal period and then return only the STOP episodes that the user was working.

b. This query is a variant of the previous one that restricts a given semantic trajectory inside a temporal period and then returns either STOP or WORKING episodes.

Code:

a)
```
SELECT *
  FROM TABLE (
            SELECT b.confined_in  (NULL,
                  tau_tll.d_period_sec (
                  tau_tll.d_timepoint_sec (2007, 10, 22, 08, 00, 00 ),
                  tau_tll.d_timepoint_sec (2007, 10, 22, 23, 59, 00 )
                  ), 'STOP' ).episodes_with ('working')
            FROM belg_sem_trajs b
            WHERE b.o_id = 216828
            AND b.semtraj_id = 2
  );
```

Visualization

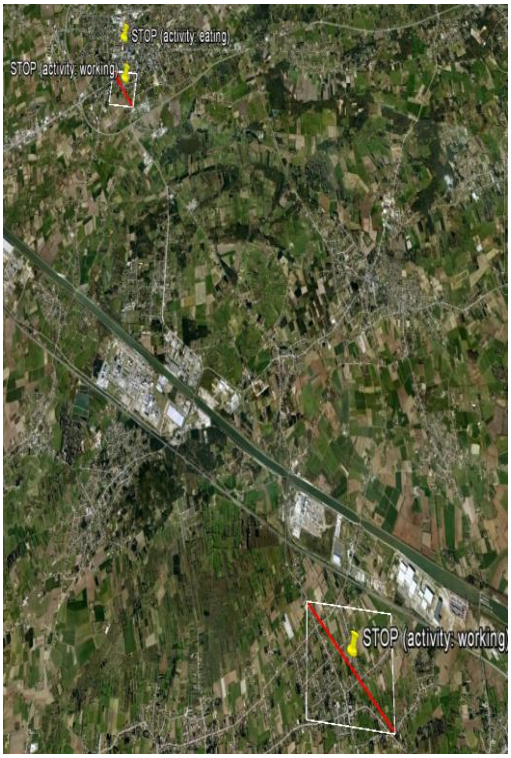
```
DECLARE
  vsrid      INTEGER := 4326;
  i          INTEGER := 0;
  void       INTEGER := 216828;
  vsemtrajid INTEGER := 2;
  sb_mps mp_array    := mp_array ();
  semtraj sem_trajectory;
BEGIN
  FOR rc IN
  (SELECT DEREF (tlink).sub_mpoint sub_mpoint
  FROM TABLE
    (SELECT b.confined_in (NULL, tau_tll.d_period_sec
(tau_tll.d_timepoint_sec (2007, 10, 22, 08, 00, 00 ), tau_tll.d_timepoint_sec
(2007, 10, 22, 23, 59, 00 ) ), 'STOP' ).episodes_with ('working')
    FROM belg_sem_trajs b
    WHERE b.o_id      = void
    AND b.semtraj_id = vsemtrajid
    ) s
  )
  LOOP
    sb_mps := mp_array (rc.sub_mpoint);
    visualizer.movingpointtable2kml (sb_mps, vsrid, 'u' || void || '_' || i
|| '_MOVPOINT.kml' );
    i := i + 1;
  END LOOP;
  SELECT VALUE (s)
  INTO semtraj
  FROM belg_sem_trajs s
  WHERE o_id      = void
  AND semtraj_id = vsemtrajid;
  visualizer.semtrajectory2kml (semtraj, 'TRUE', 'TRUE', 'TRUE');
END;
```

In the above figures there is the semantic trajectory as a whole in blue color and in red episodes returned by the query.

b)
```
SELECT *
  FROM TABLE (
SELECT b.confined_in  (NULL,
tau_tll.d_period_sec (
tau_tll.d_timepoint_sec (2007, 10, 22, 08, 00, 00 ), tau_tll.d_timepoint_sec
(2007, 10, 22, 23, 59, 00 )
), 'STOP+working' ).episodes
FROM belg_sem_trajs b
WHERE b.o_id = 216828
AND b.semtraj_id = 2
  );
```

Visualization

```
DECLARE
  vsrid      INTEGER := 4326;
  i          INTEGER := 0;
  void       INTEGER := 216828;
  vsemtrajid INTEGER := 2;
  sb_mps mp_array    := mp_array ();
  semtraj sem_trajectory;
BEGIN
  FOR rc IN
  (SELECT DEREF (tlink).sub_mpoint sub_mpoint
  FROM TABLE
    (SELECT b.confined_in  (NULL,
    tau_tll.d_period_sec (
    tau_tll.d_timepoint_sec (2007, 10, 22, 08, 00, 00 ),
tau_tll.d_timepoint_sec (2007, 10, 22, 23, 59, 00 )
    ), 'STOP+working' ).episodes
    FROM belg_sem_trajs b
    WHERE b.o_id     = void
    AND b.semtraj_id = vsemtrajid
    ) s
  )
  LOOP
    sb_mps := mp_array (rc.sub_mpoint);
    visualizer.movingpointtable2kml (sb_mps, vsrid, 'u' || void || '_' || i
|| '_MOVPOINT.kml' );
    i := i + 1;
  END LOOP;
  SELECT VALUE (s)
  INTO semtraj
  FROM belg_sem_trajs s
  WHERE o_id     = void
  AND semtraj_id = vsemtrajid;
  visualizer.semtrajectory2kml (semtraj, 'TRUE', 'TRUE', 'TRUE');
END;
```



In the above figures there is the semantic trajectory as a whole in blue color and in red episodes returned by the query.

8. Temporal range query with a text filter on STB-tree.

Description:

This query applies a temporal range query to retrieve and count only MOVE episodes, for which it calculates their total duration.

Code:

```
SELECT DEREF (tlink).o_id o_id,
  COUNT (tlink) total_moves,
  SUM (value(s).duration(). m_Value) total_duration
FROM TABLE
  (SELECT std.stb_range_episodes ('MOVE', tau_tll.d_period_sec
(tau_tll.d_timepoint_sec (2006, 01, 01, 00, 01, 00 ), tau_tll.d_timepoint_sec
(2006, 12, 31, 23, 59, 00 ) ), 'SEM_INDX' )
  FROM DUAL
  ) s
GROUP BY DEREF (tlink).o_id
ORDER BY 2 DESC;
```

SQL | All Rows Fetched: 45 in 0,78 seconds

	O_ID	TOTAL_MOVES	TOTAL_DURATION
1	122151	13	17020
2	124344	12	8847
3	127976	12	10233
4	17096	11	4264
5	8796	10	9339
6	123043	9	4744
7	127776	9	11159
8	128116	9	2181
9	122887	9	5620
10	135471	8	8812
11	126836	8	4198
12	15058	7	8480
13	5938	7	4896
14	5238	7	11780
15	4123	7	5023

9. Spatio-temporal range query with a text filter on STB-tree

Description:

This query applies a spatio-temporal range query to calculate the duration of 'Work/School' and 'working' STOP episodes that take place at a region for a period of time.

Code:

```
SELECT    SUM (VALUE (s).DURATION ().m_value) work_duration,
          DEREF (tlink).o_id o_id
     FROM TABLE (SELECT std.stb_range_episodes
                        ('STOP',
                         MDSYS.SDO_GEOMETRY (2003, 4236, NULL,
                              MDSYS.sdo_elem_info_array (1, 1003, 3),
                                        MDSYS.sdo_ordinate_array (4.985388,
51.152885, 5.994323, 52.158382) ),
                         tau_tll.d_period_sec (tau_tll.d_timepoint_sec (2007,
10, 22, 08, 00, 00),
                                               tau_tll.d_timepoint_sec (2008,
10, 22, 23, 00, 00) ), 'SEM_INDX' ) FROM DUAL ) s
    WHERE episode_tag = 'Work/School' AND activity_tag = 'working'
GROUP BY DEREF (tlink).o_id;
```

	WORK_DURATION	O_ID
1	4643	218411
2	36024	216828
3	16011	130049

10. Cross-over spatio-temporal range query with a text filter on STB-tree (filter step) and a subsequent temporal restriction of the resulting moving points (refinement step).

Description:

This query restricts the sub-trajectories of STOP episodes (that exist inside a spatio-temporal box), inside a temporal period.

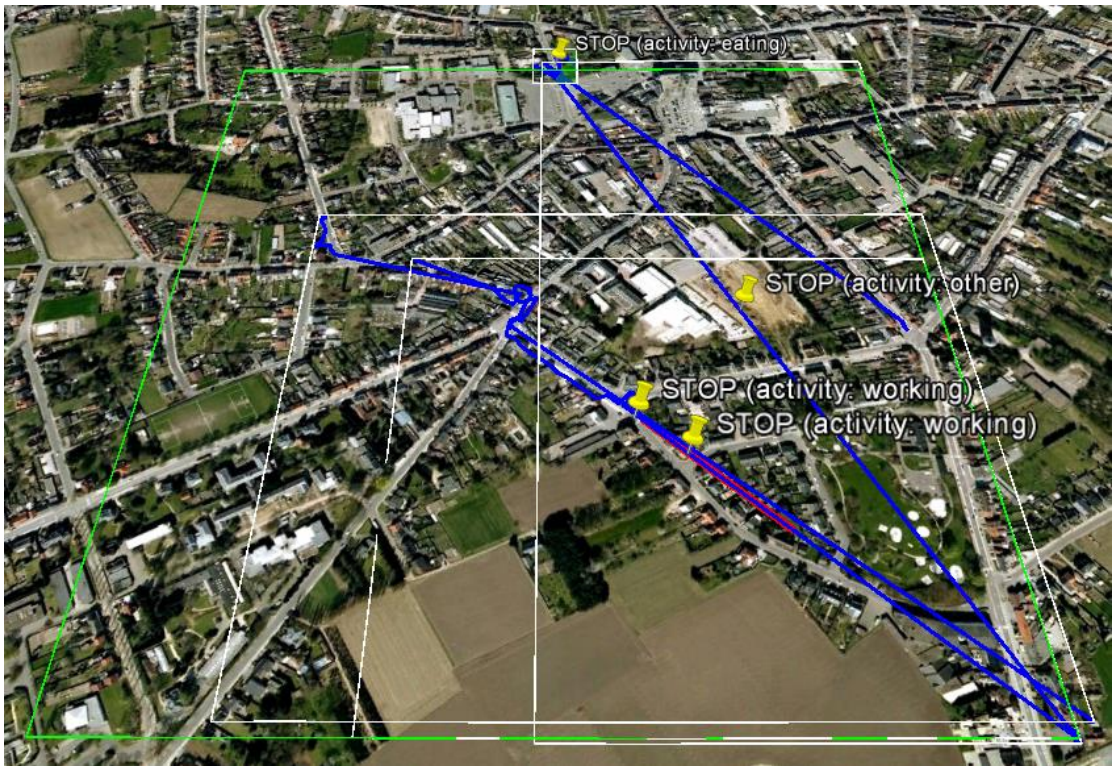
Code:

```
SELECT DEREF (tlink).sub_mpoint sub_mpoint,
  DEREF (tlink).sub_mpoint.at_period (tau_tll.d_period_sec
(tau_tll.d_timepoint_sec (2007, 10, 22, 8, 00, 00 ), tau_tll.d_timepoint_sec
(2007, 10, 22, 10, 00, 00 ) ) ) restricted_sub_mpoint,
  DEREF (tlink).o_id o_id
FROM TABLE
  (SELECT std.stb_range_episodes ('STOP', MDSYS.SDO_GEOMETRY (2003, 4326,
NULL, MDSYS.sdo_elem_info_array (1, 1003, 3 ), MDSYS.sdo_ordinate_array
(4.981388, 51.152885, 4.994323, 51.162082 ) ), tau_tll.d_period_sec
(tau_tll.d_timepoint_sec (2007, 10, 22, 5, 00, 00 ), tau_tll.d_timepoint_sec
(2007, 10, 28, 5, 00, 00 ) ), 'SEM_INDX' )
  FROM DUAL) s ;
```

Visualization

```
DECLARE
  vsrid INTEGER             := 4326;
  i     INTEGER             := 0;
  sb_mps mp_array           := mp_array ();
  rsb_mps mp_array          := mp_array ();
  vgeom MDSYS.SDO_GEOMETRY := MDSYS.SDO_GEOMETRY (2003, vsrid, NULL,
MDSYS.sdo_elem_info_array (1, 1003, 3), MDSYS.sdo_ordinate_array (4.981388,
51.152885, 4.994323, 51.162082 ) );
BEGIN
  FOR rc IN
  (SELECT DEREF (tlink).sub_mpoint sub_mpoint,
    DEREF (tlink).subtraj_id subtraj_id,
    DEREF (tlink).traj_id traj_id,
    DEREF (tlink).sub_mpoint.at_period (tau_tll.d_period_sec
(tau_tll.d_timepoint_sec (2007, 10, 22, 8, 00, 00 ), tau_tll.d_timepoint_sec
(2007, 10, 22, 10, 00, 00 ) ) ) restricted_sub_mpoint,
    DEREF (tlink).o_id o_id,
    MDSYS.sdo_geom.sdo_centroid (VALUE (s).mbb.getrectangle (vsrid), 0.01 )
mbr_centroid,
    VALUE (s).mbb.getrectangle (vsrid) mbb,
    activity_tag,
    episode_tag,
    defining_tag
  FROM TABLE
    (SELECT std.stb_range_episodes ('STOP', vgeom, tau_tll.d_period_sec
(tau_tll.d_timepoint_sec (2007, 10, 22, 5, 00, 00 ), tau_tll.d_timepoint_sec
(2007, 10, 28, 5, 00, 00 ) ), 'SEM_INDX' )
    FROM DUAL
    ) s
  )
  LOOP
    sb_mps                      := mp_array (rc.sub_mpoint);
    IF rc.restricted_sub_mpoint IS NOT NULL THEN
      rsb_mps                   := mp_array (rc.restricted_sub_mpoint);
```

```
      visualizer.movingpointtable2kml (rsb_mps, vsrid, 'u' || rc.o_id || '_'
|| i || '_RMOVPOINT.kml' );
    END IF;
    visualizer.movingpointtable2kml (sb_mps, vsrid, 'u' || rc.o_id || '_' ||
i || '_MOVPOINT.kml' );
    visualizer.placemark2kml (rc.mbr_centroid, vsrid, 'u' || rc.o_id ||
'traj' || rc.traj_id || 'subtraj' || rc.subtraj_id || '_CENTROID.kml',
rc.defining_tag || ' (activity: ' || rc.activity_tag || ')', rc.subtraj_id ||
' - ' || rc.episode_tag );
    visualizer.polygon2kml (rc.mbb, vsrid, 'u' || rc.o_id || 'traj' ||
rc.traj_id || 'subtraj' || rc.subtraj_id || '_RECTANGLE.kml' );
    i := i + 1;
  END LOOP;
  visualizer.polygon2kml (vgeom, vsrid, 'RECTANGLE.kml');
END;
```



In the above figure there are STOP episodes in blue color of the given time period and given spatial confinement (in green). Part of the episode that exists inside time period declared in at_period function is showed in red color.

11. Average duration of STOPS overlapping with a temporal period

Description:

This query finds the average duration of STOP episodes overlapping with a temporal period.

Code:

```
SELECT SUM (VALUE (s).DURATION ().m_value) / COUNT (DEREF (tlink).traj_id)
avg_mbb_stop_duration
FROM TABLE
  (SELECT std.stb_range_episodes ('STOP', MDSYS.SDO_GEOMETRY (2003, 4326,
NULL, MDSYS.sdo_elem_info_array (1, 1003, 3 ), MDSYS.sdo_ordinate_array
(4.985388, 51.152885, 4.994323, 51.158382 ) ), tau_tll.d_period_sec
```

```
(tau_tll.d_timepoint_sec (2007, 10, 22, 08, 00, 00 ), tau_tll.d_timepoint_sec
(2007, 10, 23, 08, 00, 00 ) ), 'SEM_INDX' )
  FROM DUAL
  ) s
```

	AVG_MBB_STOP_DURATION
1	95863,5

12. Index-based range query on STB-tree to identify patterns of the form "from-to-via"

Description:

Retrieve objects (actually their MOVE episodes) that start from 'Home' and go to 'Work/School' between a given temporal period

Code:

```
select deref(tlink).o_id mov_obj, activity_tag from
table(std.stb_range_episodes~~_mbr~~(
sem_episode('STOP',      'Home',null,null,null),
sem_episode('STOP',      'Work/School',null,null,null),
sem_episode('MOVE',null,null,
sem_mbb(sem_st_point(5.503502,50.953662,
tau_tll.d_timepoint_sec(2006,5,1,5,00,00)),sem_st_point(5.727093,51.026928,ta
u_tll.d_timepoint_sec(2007,6,1,5,00,00)))
,null),
'SEM_INDX')) t;
```

Query Result | SQL | All Rows Fetched: 6 in 0,5 seconds

	MOV_OBJ	ACTIVITY_TAG
1	137461	car β€" driver
2	137461	car β€" passenger
3	5238	bus
4	5238	subway/underground
5	123166	car β€" driver
6	123166	car β€" driver

How many and with what transportation mean start from 'Home' which are inside a given region, between a given temporal period and their next STOP is for 'working'.

Code:

```
SELECT COUNT(deref(tlink).o_id) sum_mov_obj,
  activity_tag
FROM TABLE(std.stb_range_episodes_mbr( sem_episode('STOP',
NULL,NULL,HERMES.SEM_MBB(HERMES.SEM_ST_POINT(4.938277,50.964135,TAU_TLL.D_TIM
EPOINT_SEC(2007,1,1,7,00,00)),HERMES.SEM_ST_POINT(5.503228,51.204038,TAU_TLL.
D_TIMEPOINT_SEC(2008,1,1,7,00,00))),NULL), sem_episode('STOP',
NULL,'working',NULL,NULL), NULL, 'SEM_INDX'))    t
GROUP BY activity_tag;
```

	SUM_MOV_OBJ	ACTIVITY_TAG
1	1	car β€" passenger
2	11	car β€" driver

The same query as the previous one, this time without imposing the constraint that the next STOP will be a 'working' one.

Code:

```
SELECT DEREF (tlink).o_id o_id,
  DEREF (tlink).traj_id traj_id,
  DEREF (tlink).sub_mpoint mpoint,
  DEREF (tlink).subtraj_id subtraj_id,
  value(t).mbb.getrectangle(4326),
  MDSYS.sdo_geom.sdo_centroid (VALUE (t).mbb.getrectangle (4326), 0.01 )
mbr_centroid,
  activity_tag,
  defining_tag
FROM TABLE (std.stb_range_episodes_mbr (sem_episode ('STOP', NULL, NULL,
hermes.sem_mbb (hermes.sem_st_point (5.162558,51.12519,
tau_tll.d_timepoint_sec (2007, 1, 1, 7, 00, 00 ) ), hermes.sem_st_point
(5.180134, 51.140135, tau_tll.d_timepoint_sec (2007, 1, 31, 7, 00, 00 ) ) ),
NULL ), sem_episode ('STOP', NULL, NULL, NULL, NULL ), NULL, 'SEM_INDX' ) ) t
```

Visualization

```
DECLARE
  stop_mbb hermes.sem_mbb := hermes.sem_mbb (hermes.sem_st_point (5.162558,
51.12519, tau_tll.d_timepoint_sec (2007, 1, 1, 7, 00, 00 ) ),
hermes.sem_st_point (5.180134, 51.140135, tau_tll.d_timepoint_sec (2007, 1,
31, 7, 00, 00 ) ) );
  res_mps mp_array        := mp_array ();
  semtraj sem_trajectory;
  vsrid INTEGER := 4326;
BEGIN
  FOR rc IN
  (SELECT DEREF (tlink).o_id o_id,
    DEREF (tlink).traj_id traj_id,
    DEREF (tlink).sub_mpoint mpoint,
    DEREF (tlink).subtraj_id subtraj_id,
    VALUE (t).mbb.getrectangle (vsrid) rect,
    MDSYS.sdo_geom.sdo_centroid (VALUE (t).mbb.getrectangle (vsrid), 0.01 )
mbr_centroid,
    activity_tag,
    defining_tag
  FROM TABLE (std.stb_range_episodes_mbr (sem_episode ('STOP', NULL, NULL,
stop_mbb, NULL ), sem_episode ('STOP', NULL, NULL, NULL, NULL ), NULL,
'SEM_INDX' ) ) t
  )
  LOOP
    res_mps := mp_array (rc.mpoint);
    visualizer.movingpointtable2kml (res_mps, vsrid, CONCAT ( 'u' || rc.o_id
|| 'traj' || rc.traj_id || 'subtraj' || rc.subtraj_id, '_Q_MOVPOINT.kml' ) );
    visualizer.polygon2kml (rc.rect, vsrid, rc.o_id || '.' || rc.traj_id ||
'.' || rc.subtraj_id || '_RECTANGLE.kml' );
    visualizer.placemark2kml (rc.mbr_centroid, vsrid, 'u' || rc.o_id ||
'traj' || rc.traj_id || 'subtraj' || rc.subtraj_id || '_CENTROID.kml', ' ',
rc.defining_tag || '-' || rc.activity_tag );
```

```
  END LOOP;
  visualizer.polygon2kml (stop_mbb.getrectangle (vsrid), vsrid,
'RECTANGLE.kml' );
END;
```

In the above figure in blue color are all MOVE episodes that have their starting Stop inside a spatiotemporal box and end to another STOP episode. The spatiotemporal box is depicted in green color.

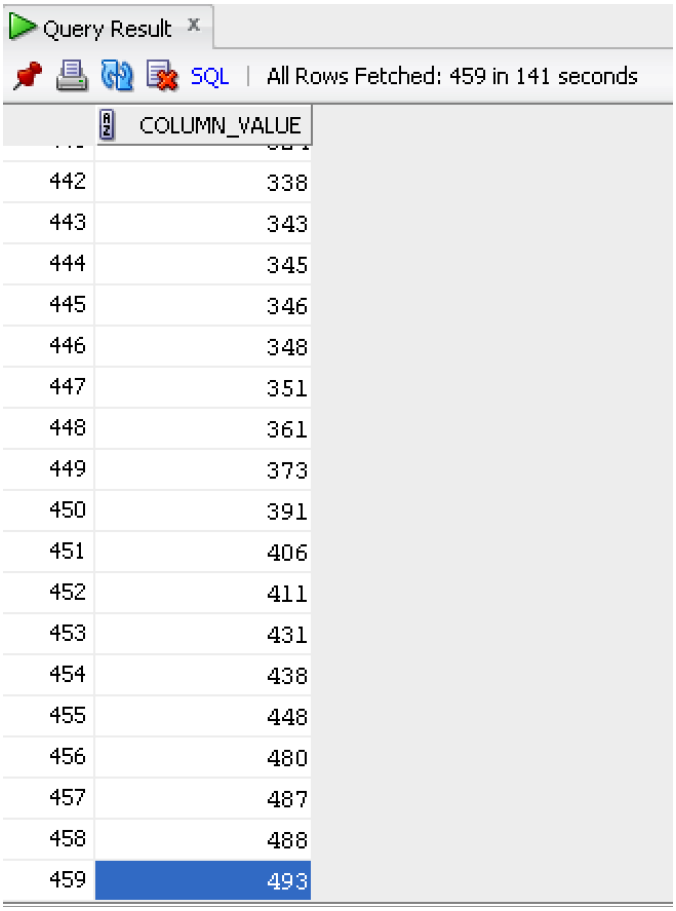
13. Index-based pattern query on STB-tree to identify patterns of movement

Description:

Retrieve the identifiers of semantic trajectories that follow the pattern of episodes given as input. The declared pattern includes objects that have an episode to a specific spatiotemporal area with any textual attributes, then after an arbitrary number of episodes reach to an episode defined as MOVE in any spatiotemporal area, where objects are using WALKING as TRANSPORTATION and in the immediate next episode objects having a STOP in any named place (null) where they are RELAXING in the specified spatiotemporal area.

Code:

```
select * from table(std.stb_patterns(sem_episode_tab(
sem_episode(null,null,null,sem_mbb(sem_st_point(468993,4201747,tau_tll.d_time
point_sec(2013,5,8,8,10,0)),sem_st_point(473993,4206747,tau_tll.d_timepoint_s
ec(2013,5,8,14,0,0))),null),
sem_episode('MOVE','TRANSPORTATION','WALKING',null,null),
sem_episode('STOP',null,'RELAXING',sem_mbb(sem_st_point(468993,4201747,tau_tl
l.d_timepoint_sec(2013,5,8,14,20,0)),sem_st_point(473993,4206747,tau_tll.d_ti
mepoint_sec(2013,5,8,23,0,0))),null)),
varchar_ntab(null,'*','>'),
'attiki_stbtree'));
```


Query Result

SQL | All Rows Fetched: 459 in 141 seconds

	COLUMN_VALUE
442	338
443	343
444	345
445	346
446	348
447	351
448	361
449	373
450	391
451	406
452	411
453	431
454	438
455	448
456	480
457	487
458	488
459	493

Analyzing the Semantic Trajectory Database/Cube

14. Origin-Destination Matrix for Semantic Trajectories

Description:

We first split the space (by partitioning the X and Y axis in segments of some given length (i.e.step)), so as to create the geometries (i.e. regions) of the OD-matrix. Then we calculate the OD-matrix, by providing the table where the semantic trajectories reside.

Usage:

```
DECLARE
   stepx    BINARY_INTEGER;
   stepy    BINARY_INTEGER;
BEGIN
   stepx := 0.5;
   stepy := 0.5;
   hermes.od_matrix.populate_rectangle_tbl (stepx, stepy);
   COMMIT;
END;

SELECT *
  FROM TABLE (hermes.od_matrix.get_odmatrix('BELG_SEM_TRAJS'))
 ;
```

SQL | Fetched 50 rows in 28,584 seconds

	STARTCELL	ENDCELL	TOTAL
1	859692	859692	95
2	859689	859689	70
3	859691	859691	46
4	859688	859688	25
5	859686	859686	12
6	859685	859685	11
7	859695	859695	5
8	859689	859688	2
9	859688	859689	2
10	859695	859691	1
11	859692	859691	1
12	859692	859688	1
13	859691	859694	1
14	859691	859692	1
15	859691	859689	1
16	859689	859686	1

15. Create and load DW

Description:

a) First we create the necessary DB objects by providing a prefix name for them. This prefix corresponds to the DW name.

b) We load data to dimensions tables by providing: i) the DW name, ii) the POIs table, the table contatining metadata of the dataset (min/max values of the spatio-temporal axes), iii) the table contacting users' information, iv) the duration of the temporal periods that will be used for the temporal dimension and v) the table containing the semantic trajectories.

c) We load the two fact tables by providing the table containing the semantic trajectories and the prefix name of the STB-tree index that has been built upon the latter.

Code:

a)
```
begin
sdw.createsdw('sem_dw');
end;
```

b)
```
begin
sdw.loaddimensions('sem_dw','belg_pois','belg_dataset_dimensions','belg_users
',2592000,'belg_sem_trajs');
end;
```

c)
```
begin
  sdw.cellstopsload('sem_dw','SEM_INDX');
end;

begin
  sdw.cellmovesload('sem_dw','SEM_INDX');
end;
```

16. Join MOVES fact table and USERS profile dimension table

Description:

Find the number of users that moved per temporal period and profession.

Code:

```
SELECT sem_dw_period_dim.period_id, sem_dw_user_profile_dim.profession,
COUNT(sem_dw_moves_fact.num_of_users) num_of_users
FROM sem_dw_moves_fact,  sem_dw_user_profile_dim,  sem_dw_period_dim
WHERE ( (sem_dw_user_profile_dim.user_profile_id =
sem_dw_moves_fact.user_profile_id )
AND (sem_dw_period_dim.period_id = sem_dw_moves_fact.period_id) )
GROUP BY sem_dw_user_profile_dim.profession, sem_dw_period_dim.period_id
ORDER BY sem_dw_period_dim.period_id ASC;
```

	PERIOD_ID	PROFESSION	NUM_OF_USERS
1	1	CLERK	4
2	1	HOUSEHOLD	1
3	1	STUDENT	1
4	2	RETIRED	1
5	4	CLERK	2
6	4	EXECUTIVE CLERK	1
7	5	CLERK	1
8	6	CLERK	4
9	6	RETIRED	1
10	6	(null)	1
11	7	CLERK	3
12	7	LABORER	1
13	8	SELF-EMPLOYED	2
14	8	STUDENT	2
15	10	RETIRED	1

17. Join MOVES fact table, USERS profile dimension table and MOVES semantic dimension table

Description:

Find the total average duration and travelled distance per gender and temporal period, for those that moved by train.

Code:

```
SELECT sem_dw_period_dim.period_id, sem_dw_user_profile_dim.gender,
          SUM (sem_dw_moves_fact.avg_travel_duration) sum_avg_dur,
          SUM (sem_dw_moves_fact.avg_distance_traveled) sum_avg_dist
     FROM sem_dw_moves_fact, sem_dw_user_profile_dim, sem_dw_move_sems_dim,
sem_dw_period_dim
    WHERE ((sem_dw_user_profile_dim.user_profile_id
=sem_dw_moves_fact.user_profile_id)
           AND (sem_dw_move_sems_dim.move_sems_id
=sem_dw_moves_fact.move_sems_id)
           AND (sem_dw_period_dim.period_id = sem_dw_moves_fact.period_id)
           AND ((sem_dw_move_sems_dim.move_mode = 'TRAIN'))
          ) GROUP BY sem_dw_user_profile_dim.gender,
sem_dw_period_dim.period_id ORDER BY sem_dw_period_dim.period_id ASC
```

	PERIOD_ID	GENDER	SUM_AVG_DUR	SUM_AVG_DIST
1	1	MAN	5033	121811,56450774486
2	1	WOMAN	1783	24741,87326075254
3	2	MAN	1472	21255,3924289534
4	4	MAN	3357	89455,2482850577
5	4	WOMAN	647	8254,98104342971
6	5	MAN	923	11103,0588393071
7	6	MAN	5894	78477,9676656374
8	6	WOMAN	255	3343,68573034657
9	7	MAN	2064	23325,38109972392
10	7	WOMAN	455	4433,28462431386
11	8	WOMAN	640	3894,02341379328
12	10	MAN	337	6326,86237521639
13	10	(null)	676	6541,72758176443

18. Aggregate type of MOVES

Description:

For each period, find the number of users that moved with the same transportation mode.

Code:

```
SELECT    sem_dw_moves_fact.period_id, sem_dw_move_sems_dim.move_mode,
count(num_of_users) total_users
    FROM sem_dw_moves_fact, sem_dw_move_sems_dim
   WHERE ((sem_dw_move_sems_dim.move_sems_id = sem_dw_moves_fact.move_sems_id
          )
         )
GROUP BY sem_dw_moves_fact.period_id, sem_dw_move_sems_dim.move_mode
ORDER BY sem_dw_moves_fact.period_id ASC;
```

PERIOD_ID	MOVE_MODE	TOTAL_USERS
1	CAR B€" PASSENGER	2
1	TRAIN	4
2	TRAIN	1
4	OTHER	1
4	TRAIN	2
5	TRAIN	1
6	OTHER	1
6	TRAIN	5
7	TRAIN	4
8	CAR B€" PASSENGER	2
8	TRAIN	2
10	TRAIN	3
11	TRAIN	3
13	OTHER	1
14	OTHER	2

19. Join MOVES, STOPS fact tables

Description:

Find the number of users that move towards a destination and the number of users that are already at the same destination during the same period of time.

Code:

```
WITH period_stopsto AS
  (SELECT sem_dw_stops_fact.period_id,COUNT (sem_dw_stops_fact.num_of_users)
num_of_stop_users,sem_dw_stop_sems_dim.stop_name
  FROM sem_dw_stops_fact,sem_dw_stop_sems_dim
  WHERE ((sem_dw_stop_sems_dim.stop_sems_id = sem_dw_stops_fact.stop_sems_id
) )
  GROUP BY sem_dw_stops_fact.period_id,sem_dw_stop_sems_dim.stop_name
  ),
  period_movesto AS
  (SELECT sem_dw_MOVES_fact.period_id,COUNT (sem_dw_MOVES_fact.num_of_users)
num_of_move_users,sem_dw_STOP_sems_dim.stop_name
  FROM sem_dw_MOVES_fact,sem_dw_STOP_sems_dim
  WHERE ((sem_dw_stop_sems_dim.stop_sems_id =
sem_dw_MOVES_fact.to_stop_sems_id) )
  GROUP BY sem_dw_MOVES_fact.period_id,sem_dw_STOP_sems_dim.stop_name
  )
SELECT
period_stopsto.period_id,period_stopsto.stop_name,period_stopsto.num_of_stop_
users,period_movesto.num_of_move_users
FROM period_stopsto FULL OUTER JOIN period_movesto
ON (period_stopsto.period_id=period_movesto.period_id AND
period_stopsto.stop_name=period_movesto.stop_name) ORDER BY 1 DESC,2;
```

PERIOD_ID	STOP_NAME	NUM_OF_STOP_USERS	NUM_OF_MOVE_USERS
25	philips	1	(null)
25	phillps	1	(null)
25	thuis	6	(null)
24	apotheek	1	(null)
24	carpet	1	(null)
24	cc	1	(null)
24	cevo	1	1
24	dochter	1	(null)
24	hengelen	1	(null)
24	ipb	1	(null)
24	kantoor	1	1
24	kliniek	1	(null)
24	moe poes	2	(null)
24	opstaan en eten	1	(null)
24	philips	1	(null)
24	school	1	1

20. Join STOPS fact table with semantic dimension and space dimension tables

Description:

For each time period, find the total average duration of all STOP episodes that take place at ‘banks’.

Code:

```
SELECT   SUM (sem_dw_stops_fact.avg_duration) sum_avg_duration ,
         sem_dw_stops_fact.period_id, sem_dw_stop_sems_dim.stop_name
    FROM sem_dw_stops_fact, sem_dw_space_dim, sem_dw_stop_sems_dim
   WHERE (     (sem_dw_space_dim.poi_id = sem_dw_stop_sems_dim.poi_id)
          AND (sem_dw_stop_sems_dim.stop_sems_id =

sem_dw_stops_fact.stop_sems_id
              )
          AND ((UPPER (sem_dw_stop_sems_dim.stop_name) LIKE UPPER ('%bank%'))
              )
         )
GROUP BY sem_dw_stops_fact.period_id, sem_dw_stop_sems_dim.stop_name
                ORDER BY sem_dw_stop_sems_dim.stop_name ASC;
```

SUM_AVG_DURATION	PERIOD_ID	STOP_NAME
601	10	axa bank
454	23	bank

21. STOPS at a given region

Description:

For each period, find the total average duration of those STOPs that are inside a given region.

Code:

```
DECLARE
  vgeom MDSYS.SDO_GEOMETRY := MDSYS.SDO_GEOMETRY (2003, 4326, NULL,
MDSYS.sdo_elem_info_array (1, 1003, 3), MDSYS.sdo_ordinate_array (4.815388,
51.112885, 4.994323, 53.158382 ) );
  vsrid INTEGER            := 4326;
BEGIN
  FOR rc_pois IN
  (SELECT poi_id
  FROM sem_dw_space_dim
  WHERE (sdo_geom.relate (MDSYS.SDO_GEOMETRY (2003, vsrid, NULL,
MDSYS.sdo_elem_info_array (1, 1003, 3), MDSYS.sdo_ordinate_array (4.815388,
51.112885, 4.994323, 53.158382 ) ), 'CONTAINS', sem_dw_space_dim.poi_geom,
0.01 ) = 'CONTAINS' )
  )
  LOOP
    FOR rc_dw IN
    (SELECT SUM (sem_dw_stops_fact.avg_duration) sum_duration,
      sem_dw_stops_fact.period_id,sem_dw_space_dim.district
    FROM sem_dw_stops_fact,sem_dw_space_dim,sem_dw_stop_sems_dim
    WHERE ( (sem_dw_space_dim.poi_id       = sem_dw_stop_sems_dim.poi_id )
    AND (sem_dw_stop_sems_dim.stop_sems_id = sem_dw_stops_fact.stop_sems_id )
    AND (sem_dw_space_dim.poi_id           = rc_pois.poi_id) )
    GROUP BY sem_dw_stops_fact.period_id,
             sem_dw_space_dim.district
    )
    LOOP
      DBMS_OUTPUT.put_line ( 'Duration: ' || rc_dw.sum_duration || '  Period
id: ' || rc_dw.period_id || '  District: ' || rc_dw.district );
    END LOOP;
  END LOOP;
END;
```

Duration: 85896 Period id: 17 District: Turnhout
Duration: 28106 Period id: 22 District: Turnhout
Duration: 85896 Period id: 17 District: Turnhout
Duration: 73271 Period id: 24 District: Turnhout
Duration: 227964 Period id: 20 District: Oud-Turnhout
Duration: 90290 Period id: 21 District: Beerse
Duration: 79027 Period id: 20 District: Oud-Turnhout
Duration: 32908 Period id: 13 District: N-Breda

References

Pelekis N., Theodoridis, Y. (2013) Semantic Aspects on Mobility Data (chapter 8). In Mobility Data Management and Exploration: Theory and Practice, Springer, 2013.

Pelekis N., Janssens D., Theodoridis, Y. (2013) On the Management and Analysis of our LifeSteps. Submitted to SIGMOD Record.

Zimmermann M., Kirste T.,Spiliopoulou M., (2009) Finding stops in error-prone trajectories of moving objects with time-based clustering. Communications in Computer and Information Science, Volume 53, pp 275-286.