pytempus Documentation

Release 1.2.3

Hugo Mercier

Sep 25, 2019

Contents:

1 Getting started **1**

1.1 Tempus main bindings 1

1.2 Isochrone plugin 3

1.3 Utilities 6

2 Indices and tables **7**

Python Module Index **9**

Index **11**

CHAPTER 1

Getting started

This project provides Python bindings for the Tempus framework and some example codes to illustrate the way it should be used.

1.1 Tempus main bindings

This page contains a definition of the existing wrappers that were defined for a range of seminal *Tempus* functions. These wrappers are stored in module *tempus.__init__*.

1.1.1 Load a plugin

`tempus.load_plugin(options, plugin_name=None, plugin_path=None)`

Load a Tempus plugin.

Parameters

- **options** – dict - Database options in a dictionary; the main entry is *db/options*
- **plugin_name** – object - String designing the name of the plugin that must be used (commonly supported: "sample_road_plugin", "sample_multi_plugin", "dynamic_multi_plugin", "sample_pt_plugin", "isochrone_plugin")
- **plugin_path** – object - Relative path to the plugin sources (alternative to *plugin_path* parameter, it is considered only if plugin_name is null)

Returns tempus.Plugin - routing plugin

1.1.2 Load a graph

`tempus.load_graph(options, graph_type='multimodal_graph')`

Load a Tempus graph

Parameters

- **options** – dict - Database options as a dictionary; the main entry is *db/options*
- **graph_type** – object - String designing the graph type to be used (supported: "multi-modal_graph", not yet supported: "ch_graph")

Returns tempus.<graph_type>.Graph - routing graph

1.1.3 Do routing requests

This class is defined to contain the routing request results. It is a wrapper that allow to consider the request itself associated with its result.

class `tempus.`**`ResultWrapper`**(*plugin_request*, *results*)

Tempus request result wrapper

Instanciated with a `Tempus.PluginRequest` (containing one or more `tempus.ResultElement`, *i.e.* `tempus.Roadmap` or `tempus.Isochrone`) and a dict containing the request resolution metrics (*i.e* number of iterations, execution time)

`__getitem__`(*key*)

Result item getter: consider results elements (of `tempus.ResultElement`)as a list, *i.e.* they must be accessed through an index *key*

Parameters **key** – integer - id of the result object that must be get

`__len__`()

built-in result element length; gives the number of stored items in the request result

A binding function *request* is defined by exploiting this class as follows:

`tempus.`**`request`**(*plugin*, *plugin_options=None*, *origin=None*, *steps=None*, *destination=None*, *allowed_transport_modes=None*, *criteria=None*, *parking_location=None*, *networks=None*)

Request the Tempus database according to the *plugin* capabilities, to get a shortest path structure between *origin* and *destination* nodes that uses only *allowed_transport_modes*

Parameters

- **plugin** – `Tempus.Plugin` - Plugin to use for answering the request (ex: isochrone_plugin)
- **plugin_options** – dict - Additional options to feed to the chosen plugin
- **origin** – integer - Id of the origin node
- **steps** – list - destination specifications (time constraint, parking at destination... for the final destination, or intermediary steps)
- **destination** – integer - Id of the destination node
- **allowed_transportation_modes** – list - Id of the allowed transportation modes; for having a mode glossary, please refer to your database (tempus.transport_mode)
- **criteria** – list - Optimization criteria (supported: `tempus.Cost.Duration`; not yet supported: `tempus.Cost.Distance`, `tempus.Cost.Calories`, `tempus.Cost.Carbon`, `tempus.Cost.Elevation`, `tempus.Cost.Landmark`, `tempus.Cost.NumberOfChanges`, `tempus.Cost.PathComplexity`, `tempus.Cost.Price`, `tempus.Cost.Security`, `tempus.Cost.Variability`)
- **parking_location** – integer - Parking node id **(NOT IMPLEMENTED)**
- **networks** – integer - Network id **(NOT IMPLEMENTED)**

Returns *ResultWrapper* - a routing request result and some running metrics

1.2 Isochrone plugin

One possibility to test Python version of Tempus is the isochrone' plugin. This page summarizes the code in *test_isochrone.py* module. It tests the Tempus isochrone plugin by running a simple isochrone request starting from a random node in the network.

Computing an isochrone needs to compute travel times to reach every other nodes from the starting node, and to compare these travel times to a fixed time limit: the isochrone is the set of nodes reachable in this amount of time

1.2.1 Module preparation

To run the testing module, some other modules must be loaded.

```
from datetime import datetime
import random

import psycopg2

import tempus
from tempus import Cost, Request

import utils
```

In addition to *tempus*, *datetime* is useful for setting up a time constraint with the request, *random* is called to print a roadmap from the origin node to a random destination node within the isochrone. *psycopg2* seems unavoidable, as a database connection is required to consider valid origin node and transportation mode. Finally, *utils* contains some useful functions in the testing scope, see *Utilities*.

1.2.2 Tempus initialization

As a mandatory step before computing isochrones, the tempus framework must be initialized.

```
tempus.init()
```

1.2.3 Connection to database

Then a connection to the database can be opened. First a database option variable is set as follows in *utils.py*:

```
g_db_options = os.getenv('TEMPUS_DB_OPTIONS', 'dbname=tempus_test_db')
```

This variable is used by *psycopg2* for database connection:

```
    conn = psycopg2.connect(utils.g_db_options)
    cursor = conn.cursor()
```

Note: At this point, it is crucial to remind that the database option declaration is a prerequisite to database connection. That supposes that an environment variable *TEMPUS_DB_OPTIONS* was declared before running the Python module. By default, this variable equals `dbname=tempus_test_db`, however its value may be changed by hand with a command similar to:

```
export TEMPUS_DB_OPTIONS="dbname=<dbname> port=<port> user=<user> password=<pwd>"
```

1.2.4 Plugin loading

As the goal here is to compute isochrones, the corresponding plugin is then loaded to be exploited in the following section.

```
plugin = tempus.load_plugin({'db/options': utils.g_db_options}, plugin_name="isochrone_plugin")
```

The database options are called again: the requests will be run in the same database than the one which received an *ad hoc* connection.

Note: The plugins currently have a second option, namely *db/schema*. It is not mentionned here as the default value is considered, *i.e. tempus*. As a consequence, the request will be solved with data stored in tables *tempus.road_node*, *tempus.road_section* and so on…

1.2.5 Routing request

After loading the plugin, then comes the routing query solving, which is fairly the main part of the isochrone computation.

```
# prepare the request
origin = utils.sample_node(cursor) # Consider a random node
constraint = \
Request.TimeConstraint(type=Request.TimeConstraintType.ConstraintAfter,
date_time=datetime(2016,10,21,6,43))
step = Request.Step(constraint=constraint)
iso_limit = 20.0
db_modes = utils.get_transport_modes(cursor)
print("Available transport modes in the database: {}".format(db_modes))
allowed_modes = [1, 3]

print(("Compute isochrone from node {} with a time threshold of {} minutes and following modes: {}"
       "").format(origin, iso_limit, [db_modes[k] for k in allowed_modes]))

# routing request
results = tempus.request(plugin = plugin,
                         origin = origin,
                         steps = [step],
                         plugin_options = { 'Isochrone/limit' : iso_limit },
                         criteria = [Cost.Duration],
                         allowed_transport_modes = allowed_modes)
```

The previous example brings into play a random node, a single constraint associated to the destination (the node must be reached after the 16/10/21 at 21:06:43, that kind of constraint is meaningful if public transport and/or time-dependent travel times are considered), a time threshold of 20 units to design the isochrone and two allowed modes, identified by a specific *id* (see *tempus.transport_mode* table to know the available modes).

Note: In this example, the isochrone is computed relatively to the duration criterion. As a consequence the threshold is expressed in minutes. Some other optimization criteria are thinkable, however they still are in development (see function documentation in *Tempus main bindings*).

Note: This example of request shows that the only used plugin options is *Isochrone/limit*. It has a default value of 10 units. However the isochrone plugin allows also to define:

- *Time/min_transfer_time*: a minimal transfer time (default value = 2 minutes),
- *Time/walking_speed* and *Time/cycling_speed*: constant walking and cycling speeds (default values of respectively 3.6 and 12km/h),
- *Time/car_parking_search_time*: a constant car parking search time (with default value of 5 minutes),
- *Time/use_speed_profiles*: a boolean (default as *false*) flag that indicates if speed profiles must be used,
- *Time/profile_name*: the speed profile name with option (which is an empty string by default),
- *Debug/verbose*: a debugging-purpose boolean that indicates if the processing must be verbose (*false* by default),
- *Multimodal/max_mode_changes*: the maximal number of mode changes (no constrained, by default)
- *Multimodal/max_pt_changes*: the maximal number of public transport changes (no constrained by default)

1.2.6 Result exploitation

Once the isochrone query has been solved, general results may be printed as follows:

```
        result_isochrone = results[0].isochrone()
        print("Resulting structure has size = {}".format(len(results)))
        print("Number of reachable nodes: {}".format(len(result_isochrone)))
        print("id, predecessor, x, y")
        print("\n".join(["{},{},{},{}".format(x.uid, x.predecessor, x.x, x.y) for x in␣
↪results[0].isochrone()]))
```

One may consequently evaluate the number of nodes that are contained in the isochrone structure, and get their characteristics: node id, predecessor id (in the isochrone searching space), x and y coordinates.

To go further, the roadmap from the origin node to each valid destination in the isochrone may be rebuilt. The following example shows how to proceed with a random chosen destination (the principle is easily reproducible and generalizable):

```
    vertices = {x.uid: x for x in result_isochrone}
    v = random.choice(range(len(vertices)))
    print("Path between node {} and node {}:".format(origin, v))
    cost_per_mode, total_wait_time = utils.browse(vertices, v)
    print("Waiting time: {:.1f} mins".format(total_wait_time))
    print("Total cost: {:.1f} mins".format(sum(cost_per_mode.values())))
    print("Accumulated costs per mode:")
    print("\n".join("{}: {:.1f} mins".format(k,v) for k,v in cost_per_mode.items()))
```

1.3 Utilities

This documentation page gather all functions contained into *samples/utils.py* module. If not working, please consider module docstrings.

Here are documented some useful functions for testing the pytempus framework.

`samples.utils.`**`road_node_id_from_coordinates`**(*cur*, *pt_xy*)
Return a node stored into the database starting from its coordinates *pt_xy*

Parameters

- **`cur`** – psycopg2 cursor - Database connection cursor
- **`pt_xt`** – tuple - point coordinates, as floating numbers

Returns integer - id of the node that corresponds to pt_xy‘

`samples.utils.`**`sample_node`**(*cur*)
Return a random node id from the tempus node table

Parameters **`cur`** – `psycopg2.cursor` - database connexion tool

Returns tuple - node id

`samples.utils.`**`get_transport_modes`**(*cur*)
Get the available transport modes in the current database

Parameters **`cur`** – `psycopg2.cursor` - database connexion tool

Returns dict - id and name for each available transport mode

`samples.utils.`**`browse`**(*vertices*, *v*)
Recursively get the itinerary steps between the path origin and *x*, a selected destination node, and print the intermediary node characteristics at each iteration

Parameters

- **`vertices`** – dict - `tempus.IsochroneValue` indexed by node ids
- **`v`** – integer - id of the node to browse

Returns tuple - cost per mode and total waiting time to reach the current destination

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

s

samples.utils, 6

Index

Symbols

__getitem__() (*tempus.ResultWrapper method*), 2
__len__() (*tempus.ResultWrapper method*), 2

B

browse() (*in module samples.utils*), 6

G

get_transport_modes() (*in module samples.utils*), 6

L

load_graph() (*in module tempus*), 1
load_plugin() (*in module tempus*), 1

R

request() (*in module tempus*), 2
ResultWrapper (*class in tempus*), 2
road_node_id_from_coordinates() (*in module samples.utils*), 6

S

sample_node() (*in module samples.utils*), 6
samples.utils (*module*), 6