QSRlib Documentation

Release 0.2.0

STRANDS

September 17, 2015

Contents

QSRlib is a library that allows computation of Qualitative Spatial Relations, as well as a development framework for rapid implementation of new QSRs.

Contents:

CHAPTER 1

Authors

- Yiannis Gatsoulis <y.gatsoulis@leeds.ac.uk>
- Christian Dondrup <cdondrup@lincoln.ac.uk>
- Peter Lightbody <pet1330@gmail.com>
- Paul Duckworth <scpd@leeds.ac.uk>

And all the STRANDS team.

CHAPTER 2

Installation

CHAPTER 3

Usage

3.1 Minimal Working Example

Compute QSRs with the MWE script *mwe.py* in *strands_qsr_lib/qsr_lib/scripts/*:

```
./mwe.py <qsr_name>
```

e.g.

```
./mwe.py rcc8
```

MWE source code:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from __future__ import print_function, division
from qsrlib.qsrlib import QSRlib, QSRlib_Request_Message
from qsrlib_io.world_trace import Object_State, World_Trace
import argparse

def pretty_print_world_qsr_trace(which_qsr, qsrlib_response_message):
    print(which_qsr, "request was made at ", str(qsrlib_response_message.timestamp_request_made)
          + " and received at " + str(qsrlib_response_message.timestamp_request_received)
          + " and computed at " + str(qsrlib_response_message.timestamp_qsrs_computed))
    print("---")
    print("Response is:")
    for t in qsrlib_response_message.qsrs.get_sorted_timestamps():
        foo = str(t) + ": "
        for k, v in zip(qsrlib_response_message.qsrs.trace[t].qsrs.keys(),
                        qsrlib_response_message.qsrs.trace[t].qsrs.values()):
            foo += str(k) + ":" + str(v.qsr) + "; "
        print(foo)

if __name__ == "__main__":
    # ********************************************************************************
    # create a QSRlib object if there isn't one already
    qsrlib = QSRlib()

    # ********************************************************************************
    # parse command line arguments
    options = sorted(qsrlib.qsrs_registry.keys())
    parser = argparse.ArgumentParser()
    parser.add_argument("qsr", help="choose qsr: %s" % options, type=str)
```

```
    args = parser.parse_args()
    if args.qsr in options:
        which_qsr = args.qsr
    else:
        raise ValueError("qsr not found, keywords: %s" % options)

    # ********************************************************************************************
    # make some input data
    world = World_Trace()
    o1 = [Object_State(name="o1", timestamp=0, x=1., y=1., width=5., length=8.),
          Object_State(name="o1", timestamp=1, x=1., y=2., width=5., length=8.),
          Object_State(name="o1", timestamp=2, x=1., y=3., width=5., length=8.)]

    o2 = [Object_State(name="o2", timestamp=0, x=11., y=1., width=5., length=8.),
          Object_State(name="o2", timestamp=1, x=11., y=2., width=5., length=8.),
          Object_State(name="o2", timestamp=2, x=11., y=3., width=5., length=8.)]
    world.add_object_state_series(o1)
    world.add_object_state_series(o2)

    # ********************************************************************************************
    # make a QSRlib request message
    qsrlib_request_message = QSRlib_Request_Message(which_qsr=which_qsr, input_data=world)
    # request your QSRs
    qsrlib_response_message = qsrlib.request_qsrs(request_message=qsrlib_request_message)

    # ********************************************************************************************
    # print out your QSRs
    pretty_print_world_qsr_trace(which_qsr, qsrlib_response_message)
```

3.2 Usage in more detail

Basically the above code consists of the following simple steps: * Create a *QSRlib* object * Convert your data in to QSRlib standard input format * Make a request to QSRlib * Parse the QSRlib response

With the first three being the necessary ones and the parsing step provided as an example to give you insight on the QSRlib response data structure.

3.2.1 Create a *QSRlib* object

```
qsrlib = QSRlib()
```

Note: This step can be omitted if you want to use QSRlib with ROS.

3.2.2 Convert your data in to QSRlib standard input format

```
world = World_Trace()
o1 = [Object_State(name="o1", timestamp=0, x=1., y=1., width=5., length=8.),
      Object_State(name="o1", timestamp=1, x=1., y=2., width=5., length=8.),
      Object_State(name="o1", timestamp=2, x=1., y=3., width=5., length=8.)]

o2 = [Object_State(name="o2", timestamp=0, x=11., y=1., width=5., length=8.),
      Object_State(name="o2", timestamp=1, x=11., y=2., width=5., length=8.),
      Object_State(name="o2", timestamp=2, x=11., y=3., width=5., length=8.)]
```

```
world.add_object_state_series(o1)
world.add_object_state_series(o2)
```

3.2.3 Make a request to QSRlib

```
# make a QSRlib request message
qsrlib_request_message = QSRlib_Request_Message(which_qsr=which_qsr, input_data=world)
# request your QSRs
qsrlib_response_message = qsrlib.request_qsrs(request_message=qsrlib_request_message)
```

Make a request to QSRlib using ROS

If you want to use ROS then you need to firstly run the QSRlib ROS server as follows:

```
rosrun qsr_lib qsrlib_ros_server.py
```

and the request is slightly different:

```
try:
    import rospy
    from qsrlib_ros.qsrlib_ros_client import QSRlib_ROS_Client
except ImportError:
    raise ImportError("ROS not found")
try:
    import cPickle as pickle
except:
    import pickle
client_node = rospy.init_node("qsr_lib_ros_client_example")
cln = QSRlib_ROS_Client()
qsrlib_request_message = QSRlib_Request_Message(which_qsr=which_qsr, input_data=world)
req = cln.make_ros_request_message(qsrlib_request_message)
res = cln.request_qsrs(req)
qsrlib_response_message = pickle.loads(res.data)
```

3.2.4 Parse the QSRlib response

```
def pretty_print_world_qsr_trace(which_qsr, qsrlib_response_message):
    print(which_qsr, "request was made at ", str(qsrlib_response_message.timestamp_request_made)
          + " and received at " + str(qsrlib_response_message.timestamp_request_received)
          + " and computed at " + str(qsrlib_response_message.timestamp_qsrs_computed))
    print("---")
    print("Response is:")
    for t in qsrlib_response_message.qsrs.get_sorted_timestamps():
        foo = str(t) + ": "
        for k, v in zip(qsrlib_response_message.qsrs.trace[t].qsrs.keys(),
                        qsrlib_response_message.qsrs.trace[t].qsrs.values()):
            foo += str(k) + ":" + str(v.qsr) + "; "
        print(foo)
```

CHAPTER 4

Supported QSRs

Currently, the following QSRs are supported:

ID	Name	3D	Reference
argd	Qualitative Argument Distances	?	?
argprobd	Qualitative Argument Probabilistic Distances	?	?
cardir	Cardinal Directions	NO	[Andrew1991]
mos	Moving or Stationary	?	?
qtcbcs	Qualitative Trajectory Calculus *bc* Simplified	?	?
qtcbs	Qualitative Trajectory Calculus *b* Simplified	?	?
qtccs	Qualitative Trajectory Calculus *c* Simplified	?	?
rcc2	Region Connection Calculus 2		?
rcc3	Region Connection Calculus 3		?
rcc8	Region Connection Calculus 8		?

4.1 References

[Andrew1991] Andrew, U. F. "Qualitative spatial reasoning about cardinal directions." Proc. of the 7th Austrian Conf. on Artificial Intelligence. Baltimore: Morgan Kaufmann. 1991.

CHAPTER 5

For QSR developers

In order to extend QSRlib with a new QSR the following two steps are needed: * Implement a new QSR * Register the new QSR with QSRlib

5.1 Howto: Implement a new QSR

Find below a minimally working example:

```
# -*- coding: utf-8 -*-
from __future__ import print_function, division
from qsrlib_qsrs.qsr_dyadic_abstractclass import QSR_Dyadic_1t_Abstractclass


class QSR_MWE(QSR_Dyadic_1t_Abstractclass):
    def __init__(self):
        super(QSR_MWE, self).__init__()
        self._unique_id = "mwe"
        self.all_possible_relations = ["left", "together", "right"]
        self._dtype = "points"

    def _compute_qsr(self, data1, data2, qsr_params, **kwargs):
        return {
            data1.x < data2.x: "left",
            data1.x > data2.x: "right"
        }.get(True, "together")
```

Line by line explanation.

```
class QSR_MWE(QSR_Dyadic_1t_Abstractclass):
```

Our class inherits from one of the special case abstract classes (more to this later).

```
def __init__(self):
    super(QSR_MWE, self).__init__()
    self._unique_id = "mwe"
    self.all_possible_relations = ["left", "together", "right"]
    self._dtype = "points"
```

In the constructor we need to define our QSR members. In this MWE the minimum parameters that need to be defined are: * *self._unique_id*: This is the name of the QSR. You can call it what you want but it must be unique among all QSRs and preferably as short as possible with camelCase names preferred. * *self.all_possible_relations*: A list (or tuple) of all the possible values that the QSR can take. It can be anything you like. * *self._dtype*: With what type of

data your *_compute_qsr* methods works with. Are they points or bounding boxes for example. For what you can use look in the parent class member *self._allowed_dtypes*.

```
def _compute_qsr(self, data1, data2, qsr_params, **kwargs):
    return {
        data1.x < data2.x: "left",
        data1.x > data2.x: "right"
    }.get(True, "together")
```

The only method you need to implement that computes the QSR you are implementing.

IMPORTANT NOTE: There are different types of parent classes that you can inherit from. You can see them in the *qsr_monadic_abstractclass.py* and *qsr_dyadic_abstractclass.py* files. If one of the "special case" classes like in this example the class *QSR_Dyadic_1t_Abstractclass* does not suit you then you can inherit from *QSR_Monadic_Abtractclass* or *QSR_Dyadic_Abstracclass*. In this case you will also have to provide your own *make_world_qsr_trace* (see the special cases for some example ideas) but you are not required to specify anymore the member *self._dtype* and also not required to implement a *_compute_qsr* method (unless you want to for better code practice in which we recommend that you use private scope, i.e. name it as *__compute_qsr*).

Lastly, if none of the monadic and dyadic family classes allow you to implement your QSR (e.g. you want a triadic QSR) then feel free to extend it in a similar manner or file an issue in github and we will try to implement the support infrastructure the quickest possible.

5.2 Howto: Register the new QSR with QSRlib

Add to *strands_qsr_lib/qsr_lib/src/qsrlib_qsrs/__init__.py* the following:

Import your class name in the imports (before the *qsrs_registry* line). E.g. for above QSR add the following line:

```
from qsr_new_mwe import QSR_MWE
```

Add the new QSR class name in *qsrs_registry*. E.g. for above QSR:

```
qsrs_registry = (<some other QSR class names>,
                 QSR_MWE)
```

CHAPTER 6

API

6.1 qsrlib package

6.1.1 Submodules

qsrlib.qsrlib module

QSRlib module.

class qsrlib.qsrlib.**QSRlib**(*help=False*)
Bases: object

The LIB

help()
stdout help message about QSRlib

qsrs_registry
Getter.

Returns self.__qsrs_registry

Return type dict

request_qsrs(*req_msg*)
Main function of the QSRlib that does all the magic; returns the computed requested QSRs.

Parameters **req_msg** (QSRlib_Request_Message) – A request message containing the necessary data and other options.

Returns Response message containing the computed requested QSRs and other information.

Return type *QSRlib_Response_Message*

class qsrlib.qsrlib.**QSRlib_Request_Message**(*which_qsr*, *input_data*, *dynamic_args={}*, *req_made_at=None*)
Bases: object

Input to QSRlib request calls containing all the necessary data.

dynamic_args = None
dict: User args passed dynamically during the request.

input_data = None
World_Trace: The input data.

made_at = None
datetime.datetime (*datetime.datetime.now()*): Time the request was made.

which_qsr = None
str or list: The name(s) of the wanted QSR(s) to be computed.

class qsrlib.qsrlib.**QSRlib_Response_Message**(*qsrs*, *req_made_at*, *req_received_at*, *req_finished_at*)
Bases: object

The response message of QSRlib containing the QSRs and time processing information.

qsrs = None
World_QSR_Trace: Holds the QSRs.

req_finished_at = None
datetime.datetime : Time the QSRlib finished processing the request.

req_made_at = None
datetime.datetime : Time the request was made.

req_received_at = None
datetime.datetime : Time the request was received in QSRlib.

6.1.2 Module contents

6.2 qsrlib_io package

6.2.1 Submodules

qsrlib_io.world_trace module

class qsrlib_io.world_trace.**Object_State**(*name*, *timestamp*, *x=nan*, *y=nan*, *z=nan*, *xsize=nan*, *ysize=nan*, *zsize=nan*, *rotation=()*, **args*, ***kwargs*)
Bases: object

Data class structure that is holding various information about an object.

name = None
str: The name of the object

return_bounding_box_2d(*xsize_minimal=0*, *ysize_minimal=0*)
Compute the 2D bounding box of the object.

Parameters

- **xsize_minimal** (*int or float*) – If object has no x-size (i.e. simply a point) then compute bounding box based on this minimal x-size.
- **ysize_minimal** – If object has no y-size (i.e. simply a point) then compute bounding box based on this minimal y-size.

Returns The coordinates of the upper-left and bottom-right corners of the bounding box.

Return type list

rotation
tuple or list of floats: Rotation of the object in roll-pitch-yaw form or quaternion (x,y,z,w) one.

timestamp = None
float: The timestamp of the object state, which matches the corresponding key *t* in *World_Trace.trace[t]*.

x = None
int or float: x-coordinate of the center point.

xsize
positive int or float: Total x-size

y = None
int or float: y-coordinate of the center point.

ysize
positive int or float: Total y-size

z = None
int or float: z-coordinate of the center point.

zsize
positive int or float: Total z-size

class qsrlib_io.world_trace.**World_State**(*timestamp*, *objects=None*)
Bases: object

Data class structure that is holding various information about the world at a particular time.

add_object_state(*object_state*)
Add/Overwrite an object state.

Parameters **object_state** (Object_State) – Object state to be added in the world state.

Returns

objects = None
dict: Holds the state of the objects that exist in this world state, i.e. a dict of objects of type Object_State with the keys being the objects names.

timestamp = None
float: The timestamp of the object, which matches the corresponding key *t* in *World_Trace.trace[t]*.

class qsrlib_io.world_trace.**World_Trace**(*description=''*, *trace=None*)
Bases: object

Data class structure that is holding a time series of the world states.

add_object_state(*object_state*, *timestamp=None*)
Add/Overwrite an Object_State object.

Parameters

- **object_state** (Object_State) – The object state.
- **timestamp** (*int or float*) – The timestamp where the object state is to be inserted, if not given it is added in the timestamp of the object state.

Returns

add_object_state_series(*object_states*)
Add a series of object states.

Parameters **object_states** (*list or tuple*) – The object states, i.e. a list of Object_State objects.

Returns

add_object_track_from_list(*obj_name*, *track*, *t0=0*, ***kwargs*)
Add the objects data to the world_trace from a list of values

Parameters

- **obj_name** (*str*) – name of object
- **track** (*list*) – List of values as [[x1, y1, w1, l1], [x2, y2, w2, l2], ...] or [[x1, y1], [x2, y2], ...].
- **t0** (*int or float*) – First timestamp to offset timestamps.
- **kwargs** – Optional arguments.

Returns

description = None
str: Optional description of the world.

get_at_timestamp_range(*start*, *finish=None*, *copy_by_reference=False*, *include_finish=True*)
Return a subsample between start and finish timestamps.

Parameters

- **start** (*int or float*) – The start timestamp.
- **finish** – The finish timestamp. If empty then finish is set to the last timestamp.
- **copy_by_reference** (*bool*) – Return by value or by reference.
- **include_finish** (*bool*) – Whether to include or not the world state at the finish timestamp.

Returns A subsample between start and finish.

Return type *World_Trace*

get_for_objects(*objects_names*, *copy_by_reference=False*)
Return a subsample for requested objects.

Parameters

- **objects_names** (*list or tuple*) – The requested objects names.
- **copy_by_reference** (*bool*) – Return by value or by reference.

Returns A subsample for the requested objects.

Return type *World_Trace*

get_for_objects_at_timestamp_range(*start*, *finish*, *objects_names*, *copy_by_reference=False*, *include_finish=True*, *time_slicing_first=True*)
Return a subsample for requested objects between start and finish timestamps.

Parameters

- **start** (*int or float*) – The start timestamp.
- **finish** (*bool*) – The finish timestamp.
- **objects_names** – The requested objects names.
- **copy_by_reference** (*bool*) – Return by value or by reference.
- **include_finish** (*bool*) – Whether to include or not the world state at the finish timestamp.
- **time_slicing_first** (*bool*) – Perform time slicing first or object slicing, can be used to optimize the call.

Returns A subsample for the requested objects between start and finish timestamps.

Return type *World_Trace*

get_last_state(*copy_by_reference=False*)
Get the last world state.

Parameters **copy_by_reference** (*bool*) – Return by value or by reference.

Returns

get_sorted_timestamps()
Return a sorted list of the timestamps.

Returns A sorted list of the timestamps.

Return type list

trace = None
dict: A time series of world states, i.e. a dict of objects of type World_State with the keys being the timestamps.

qsrlib_io.world_qsr_trace module

class qsrlib_io.world_qsr_trace.**QSR**(*timestamp*, *between*, *qsr*, *qsr_type=''*)
Bases: object

Data class structure that is holding the QSR and other related information.

between = None
str: For which object(s) is the QSR for. For multiple objects names are separated by commas, e.g. "o1,o2".

qsr = None
dict: The QSR value(s). It is a dictionary where the keys are the unique names of each QSR and the values are the QSR values as strings.

timestamp = None
float: The timestamp of the QSR, which usually matches the corresponding key *t* in *World_QSR_Trace.trace[t]*.

type = None
str: The name of the QSR. For multiple QSRs it is usually a sorted comma separated string.

class qsrlib_io.world_qsr_trace.**World_QSR_State**(*timestamp*, *qsrs=None*)
Bases: object

Data class structure that is holding various information about the QSR world at a particular time.

add_qsr(*qsr*)
Add/Overwrite a QSR object to the state.

Parameters **qsr** (QSR) – QSR to be added in the world QSR state.

Returns

qsrs = None
dict: Holds the QSRs that exist in this world QSR state, i.e. a dict of objects of type QSR with the keys being the object(s) names that these QSR are for.

timestamp = None
float: The timestamp of the state, which matches the corresponding key *t* in *World_QSR_Trace.trace[t]*.

class qsrlib_io.world_qsr_trace.**World_QSR_Trace**(*qsr_type*, *trace=None*)
Bases: object

Data class structure that is holding a time series of the world QSR states.

add_qsr(*qsr*, *timestamp*)
Add/Overwrite a QSR at timestamp.

Parameters

- **qsr** (QSR) – The QSR object to be added.
- **timestamp** (*float*) – The timestamp of the QSR.

Returns

add_world_qsr_state(*world_qsr_state*)
Add/Overwrite a world QSR state.

Parameters **world_qsr_state** (World_QSR_State) – The world QSR state to be added.

Returns

get_at_timestamp_range(*start*, *finish=None*, *copy_by_reference=False*, *include_finish=True*)
Return a subsample between start and finish timestamps.

Parameters

- **start** (*int or float*) – The start timestamp.
- **finish** – The finish timestamp. If empty then finish is set to the last timestamp.
- **copy_by_reference** (*bool*) – Return by value or by reference.
- **include_finish** (*bool*) – Whether to include or not the world state at the finish timestamp.

Returns A subsample between start and finish.

Return type *World_QSR_Trace*

get_for_objects(*objects_names*, *copy_by_reference=False*)
Return a subsample for requested objects.

Parameters

- **objects_names** (*list or tuple*) – The requested objects names.
- **copy_by_reference** (*bool*) – Return by value or by reference.

Returns A subsample for the requested objects.

Return type *World_QSR_Trace*

get_for_objects_at_timestamp_range(*start*, *finish*, *objects_names*, *copy_by_reference=False*, *include_finish=True*, *time_slicing_first=True*)
Return a subsample for requested objects between start and finish timestamps.

Parameters

- **start** (*int or float*) – The start timestamp.
- **finish** (*bool*) – The finish timestamp.
- **objects_names** – The requested objects names.
- **copy_by_reference** (*bool*) – Return by value or by reference.
- **include_finish** (*bool*) – Whether to include or not the world state at the finish timestamp.
- **time_slicing_first** (*bool*) – Perform time slicing first or object slicing, can be used to optimize the call.

Returns A subsample for the requested objects between start and finish timestamps.

Return type *World_QSR_Trace*

get_for_qsrs(*qsrs_list*)
Return a subsample for requested QSRs only.

Parameters **qsrs_list** (*list*) – List of requested QSRs.

Returns A subsample for the requested QSRs.

Return type *World_QSR_Trace*

get_last_state(*copy_by_reference=False*)
Get the last world QSR state.

Parameters **copy_by_reference** (*bool*) – Return by value or by reference.

Returns

get_sorted_timestamps()
Return a sorted list of the timestamps.

Returns A sorted list of the timestamps.

Return type list

put_empty_world_qsr_state(*timestamp*)
Put an empty World_QSR_State object at timestamp.

Parameters **timestamp** (*float*) – Timestamp of where to add an empty World_QSR_State

Returns

qsr_type = None
str: The name of the QSR. For multiple QSRs it is usually a sorted comma separated string.

trace = None
dict: A time series of world QSR states, i.e. a dict of objects of type World_QSR_State with the keys being the timestamps.

6.2.2 Module contents

6.3 qsrlib_qsrs package

6.3.1 Submodules

qsrlib_qsrs.qsr_abstractclass module

class qsrlib_qsrs.qsr_abstractclass.**QSR_Abstractclass**
Bases: object

Root abstract class of the QSR implementators.

Abstract properties

- **_unique_id** (*str*): Unique identifier of a QSR.
- **_all_possible_relations** (*tuple*): All possible relations of a QSR.
- **_dtype** (*str*): Kind of data the QSR operates with, see self._dtype_map for possible values.

Members

- **_dtype_map** (*dict*): Mapping of _dtype to methods. It is equal to:

```
`python {"points":  self._return_points, "bounding_boxes":
self._return_bounding_boxes_2d, "bounding_boxes_2d":
self._return_bounding_boxes_2d} `
```

all_possible_relations
Getter for self._all_possible_relations.

Returns All the possible relations of the QSR.

Return type tuple

get_qsrs(***req_params*)
Compute the QSRs.

This method is called from QSRlib so no need to call it directly from anywhere else.

Parameters **req_params** (*dict*) – The request parameters.

Returns Computed world qsr trace.

Return type *qsrlib_io.world_qsr_trace.World_QSR_Trace*

make_world_qsr_trace(*world_trace*, *timestamps*, *qsr_params*, *req_params*, ***kwargs*)
The main function that generates the world QSR trace.

•QSR classes inheriting from the general purpose meta-abstract classes (e.g. **'QSR_Monadic_Abstractclass'_**,

[QSR_Dyadic_Abstractclass](qsrlib_qsrs.qsr_dyadic_abstractclass.QSR_Dyadic_Abstractclass), etc.) need to provide this function. * When inheriting from one of the special case meta-abstract classes (e.g. QSR_Monadic_2t_Abstractclass <qsr-lib_qsrs.qsr_monadic_abstractclass.QSR_Monadic_2t_Abstractclass>, QSR_Dyadic_1t_Abstractclass qsrlib_qsrs.qsr_dyadic_abstractclass.QSR_Dyadic_1t_Abstractclass, etc.) then usually there is no need to do so; check with the documentation of these special cases to see if they already implement one.

Parameters

- **world_trace** (qsrlib_io.world_trace.World_Trace) – The input data.
- **timestamps** (*list*) – List of sorted timestamps of *world_trace*.
- **qsr_params** (*dict*) – QSR specific parameters passed in *dynamic_args*.
- **dynamic_args** (*dict*) – The dynamic arguments passed with the request.
- **kwargs** – Optional further arguments.

Returns The computed world QSR trace.

Return type *qsrlib_io.world_qsr_trace.World_QSR_Trace*

unique_id
Getter for self._unique_id.

Returns The unique identifier of the QSR.

Return type str

qsrlib_qsrs.qsr_arg_prob_relations_distance module

Class for probabilistic distance QSR

Author Christian Dondrup <cdondrup@lincoln.ac.uk>

Organization University of Lincoln

class qsrlib_qsrs.qsr_arg_prob_relations_distance.**QSR_Arg_Prob_Relations_Distance**
Bases: *qsrlib_qsrs.qsr_arg_relations_distance.QSR_Arg_Relations_Distance*

qsrlib_qsrs.qsr_arg_relations_abstractclass module

class qsrlib_qsrs.qsr_arg_relations_abstractclass.**QSR_Arg_Relations_Abstractclass**
Bases: *qsrlib_qsrs.qsr_dyadic_abstractclass.QSR_Dyadic_1t_Abstractclass*

qsrlib_qsrs.qsr_arg_relations_distance module

class qsrlib_qsrs.qsr_arg_relations_distance.**QSR_Arg_Relations_Distance**
Bases: *qsrlib_qsrs.qsr_arg_relations_abstractclass.QSR_Arg_Relations_Abstractclass*

qsrlib_qsrs.qsr_dyadic_abstractclass module

class qsrlib_qsrs.qsr_dyadic_abstractclass.**QSR_Dyadic_1t_Abstractclass**
Bases: *qsrlib_qsrs.qsr_dyadic_abstractclass.QSR_Dyadic_Abstractclass*

Special case abstract class of dyadic QSRs.

Works with dyadic QSRs that require data over one timestamp.

make_world_qsr_trace(*world_trace*, *timestamps*, *qsr_params*, *req_params*, ***kwargs*)
Compute the world QSR trace from the arguments.

Parameters

- **world_trace** (qsrlib_io.world_trace.World_Trace) – The input data.
- **timestamps** (*list*) – List of sorted timestamps of *world_trace*.
- **qsr_params** (*dict*) – QSR specific parameters passed in *dynamic_args*.
- **dynamic_args** (*dict*) – The dynamic arguments passed with the request.
- **kwargs** – Optional further arguments.

Returns The computed world QSR trace.

Return type *qsrlib_io.world_qsr_trace.World_QSR_Trace*

class qsrlib_qsrs.qsr_dyadic_abstractclass.**QSR_Dyadic_Abstractclass**
Bases: *qsrlib_qsrs.qsr_abstractclass.QSR_Abstractclass*

Abstract class of dyadic QSRs, i.e. QSRs that are computed over two objects.

qsrlib_qsrs.qsr_monadic_abstractclass module

class qsrlib_qsrs.qsr_monadic_abstractclass.**QSR_Monadic_2t_Abstractclass**
Bases: *qsrlib_qsrs.qsr_monadic_abstractclass.QSR_Monadic_Abstractclass*

Special case abstract class of monadic QSRs.

Works with monadic QSRs that require data over two timestamps.

make_world_qsr_trace(*world_trace*, *timestamps*, *qsr_params*, *req_params*, ***kwargs*)
Compute the world QSR trace from the arguments.

Parameters

- **world_trace** (qsrlib_io.world_trace.World_Trace) – The input data.
- **timestamps** (*list*) – List of sorted timestamps of *world_trace*.
- **qsr_params** (*dict*) – QSR specific parameters passed in *dynamic_args*.
- **dynamic_args** (*dict*) – The dynamic arguments passed with the request.
- **kwargs** – Optional further arguments.

Returns The computed world QSR trace.

Return type *qsrlib_io.world_qsr_trace.World_QSR_Trace*

class qsrlib_qsrs.qsr_monadic_abstractclass.**QSR_Monadic_Abstractclass**
Bases: *qsrlib_qsrs.qsr_abstractclass.QSR_Abstractclass*

Abstract class of monadic QSRs, i.e. QSRs that are computed over a single object.

qsrlib_qsrs.qsr_moving_or_stationary module

class qsrlib_qsrs.qsr_moving_or_stationary.**QSR_Moving_or_Stationary**
Bases: *qsrlib_qsrs.qsr_monadic_abstractclass.QSR_Monadic_2t_Abstractclass*

Computes moving or stationary relations: 'm': moving, 's': stationary

qsrlib_qsrs.qsr_new_mwe module

class qsrlib_qsrs.qsr_new_mwe.**QSR_MWE**
Bases: *qsrlib_qsrs.qsr_dyadic_abstractclass.QSR_Dyadic_1t_Abstractclass*

Minimal Working Example (MWE) of making a new QSR.

Members:

- _unique_id: "mwe"
- _all_possible_relations: ("left", "together", "right")
- _dtype: "points"

Some explanation about the QSR. Maybe a reference if it exists.

qsrlib_qsrs.qsr_qtc_b_simplified module

Example that shows how to implement QSR makers.

Author Christan Dondrup <cdondrup@lincoln.ac.uk>, Yiannis Gatsoulis <y.gatsoulis@leeds.ac.uk>

Organization University of Lincoln

Date 10 September 2014

Version 0.1

Status Development

Copyright STRANDS default

Notes future extension to handle polygons, to do that use matplotlib.path.Path.contains_points although might want to have a read on the following also... http://matplotlib.1069221.n5.nabble.com/How-to-properly-use-path-Path-contains-point-td40718.html

class qsrlib_qsrs.qsr_qtc_b_simplified.**QSR_QTC_B_Simplified**
Bases: *qsrlib_qsrs.qsr_qtc_simplified_abstractclass.QSR_QTC_Simplified_Abstractclass*

Make default QSRs and provide an example for others

qtc_to_output_format(*qtc*)
Overwrite this for the different QTC veriants to select only the parts from the QTCC tuple that you would like to return. Example for QTCB: return qtc[0:2]

Parameters **qtc** – The full QTCC tuple [q1,q2,q4,q5]

Returns "q1,q2" of {"qtcbs": "q1,q2"} if future is True

qsrlib_qsrs.qsr_qtc_bc_simplified module

Example that shows how to implement QSR makers.

Author Christan Dondrup <cdondrup@lincoln.ac.uk>

Organization University of Lincoln

Date 10 September 2014

Version 0.1

Status Development

Copyright STRANDS default

Notes future extension to handle polygons, to do that use matplotlib.path.Path.contains_points although might want to have a read on the following also... http://matplotlib.1069221.n5.nabble.com/How-to-properly-use-path-Path-contains-point-td40718.html

class qsrlib_qsrs.qsr_qtc_bc_simplified.**QSR_QTC_BC_Simplified**
Bases: *qsrlib_qsrs.qsr_qtc_simplified_abstractclass.QSR_QTC_Simplified_Abstractclass*

Make default QSRs and provide an example for others

make_world_qsr_trace(*world_trace*, *timestamps*, *qsr_params*, *req_params*, ***kwargs*)

qtc_to_output_format(*qtc*)
Overwrite this for the different QTC veriants to select only the parts from the QTCC tuple that you would like to return. Example for QTCB: return qtc[0:2]

Parameters **qtc** – The full QTCC tuple [q1,q2,q4,q5]

Returns "q1,q2,q4,q5" or {"qtcbcs": "q1,q2,q4,q5"} if future is True

qsrlib_qsrs.qsr_qtc_c_simplified module

Example that shows how to implement QSR makers.

Author Christan Dondrup <cdondrup@lincoln.ac.uk>

Organization University of Lincoln

Date 10 September 2014

Version 0.1

Status Development

Copyright STRANDS default

Notes future extension to handle polygons, to do that use matplotlib.path.Path.contains_points although might want to have a read on the following also... http://matplotlib.1069221.n5.nabble.com/How-to-properly-use-path-Path-contains-point-td40718.html

class qsrlib_qsrs.qsr_qtc_c_simplified.**QSR_QTC_C_Simplified**
Bases: *qsrlib_qsrs.qsr_qtc_simplified_abstractclass.QSR_QTC_Simplified_Abstractclass*

Make default QSRs and provide an example for others

qtc_to_output_format(*qtc*)
Overwrite this for the different QTC veriants to select only the parts from the QTCC tuple that you would like to return. Example for QTCB: return qtc[0:2]

Parameters **qtc** – The full QTCC tuple [q1,q2,q4,q5]

Returns "q1,q2,q4,q5" or {"qtccs": "q1,q2,q4,q5"} if future is True

qsrlib_qsrs.qsr_qtc_simplified_abstractclass module

Created on Mon Jan 19 11:22:16 2015

@author: cdondrup

class qsrlib_qsrs.qsr_qtc_simplified_abstractclass.**QSR_QTC_Simplified_Abstractclass**
Bases: *qsrlib_qsrs.qsr_dyadic_abstractclass.QSR_Dyadic_Abstractclass*

print "where,

" "it is always necessary to have two agents in every timestep: " "x, y: the xy-coords of the agents " "quantisation_factor: the minimum distance the agents must diverge from the double cross between two timesteps to be counted as movement. Must be in the same unit as the x,y coordinates. " "validate: TruelFalse validates the QTC sequence to not have illegal transitions. This inserts necessary transitional steps and messes with the timesteps."

create_qtc_string(*qtc*)

make_world_qsr_trace(*world_trace*, *timestamps*, *qsr_params*, *req_params*, ***kwargs*)

qtc_to_output_format(*qtc*)
Overwrite this for the different QTC variants to select only the parts from the QTCC tuple that you would like to return. Example for QTCB: return qtc[0:2]

Parameters **qtc** – The full QTCC tuple [q1,q2,q4,q5]

Returns The part of the tuple you would to have as a result using create_qtc_string

return_all_possible_state_combinations()
Method that returns all possible state combinations for the qtc_type defined for this calss instance.

Returns

- String representation as a list of possible tuples
- Integer representation as a list of lists of possible tuples

exception qsrlib_qsrs.qsr_qtc_simplified_abstractclass.**QTCException**
Bases: exceptions.Exception

qsrlib_qsrs.qsr_rcc2_rectangle_bounding_boxes_2d module

class qsrlib_qsrs.qsr_rcc2_rectangle_bounding_boxes_2d.**QSR_RCC2_Rectangle_Bounding_Boxes_2D**
Bases: *qsrlib_qsrs.qsr_rcc_abstractclass.QSR_RCC_Abstractclass*

RCC2 relations.

 # 'dc' bb1 is disconnected from bb2 # 'c' bb1 is connected to bb2

qsrlib_qsrs.qsr_rcc3_rectangle_bounding_boxes_2d module

class qsrlib_qsrs.qsr_rcc3_rectangle_bounding_boxes_2d.**QSR_RCC3_Rectangle_Bounding_Boxes_2D**
Bases: *qsrlib_qsrs.qsr_rcc_abstractclass.QSR_RCC_Abstractclass*

Computes symmetrical RCC3 relations: 'dc':disconnected, 'po':partial overlap, 'o': occluded/part of

'dc' bb1 is disconnected from bb2 # 'po' bb1 partially overlaps bb2 # 'o' bb1 is occluded or part of bb2

qsrlib_qsrs.qsr_rcc8_rectangle_bounding_boxes_2d module

class qsrlib_qsrs.qsr_rcc8_rectangle_bounding_boxes_2d.**QSR_RCC8_Rectangle_Bounding_Boxes_2D**
Bases: *qsrlib_qsrs.qsr_rcc_abstractclass.QSR_RCC_Abstractclass*

RCC8

'dc' bb1 is disconnected from bb2 # 'ec' bb1 is externally connected with bb2 # 'po' bb1 partially overlaps bb2 # 'eq' bb1 equals bb2 # 'tpp' bb1 is a tangential proper part of bb2 # 'ntpp' bb1 is a non-tangential proper part of bb2 # 'tppi' bb2 is a tangential proper part of bb1 # 'ntppi' bb2 is a non-tangential proper part of bb1

qsrlib_qsrs.qsr_rcc_abstractclass module

class qsrlib_qsrs.qsr_rcc_abstractclass.**QSR_RCC_Abstractclass**
Bases: *qsrlib_qsrs.qsr_dyadic_abstractclass.QSR_Dyadic_1t_Abstractclass*

Abstract class for the QSR makers

where,

x1, y2: the xy-coords of the top-left corner of the rectangle x2, y2: the xy-coords of the bottom-right corner of the rectangle

6.3.2 Module contents

6.4 qsrlib_ros package

6.4.1 Submodules

qsrlib_ros.qsrlib_ros_client module

class qsrlib_ros.qsrlib_ros_client.**QSRlib_ROS_Client**(*service_node_name='qsr_lib'*)
Bases: object

make_ros_request_message(*qsrlib_request_message*)
Make a QSRlib ROS service request message from standard QSRlib request message.

Parameters **qsrlib_request_message** (qsrlib.qsrlib.QSRlib_Request_Message) – The standard QSRlib request message.

Returns The ROS service request message.

Return type qsr_lib.srv.RequestQSRsRequest

request_qsrs(*req*)
Request to compute QSRs.

Parameters **req** (*qsr_lib.srv.RequestQSRsRequest*) – The request message.

Returns The ROS service response.

Return type qsr_lib.srv.RequestQSRsResponse

service_topic_names = None
dict: The topic names of the services.

6.4.2 Module contents

6.5 qsrlib_utils package

6.5.1 Submodules

qsrlib_utils.combinations_and_permutations module

qsrlib_utils.combinations_and_permutations.**possible_pairs**(*s*, *mirrors=True*)
Return possible pairs from a set of values.

Assume *s = ['a', 'b']*. Then return examples for the following calls are:

- *possible_pairs(s)* returns *[('a', 'b'), ('b', 'a')]*
- *possible_pairs(s, mirros=False)* returns *[('a', 'b')]*

Parameters

- **s** (*set or list or tuple*) – Names of the elements from which the pairs will be created.
- **mirrors** (*bool*) – Include mirrors or not.

Returns A list of pairs as tuples.

Return type list

qsrlib_utils.combinations_and_permutations.**possible_pairs_between_two_lists**(*s1*, *s2*, *mirrors=True*)
Return possible pairs between the elements of two sets.

Assume *s1 = ['a', 'b']* and *s2 = ['c', 'd']*. Then return examples for the following calls are:

- *possible_pairs_between_two_lists(s1, s2)* returns *[('a', 'c'), ('a', 'd'), ('b', 'c'), ('b', 'd'), ('c', 'a'), ('c', 'b'), ('d', 'a'), ('d', 'b')]*.
- *possible_pairs_between_two_lists(s1, s2, mirrors=False)* returns [('a', 'c'), ('a', 'd'), ('b', 'c'), ('b', 'd')].

Parameters

- **s1** (*set or list or tuple*) – Names of the first elements.
- **s2** (*set or list or tuple*) – Names of the second elements.
- **mirrors** (*bool*) – Include mirrors or not.

Returns A list of pairs as tuples.

Return type list

qsrlib_utils.ros_utils module

qsrlib_utils.ros_utils.**convert_pythondatetime_to_rostime**(*pythondatetime*)
Convert datetime from python format to ROS format.

Parameters **pythondatetime** (*datetime.datetime*) – Python format datetime.

Returns ROS time.

Return type rospy.Time

qsrlib_utils.utils module

qsrlib_utils.utils.**flatten_list**(*l*)
Flatten an irregular list, i.e. a list containing a mixture of iteratable and non-iteratable items, returning a generator object.

Parameters **l** (*list or tuple*) – The list to flatten.

Returns Flattened list as a generator. Use *list(flatten_list(l))* to get a list back.

Return type generator

qsrlib_utils.utils.**isnan**(*x*)
Check if nan.

Parameters **x** (*int or float*) – The value to be checked.

Returns Whether nan or not.

Return type bool

qsrlib_utils.utils.**load_dynamic_args_from_file**(*path*)
Load *dynamic_args* from a yaml file.

Parameters **path** (*str*) – The filename including its path.

Returns

qsrlib_utils.utils.**merge_world_qsr_traces**(*world_qsr_traces*, *qsr_type=''*)
Merge a list of traces into one world_qsr_trace. It offers no protection versus overwriting previously existing relations.

Parameters

- **world_qsr_traces** (*list or tuple*) – The World_QSR_Trace objects to be merged.
- **qsr_type** (*str*) – The QSR type of the merged object.

Returns The merged *world_qsr_traces*.

Return type *World_QSR_Trace*

6.5.2 Module contents

CHAPTER 7

Indices and tables

- genindex
- modindex

q

A

B

C

D

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

W

X

Y

Z