QSRlib Documentation

Release 0.4.0

STRANDS

Sep 01, 2017

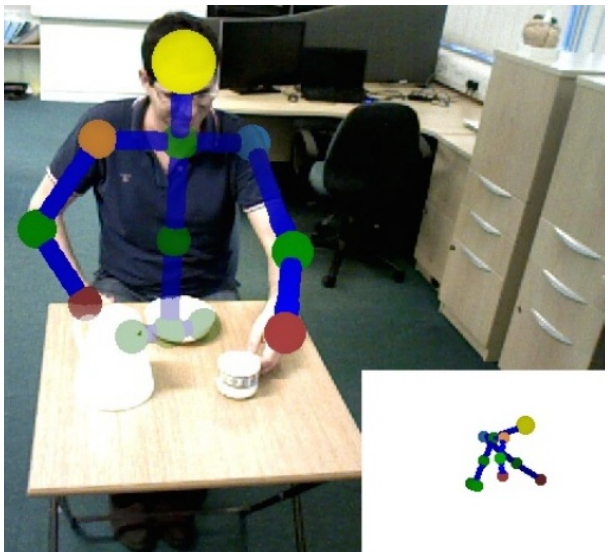
Contents

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

The aims of QSRlib are to:

- provide a number of QSRs that are well known, and in common use in scientific community;
- expose these QSRs via a standard IO interface that allows quick and easy re-usability, including a ROS interface to allow use in cognitive robotic systems;
- provide a flexible and easy to use infrastructure that allows rapid development of new QSRs that extend the library;
- deliver abstracted QSRs over time in an aggregated representation that facilitates further inference.

A typical usage of QSRlib would be an intelligent system, such as a robot for example, which acquires visual data via an RGB-D camera, such as a Kinect, and via object recognition and skeleton tracking is able to perceive the individual entities in the world. The system can then make calls to QSRlib in order to abstract this input data and form a qualitative representation of the perceived world scene. This could then be used to recognise activities in natural scenes such as the one shown in the image below, using already learnt models expressed using QSRs in the QSRlib library.



QSRlib has been used in various research and teaching projects. Some selective case studies are briefly described. In[1] QSRlib was used to rapidly experiment with multiple different types of qualitative representations in order to identify the most suitable one for learning human motion behaviours as perceived by a mobile robot that was deployed for a duration of 6 weeks in an office environment. QSRlib was used to quickly experiment with suitable representations for classifying scenes and environments from visual data[2][3]. The library was also used to compute qualitative relations between a robot and humans moving in order to plan and execute safe path navigation taking into consideration their movement patterns[4]. The library has also been used in a number of teaching projects (e.g. recognizing gestures for controlling a device, recognizing someone having breakfast, etc.), allowing the students to concentrate on the more interesting, high level, parts of their projects rather than spending a good portion of their project time in developing the low level tools they need (and which are the same from project to project).

[1] Duckworth, P.; Gatsoulis, Y.; Jovan, F.; Hawes, N.; Hogg, D. C.; and Cohn, A. G. 2016. Unsupervised Learning of Qualitative Motion Behaviours by a Mobile Robot. In Proc. of the Intl. Conf. on Autonomous Agents and Multiagent Systems (AAMAS'16).

[2] Thippur, A.; Burbridge, C.; Kunze, L.; Alberti, M.; Folkesson, J.; Jensfelt, P.; and Hawes, N. 2015. A Comparison of Qualitative and Metric Spatial Relation Models for Scene Understanding. In 29th Conference on Artificial Intelligence (AAAI'15).

[3] Kunze, L.; Burbridge, C.; Alberti, M.; Thippur, A.; Folkesson, J.; Jensfelt, P.; and Hawes, N. 2014. Com- bining Top-down Spatial Reasoning and Bottom-up Object Class Recognition for Scene Understanding. In IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'14).

[4] Dondrup, C.; Bellotto, N.; Hanheide, M.; Eder, K.; and Leonards, U. 2015. A Computational Model of Human- Robot Spatial Interactions Based on a Qualitative Trajectory Calculus. Robotics 4(1):63–102.

CHAPTER 1

Get Started

A list of included QSRs can be found in this *link*.

For installation instructions refer to the *install guidelines*.

To get started with using QSRlib read and go through the *usage examples*.

For QSR developers *this documentation page* provides all the details.

If you run into an issue please submit a ticket at https://github.com/strands-project/strands_qsr_lib/issues.

For comments and feedback please use the ticket system as well at https://github.com/strands-project/strands_qsr_lib/issues.

QSRlib is released under *MIT license*.

CHAPTER 2

QSRs

Qualitative Distance Calculus

Description

Qualitative Distance Calculus (QDC) relations define qualitative spatial distance relations between two objects according to the relations names and distance thresholds defined by the user.

Relations

The relations are defined by the user-specified relations names and the corresponding distance thresholds.

API

The API can be found *here*.

References

Probabilistic Qualitative Distance Calculus

Description

Probabilistic Qualitative Distance Calculus (PQDC) relations define qualitative spatial distance relations between two objects according to the relations names and distance thresholds defined by the user, just like *Qualitative Distance Calculus*, with the difference that they are modelled as Gaussian distributions allowing overlap between the boundaries and fuzzy selection in the common areas.

Relations

The relations are defined by the user-specified relations names and the corresponding distance thresholds.

API

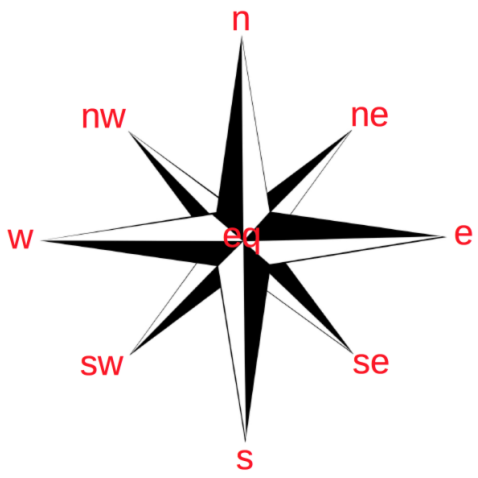
The API can be found *here*.

Cardinal Directions

Description

Cardinal Directions (CarDir) are compass relations between two objects, and their minimum set consists of four relations north (n), east (e), south (s) and west (w). Intermediate relations between the main relations are north-east (ne), south-east (se), south-west (sw) and north-west (nw). These relations are shown schematically in the compass rose shown in the figure below, with the addition of an 'equal' (eq) relation when the objects are together.

For the computation of cardinal directions between two regions we use a point-based model (examples are[1][2][3][4]), i.e. we approximate the regions by their centroid.



API

The API can be found *api*.

[1] Frank, A.U., 1990. Qualitative Spatial Reasoning about Cardinal Directions. In M. Mark & D. White, eds. Autocarto 10. Baltimore: ACSM/ASPRS.

[2] Frank, A.U., 1992. Qualitative spatial reasoning about distances and directions in geographic space. Journal of Visual Languages & Computing, 3(4), pp.343–371.

[3] Frank, A.U., 1996. Qualitative Spatial Reasoning: Cardinal Directions as an Example. Geographical Information Systems, 10(3), pp.269–290.

[4] Ligozat, G.E., 1998. Reasoning about Cardinal Directions. Journal of Visual Languages & Computing, 9(1), pp.23–44.

References

Moving or Stationary

Description

Moving or Stationary (MOS) is a simple qualitative relation of an object determining whether it is moving or not with respect to its coordinate frame origin.

Relations

The relations are simply:

- **moving** when the object is found to have changed its position between two given frames,
- **stationary** when the object is found to not have changed its position between two given frames.

API

The API can be found *here*.

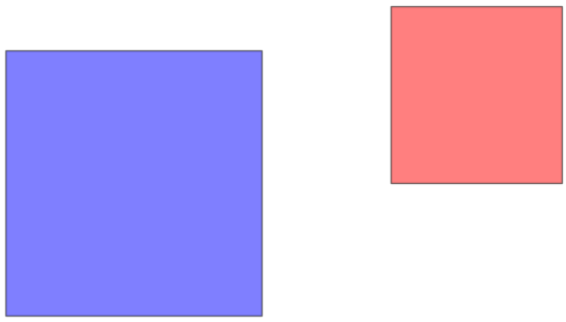
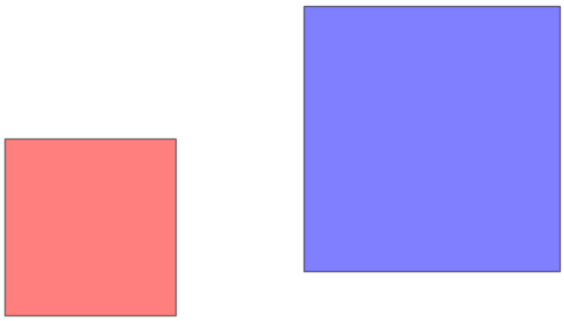
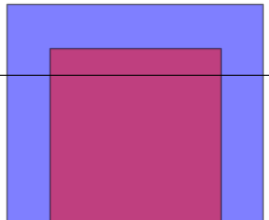
Minimal Working Example

Description

Minimal Working Example (MWE) is a simple directional QSR aimed to demonstrate how to implement a new QSR and integrate it in QSRlib.

Relations

All the possible MWE relations between a blue object X and a red object Y are:

Relation	Illustration	Interpretation
X **left** Y		X is to the left of Y.
X **right** Y		X is to the right of Y.
		

API

The API can be found *here*.

Qualitative Trajectory Calculus *b*

Description

API

The API can be found *here*.

References

Qualitative Trajectory Calculus c

Description

API

The API can be found *here*.

References

Qualitative Trajectory Calculus *bc*

Description

API

The API can be found *here*.

References

Rectangle Algebra

Description

Rectangle Algebra (RA)[12] computes *Allen's Interval Algebra* relations on the projected to their xy-axes segments between two 2D-rectangles.

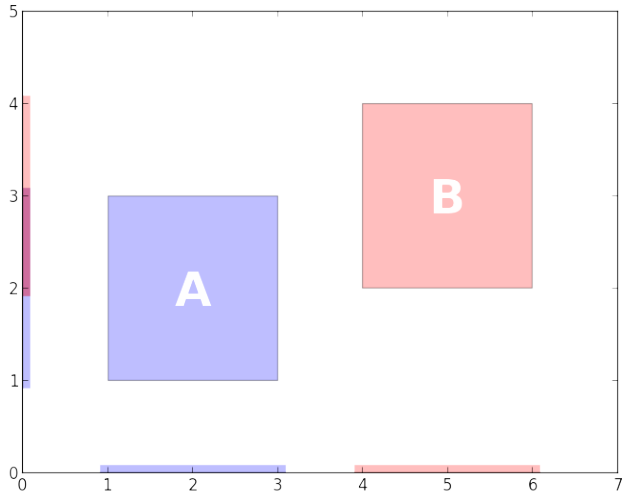
1

16. Balbiani, J.-F. Condotta and L. F. del Cerro: A model for reasoning about bi-dimensional temporal relations. In Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), A.G. Cohn, L. K. Schubert and S. C. Shapiro (eds). Morgan Kaufmann, pp. 124–130. Trento, Italy, June 2–5 1998.

2

Relations

For example the RA relation between boxes A and B in the case depicted in the figure below is `A(<, o)B`.



The full set of the RA relations is determined by *Allen's Interval Algebra* relations. Therefore, since there are 13 Allen's relations, RA defines 169 possible relations over the xy segments of two rectangles.

API

The API can be found *here*.

References

Region Connection Calculus 2

Description

Region Connection Calculus (RCC)[12] is intended to serve for qualitative spatial representation and reasoning. RCC abstractly describes regions (in Euclidean space, or in a topological space) by their possible relations to each other.

RCC2 is the simplest form of RCC and consists of 2 basic relations that are possible between two regions. It is a stripped down version of *RCC8*. The mapping from RCC8 to RCC2 can be seen below:

16. Balbiani, J.-F. Condotta and L. F. del Cerro: A new tractable subclass of the rectangle algebra. In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99), T. Dean (ed.). Morgan Kaufmann, pp. 442–447. Stockholm, Sweden, July 31–August 6, 1999.

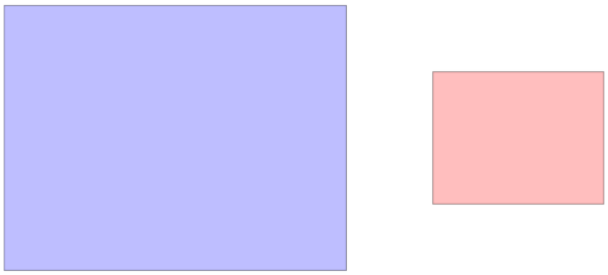
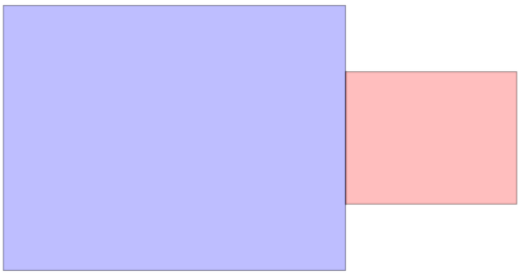
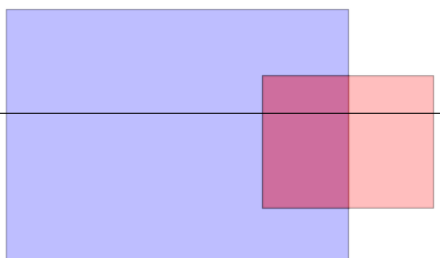
[1] Randell, D. A., Cui, Z. and Cohn, A. G.: A spatial logic based on regions and connection, Proc. 3rd Int. Conf. on Knowledge Representation and Reasoning, Morgan Kaufmann, San Mateo, pp. 165–176, 1992. (link)

[2] Anthony G. Cohn, Brandon Bennett, John Gooday, Micholas Mark Gotts: Qualitative Spatial Representation and Reasoning with the Region Connection Calculus. GeoInformatica, 1, 275–316, 1997.

RCC8	RCC2
dc	dc
ec	c
po	
tpp	
ntpp	
eq	
tppi	
ntppi	

Relations

All the possible RCC5 relations between a blue object X and a red object Y are:

Relation	Illustration	Interpretation
X **dc** Y		X is disconnected from Y.
X **c** Y		X is connected to Y.
		

API

The API can be found *here*.

References

Region Connection Calculus 4

Description

Region Connection Calculus (RCC)[1][2] is intended to serve for qualitative spatial representation and reasoning. RCC abstractly describes regions (in Euclidean space, or in a topological space) by their possible relations to each other.

RCC4 consists of 4 basic relations that are possible between two regions; it is a stripped down version of *RCC8*. The mapping from RCC8 to RCC4 can be seen below:

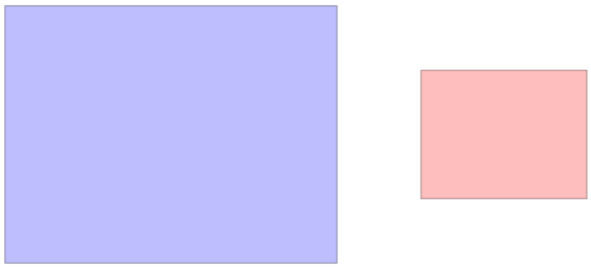
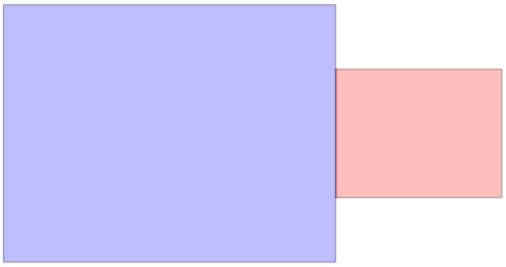
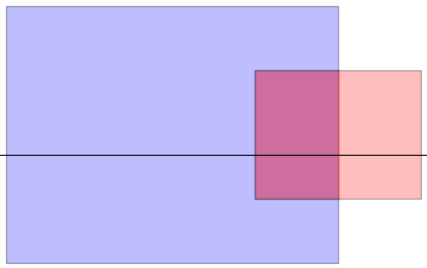
RCC8	RCC4
dc	dc
ec	po
po	
tpp	pp
ntpp	
eq	
tppi	ppi
ntppi	

Relations

All the possible RCC4 relations between a blue object X and a red object Y are:

[1] Randell, D. A., Cui, Z. and Cohn, A. G.: A spatial logic based on regions and connection, Proc. 3rd Int. Conf. on Knowledge Representation and Reasoning, Morgan Kaufmann, San Mateo, pp. 165–176, 1992. (link)

[2] Anthony G. Cohn, Brandon Bennett, John Gooday, Micholas Mark Gotts: Qualitative Spatial Representation and Reasoning with the Region Connection Calculus. GeoInformatica, 1, 275–316, 1997.

Relation	Illustration	Interpretation
X **dc** Y		X is disconnected from Y.
X **po** Y		X is partially overlapping Y.
		

API

The API can be found *here*.

References

Region Connection Calculus 5

Description

Region Connection Calculus (RCC)[1][2] is intended to serve for qualitative spatial representation and reasoning. RCC abstractly describes regions (in Euclidean space, or in a topological space) by their possible relations to each other.

RCC5 consists of 5 basic relations that are possible between two regions; it is a stripped down version of *RCC8*. The mapping from RCC8 to RCC5 can be seen below:

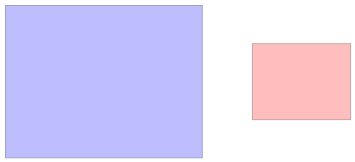
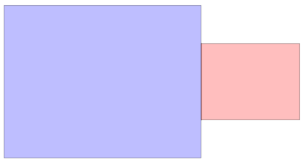
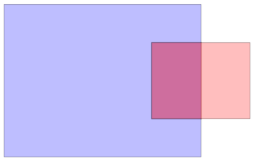
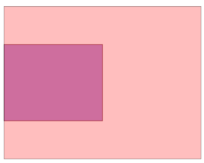
RCC8	RCC5
dc	dr
ec	
po	po
tpp	pp
ntpp	
eq	eq
tppi	ppi
ntppi	

Relations

All the possible RCC5 relations between a blue object X and a red object Y are:

[1] Randell, D. A., Cui, Z. and Cohn, A. G.: A spatial logic based on regions and connection, Proc. 3rd Int. Conf. on Knowledge Representation and Reasoning, Morgan Kaufmann, San Mateo, pp. 165–176, 1992. (link)

[2] Anthony G. Cohn, Brandon Bennett, John Gooday, Micholas Mark Gotts: Qualitative Spatial Representation and Reasoning with the Region Connection Calculus. GeoInformatica, 1, 275–316, 1997.

Relation	Illustration	Interpretation
X **dr** Y		X is discrete from Y.
X **po** Y		X is partially overlapping Y.
X **pp** Y		X is a proper part of Y.
		

API

The API can be found *here*.

References

Region Connection Calculus 8

Description

Region Connection Calculus (RCC)[1][2] is intended to serve for qualitative spatial representation and reasoning. RCC abstractly describes regions (in Euclidean space, or in a topological space) by their possible relations to each other.

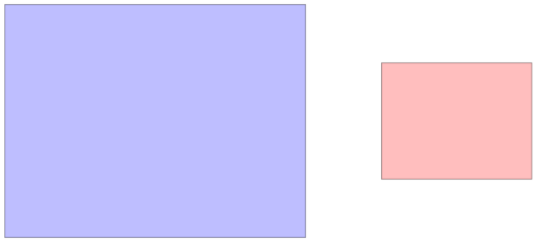
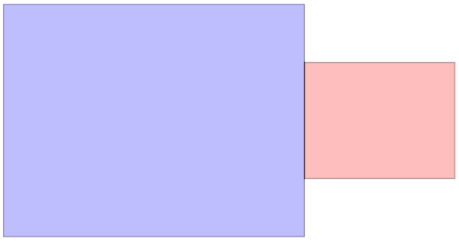
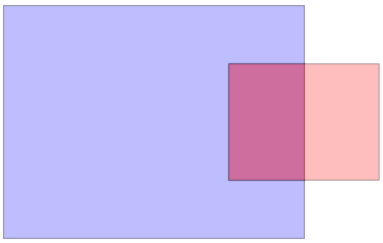
RCC8 consists of 8 basic relations that are possible between two regions.

Relations

All the possible RCC8 relations between a blue object X and a red object Y are:

[1] Randell, D. A., Cui, Z. and Cohn, A. G.: A spatial logic based on regions and connection, Proc. 3rd Int. Conf. on Knowledge Representation and Reasoning, Morgan Kaufmann, San Mateo, pp. 165–176, 1992. (link)

[2] Anthony G. Cohn, Brandon Bennett, John Gooday, Micholas Mark Gotts: Qualitative Spatial Representation and Reasoning with the Region Connection Calculus. GeoInformatica, 1, 275–316, 1997.

Relation	Illustration	Interpretation
X **dc** Y		X is disconnected from Y.
X **ec** Y		X is externally connected to Y.
		

X **po**

X is partially overlapping Y.

API

The API can be found *here*.

References

Ternary Point Configuration Calculus

Description

Ternary Point Configuration Calculus (TPCC) deals with point-like objects in the 2D-plane. It is a application based variant of the Double Cross calculus, allowing finer distinctions of positional information than the calculi presented before.

Relations

See also:

Introduction to the Ternary Point Configuration Calculus

API

The API can be found *here*.

References

Currently, the following QSRs are included in the library:

ID	Name	Links	Reference
argd	Qualitative Distance Calculus	*descr.* | *api*	[7]
argprobd	Probablistic Qualitative Distance Calculus	*descr.* | *api*	
cardir	Cardinal Directions	*descr.* | *api*	[1]
mos	Moving or Stationary	*descr.* | *api*	
mwe	Minimal Working Example	*descr.* | *api*	
qtcbs	Qualitative Trajectory Calculus *b*	*descr.* | *api*	[8][9]
qtccs	Qualitative Trajectory Calculus *c*	*descr.* | *api*	[8][9]
qtcbcs	Qualitative Trajectory Calculus *bc*	*descr.* | *api*	[8][9]
ra	Rectangle Algebra	*descr.* | *api*	[5]
rcc2	Region Connection Calculus 2	*descr.* | *api*	[2][3]
rcc4	Region Connection Calculus 4	*descr.* | *api*	[2][3]
rcc5	Region Connection Calculus 5	*descr.* | *api*	[2][3]
rcc8	Region Connection Calculus 8	*descr.* | *api*	[2][3]
tpcc	Ternary Point Configuration Calculus	*descr.* | *api*	[4]

Special Topics

Allen's Interval Algebra

Description

Allen's Interval Algebra[1] (AIA) is a system for reasoning about temporal relations. The calculus defines possible relations between time intervals and provides a composition table that can be used as a basis for reasoning about temporal descriptions of events[2].

AIA is used in a variety of QSRs and other topics in QSRlib, e.g. in the *Regional Algebra* QSR, in the *QSR graphs*, etc.

[7] Clementini, E.; Felice, P. D.; and Hernandez, D. 1997. Qualitative representation of positional information. Artificial Intelligence 95(2):317–356.

[1] Frank, A. U. 1990. Qualitative Spatial Reasoning about Cardinal Directions. In Mark, M., and White, D., eds., Au- tocarto 10. Baltimore: ACSM/ASPRS.

[8] Van de Weghe, N.; Cohn, A.; De Tre , B.; and De Maeyer, P. 2005. A Qualitative Trajectory Calculus as a basis for representing moving objects in Geographical Information Systems. Control and Cybernetics 35(1):97–120.

[9] Delafontaine, M.; Cohn, A. G.; and Van de Weghe, N. 2011. Implementing a qualitative calculus to analyse moving point objects. Expert Systems with Applications 38(5):5187–5196.

[5]

16. Balbiani, J.-F. Condotta and L. F. del Cerro: A model for reasoning about bi-dimensional temporal relations. In Proc. of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), A.G. Cohn, L. K. Schubert and S. C. Shapiro (eds). Morgan Kaufmann, pp. 124–130. Trento, Italy, June 2–5 1998.

[2]

4. (a) Randell, Z. Cui and A. G. Cohn: A spatial logic based on regions and connection. In Proc. 3rd Int. Conf. on Knowledge Representation and Reasoning, Morgan Kaufmann, San Mateo, pp. 165–176, 1992.

[3]

1. (a) Cohn, B. Bennett, J. Gooday and M. M. Gotts: Qualitative Spatial Representation and Reasoning with the Region Connection Calculus. GeoInformatica, 1, pp. 275–316, 1997.

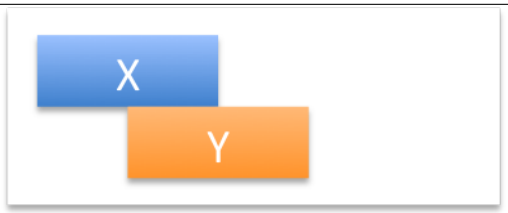
[4] Moratz, R.; Nebel, B.; and Freksa, C. 2003. Qualita- tive spatial reasoning about relative position: The tradeoff between strong formal properties and successful reasoning about route graphs. In Freksa, C.; Brauer, W.; Habel, C.; and Wender, K. F., eds., Lecture Notes in Artificial Intelligence 2685: Spatial Cognition III. Berlin, Heidelberg: Springer Verlag. 385–400.

[1] James F. Allen: Maintaining knowledge about temporal intervals. Communications of the ACM, 1983.

[2] Wikipedia: Allen's interval algebra. Wikipedia, accessed October 2015, https://en.wikipedia.org/wiki/Allen%27s_interval_algebra

Relations

Allen's Interval Algebra defines 13 possible qualitative temporal base relations between two intervals X and Y:

Relation	Inverse	Illustration	Interpretation
X < Y	Y > X		X takes place before Y.
X = Y	Y = X		X is equal to Y.
X **m** Y	Y **mi** X		X meets Y.
X **o** Y	Y **oi** X		X overlaps with Y.
X **d** Y	Y **di** X		X takes place during Y.
X **s** Y	Y **si** X		X starts Y.
X **f** Y	Y **fi** X		X finishes Y.

References

Qualitative Spatio-Temporal Activity Graphs

Description

A Qualitative Spatio-Temporal Activity Graph (QSTAG) provides a compact and efficient graph structure to represent both qualitative spatial and temporal information about entities, allowing the use of standard graph comparison techniques.

We request the QSTAG in the QSRLib framework using the dynamic argument dictionary: dynamic_args[“qstag”] in the QSRLib Request Message, and it is returned in the QSRLib Response Message as a class qstag(), with main class components episodes (list), graph (iGraph object) and graphlets() (class).

Episodes

To begin, the QSR_World_Trace is converted into a QSR Episode representation. We define a QSR episode as a compressed, subsequent, set of the same qualitative relation between a set of entities. An episode is then expressed as a tuple containing an entity set, a set of QSR relations, and an interval of time over which they hold. An example episode:

([human, door], {QTCbs: [+, 0]}, (0, 10)).

The list of all QSR episodes generated from the QSR_World_Trace is returned in:

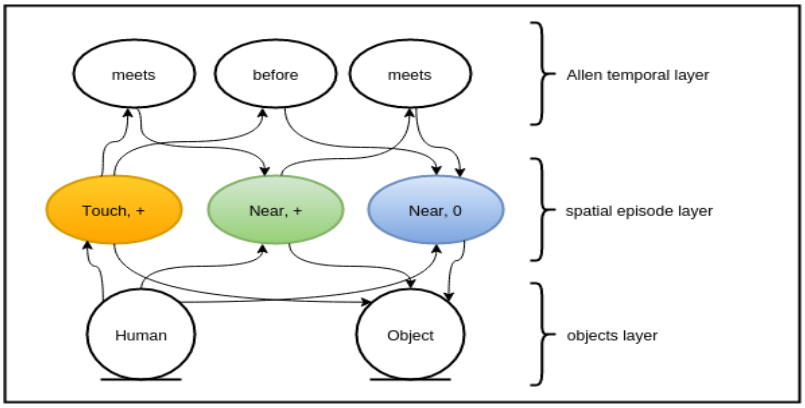
QSRLib_Response_Message.qstag.episodes

QSTAG

From a list of QSR episodes, a QSTAG is generated by abstracting over the temporal sequence of episodes. This is done by taking the Allen interval algebra (IA)[1] relation between each pair of episodes in the set. The structure of a QSTAG is comprised of the following three layers, and a directed edge set:

- An objects layer, containing one nodes per unique object in the set of QSR episodes.
- A spatial episode layer, containing one node per QSR episode in the given set.
- An Allen temporal relations layer, containing one node per pair of QSR episodes and encoding the IA relation that holds between the two QSR episodes.

An example of a QSTAG:



[1] James F. Allen: Maintaining knowledge about temporal intervals. Communications of the ACM, 1983.

consisting of two objects, three QSR episodes (encoding Argument Distance and QTCbs relations), and three IA relation nodes which hold between the episodes' intervals.

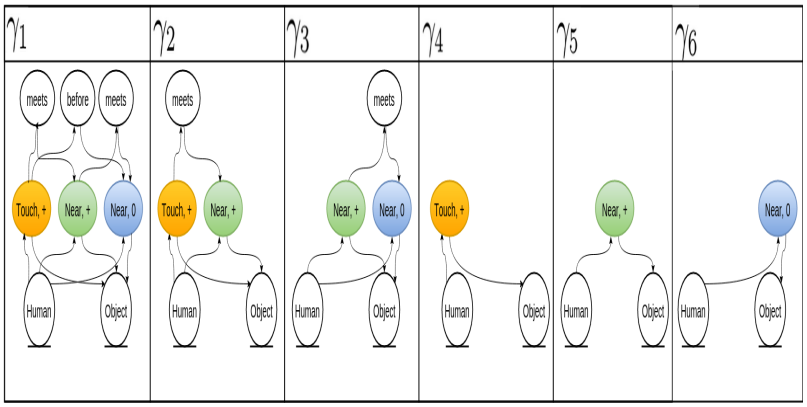
For a QSTAG, an iGraph object is returned in QSRLib_Response_Message.qstag.graph and a utility function is provided for creating a dot image file, available in qsrlib_qstag.utils.graph2dot(<< qstag >>, "<<file_path>>.dot")} module.

Graphlets

A graphical representation of a timeseries of QSRs facilitates the use of graph matching techniques. Given a QSTAG, it is possible to represent this as a set of overlapping subgraphs with certain properties. We define these sub-graphs as graphlets with properties:

- they maintain the layer structure of a QSTAG,
- the objects layer is restricted to at most n objects,
- the spatial episode layer contains up to m QSR episode nodes which are contiguous in time; this allows the sub-graphs to temporally overlap,
- the temporal layer contains up to (m*(m-1))/2 IA relation nodes (one per pair of spatial episode nodes), where n and m are supplied parameters in the dynamic argument dictionary (more details below).

This allows the QSTAG to be represented as a bag of overlapping graphlets, where each graphlet obeys the four properties. The example QSTAG above can be represented as a set of six graphlets which hold the above properties using n=2 and m=3. All are displayed here:



Graphlets() is a sub-class of a QSTAG and contains all the information about the graphlets of a QSTAG and is returned in:

QSRLib_Response_Message.qstag.graphlets

The main members of the Graphlet class are graphlets (a dictionary), code_book (a list) and histogram (a list).

Each graphlet generated from a QSTAG is represented as both an iGraph object, and also as a graph hash code for efficient comparisons. The Graphlets.graphlets dictionary combines the two as key-value pairs, where keys are unique hash codes, and values are the corresponding iGraph objects; Returned in:

QSRLib_Response_Message.qstag.graphlets.graphlets.

Also returned in the Response Message is a code book of the unique hash codes, and a histogram of the number of occurrences in the QSTAG (both lists - in the same order); Returned in:

QSRLib_Response_Message.qstag.graphlets.code_book QSRLib_Response_Message.qstag.graphlets.histogram

A code book and histogram lists are intended to be zipped together and allow for easy post analysis of multiple QSTAGs representing multiple observations. Implementation details of these attributes are given in the below section.

Usage

To use, first create a QSR_World_Trace (qsrlib_response_message) in the normal way.

Standard Steps for Creating QSR_World_Trace:

- Create a QSRlib object
- Convert your data in to QSRlib standard input format
- Make a request to QSRlib (or a request to QSRlib using ROS)

Steps for Creating a QSTAG:

Make sure "qstag" is a key of the dynamic argument dictionary when you make a request to QSRLib. Then the response message will contain a QSTAG object:

```
qstag = qsrlib_response_message.qstag
```

The value of the dynamic_args["qstag"] dictionary must contain a dictionary of graphlet *parameters*, which includes the min and max number of object rows included in the graohlet and a maximum number of spatial episode nodes. Optionally a dictionary of *object types* can also be supplied. Also, if you want the spatial relations only created for certain objects, use the *qsr_for* dictionary. An example is give here:

```
object_types = {"o1": "Human",
                "o2": "Chair"}

which_qsr = ["qtcbs", "argd", "mos"]

dynamic_args = {"qtcbs": {"quantisation_factor": args.quantisation_factor,
                          "validate": args.validate,
                          "no_collapse": args.no_collapse,
                          "qsrs_for": [("o1", "o2"),("o1", "o3")]},

                "argd": {"qsr_relations_and_values": args.distance_threshold,
                         "qsrs_for":  [("o1", "o2")]},

                "mos": {"qsrs_for": [("o1"), ("o2")]},

                "qstag": {"params" : {"min_rows":1, "max_rows":1, "max_eps":3}
                         {"object_types" : object_types}}
                }
```

Visualize the QSTAG

A utility function to save the QSTAG as a dot file, and convert it to a png image is provided in the *qsrlib_qstag.utils.graph2dot(<qstag>, "<file_path>.dot")* module. E.g.

```
qstag = qsrlib_response_message.qstag
qsrlib_qstag.utils.graph2dot(qstag, '/tmp/act_gr.dot')
os.system('dot -Tpng /tmp/act_gr.dot -o /tmp/act_gr.png')
```

Parse the Episodes, QSTAG and Graphlets

```
qstag = qsrlib_response_message.qstag

print("All the Episodes...")
```

```
for episode in qstag.episodes:
    print(episode)

print("The QSTAG iGraph: \n", qstag.graph)

print("All the Graph NODES:")
for node in qstag.graph.vs():
    print(node)

print("All the Graph EDGES:")
for edge in qstag.graph.es():
    print(edge, " from: ", edge.source, " to: ", edge.target)

print("Graphlets:")
for i, j in qstag.graphlets.graphlets.items():
    print("\n", i, j)

print("Histogram:")
for i, j in zip(qstag.graphlets.code_book, qstag.graphlets.histogram):
    print("\n", i, ": ", j)
```

Example of QSTAG code

An example script for generating a simple QSTAG is available in */strands_qsr_lib/qsr_lib/scripts/*:

```
./qstag_example.py <qsr_name>
```

e.g.

```
./qstag_example.py qtcbs
```

References

Allen's Interval Algebra

Allen's Interval Algebra is a calculus for temporal reasoning. For further details see *this page*.

Qualitative Spatio-Temporal Activity Graphs

QSRlib provides also functionalities to represent time-series QSRs as a graph structure, called *Qualitative Spatio-Temporal Activity Graphs* (QSTAG). For details, please refer to its *documentation*.

References

CHAPTER 3

Installation

Dependencies

- *numpy* which usually gets installed with python.
- *matplotlib* is used by some visualization modules, but again it should be already installed in your system.
- *igraph* (used by *QSTAGs*).
  - In linux it should be in your distro's repositories. For example in modern ubuntu you can install it as `apt-get install python-igraph` (don't forget the *sudo* if needed).
  - For other systems please refer to igraph installation instructions.

Install as

You can install QSRlib as a standalone python package, or if you have ROS installed and you want to use it within ROS you can install as a ROS package. The installation steps for each are explained below.

standalone python package

Installation as a python package consists of the following two steps:

1. Clone the QSRlib git repository.
2. Include the QSRlib source folder.

For example, let's say you want to install it in your home directory. Then the bash commands for the above two steps are:

```
git clone https://github.com/strands-project/strands_qsr_lib.git
export PYTHONPATH=$HOME/strands_qsr_lib/qsr_lib/src:$PYTHONPATH
```

Note: You can also include the export in your *.bashrc* file.

ROS catkin package

You need to firstly follow the installation instructions for ROS. Then clone the QSRlib git repository in your catkin workspace src folder, and don't forget to make it.

Note: For STRANDS developers and users, note that QSRlib can be installed using the usual software guidelines.

Verify installation

Depending on whether it has been installed as a standalone python package as a ROS package, you can test that the installation and setup works as follows repsectively:

standalone python package

If QSRlib was installed as standalone python package, then firstly from the root directory where you installed QSRlib go to *qsr_lib/scripts/* directory and run the *mwe.py* script as follows:

```
./mwe rcc8
```

If you see an output similar to the following then everything is working fine.

```
rcc8 request was made at  2016-05-17 10:14:39.482158 and received at 2016-05-17␣
↪10:14:39.482166 and finished at 2016-05-17 10:14:39.482805
---
Response is:
0.0: o2,o1:{'rcc8': 'dc'}; o2,o3:{'rcc8': 'po'}; o1,o3:{'rcc8': 'po'}; o1,o2:{'rcc8':
↪'dc'}; o3,o2:{'rcc8': 'po'}; o3,o1:{'rcc8': 'po'};
1.0: o2,o1:{'rcc8': 'dc'}; o2,o3:{'rcc8': 'po'}; o1,o3:{'rcc8': 'po'}; o1,o2:{'rcc8':
↪'dc'}; o3,o2:{'rcc8': 'po'}; o3,o1:{'rcc8': 'po'};
2.0: o2,o1:{'rcc8': 'po'}; o2,o3:{'rcc8': 'po'}; o1,o3:{'rcc8': 'po'}; o1,o2:{'rcc8':
↪'po'}; o3,o2:{'rcc8': 'po'}; o3,o1:{'rcc8': 'po'};
3.0: o2,o1:{'rcc8': 'ec'}; o2,o3:{'rcc8': 'po'}; o1,o3:{'rcc8': 'po'}; o1,o2:{'rcc8':
↪'ec'}; o3,o2:{'rcc8': 'po'}; o3,o1:{'rcc8': 'po'};
4.0: o2,o1:{'rcc8': 'ec'}; o2,o3:{'rcc8': 'po'}; o1,o3:{'rcc8': 'dc'}; o1,o2:{'rcc8':
↪'ec'}; o3,o2:{'rcc8': 'po'}; o3,o1:{'rcc8': 'dc'};
```

ROS catkin package

If QSRlib was installed as a ROS package and given that your ROS installation is setup and works, then firstly bring a *roscore* up if you don't have one already:

```
roscore
```

Then bring the QSRlib server node up:

```
rosrun qsr_lib qsrlib_ros_server.py
```

You should see a message:

```
[INFO] [WallTime: 1463477006.669860] QSRlib_ROS_Server up and running, listening to:␣
↪qsr_lib/request
```

Then you can request QSRs by running for example:

```
rosrun qsr_lib example_extended.py rcc8 --ros
```

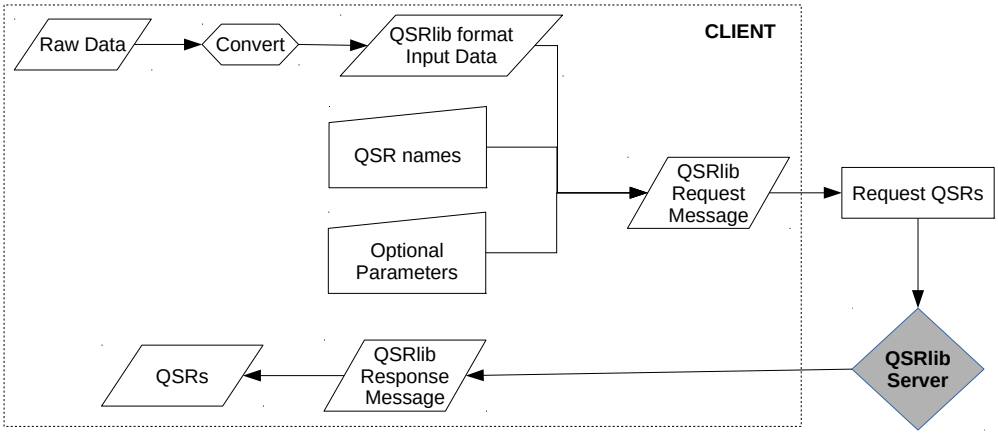
And you should see an output similar to the one below:

```
rcc8 request was made at  2016-05-17 10:24:49.355499 and received at 2016-05-17␣
↪10:24:50.014800 and finished at 2016-05-17 10:24:50.015549
---
Response is:
0.0: o2,o1:{'rcc8': 'dc'}; o1,o2:{'rcc8': 'dc'}; o1,o3:{'rcc8': 'dc'}; o1,o4:{'rcc8':
↪'dc'}; o2,o4:{'rcc8': 'dc'}; o2,o3:{'rcc8': 'dc'}; o3,o2:{'rcc8': 'dc'}; o3,o1:{
↪'rcc8': 'dc'}; o3,o4:{'rcc8': 'ntpp'}; o4,o3:{'rcc8': 'ntppi'}; o4,o2:{'rcc8': 'dc'}
↪; o4,o1:{'rcc8': 'dc'};
1.0: o2,o1:{'rcc8': 'dc'}; o1,o2:{'rcc8': 'dc'}; o1,o3:{'rcc8': 'dc'}; o1,o4:{'rcc8':
↪'dc'}; o2,o4:{'rcc8': 'dc'}; o2,o3:{'rcc8': 'dc'}; o3,o2:{'rcc8': 'dc'}; o3,o1:{
↪'rcc8': 'dc'}; o3,o4:{'rcc8': 'ntpp'}; o4,o3:{'rcc8': 'ntppi'}; o4,o2:{'rcc8': 'dc'}
↪; o4,o1:{'rcc8': 'dc'};
2.0: o2,o1:{'rcc8': 'dc'}; o1,o2:{'rcc8': 'dc'}; o1,o3:{'rcc8': 'ec'}; o1,o4:{'rcc8':
↪'po'}; o2,o4:{'rcc8': 'dc'}; o2,o3:{'rcc8': 'dc'}; o3,o2:{'rcc8': 'dc'}; o3,o1:{
↪'rcc8': 'ec'}; o3,o4:{'rcc8': 'ntpp'}; o4,o3:{'rcc8': 'ntppi'}; o4,o2:{'rcc8': 'dc'}
↪; o4,o1:{'rcc8': 'po'};
```

CHAPTER 4

For users

The following figure presents a flowchart with the main step processes for computing QSRs via the library. Raw data first needs to be converted into the common input data format of QSRlib, which represents a timeseries of the states of the perceived objects, such Cartesian position and rotation, size of the object in each dimension, and allows other custom information about the objects to be kept on a per QSR-need basis. Utility functions are provided that allow easy conversion of the raw data to this standard input data structure. This input data structure, the names of the requested QSRs to be computed and other options that control their behaviours are used to create a request message to the QSRlib server, which upon computation returns a response message that includes the computed QSRs as an output data structure similar to the input one, i.e. a timeseries of the QSRs between the objects, as well as other information.



The following minimal working example explains how to conduct these steps and use the library to compute QSRs from your data.

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)
```

Line-by-line explanation

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.

Create a *QSRlib* object

```
qsrlib = QSRlib()
```

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

Convert your data in to QSRlib standard input format

Below is one way of creating your input data. You can find more details on how to convert your data in QSRlib input format in the Section about the *I/O data structures*.

```
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 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)
```

Via 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)
```

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)
```

Advanced Topics

I/O data structures

Input: *World_Trace*

QSRlib acccepts input in its own standard format, which is a *World_Trace* object.

World_Trace provides a number of convenience methods to convert your data into this format. One additional handy method, further to the one presented earlier in the *MWE* section, is *add_object_track_from_list*, which allows to add an object's positions stored in a list of tuples.

The main member of *World_Trace* is `trace`, which is a python dictionary with keys being timestamps and values being objects of the class *World_State*. In a *World_State* object the main member is `objects`, which is again a dictionary with keys being the unique name of the object and values objects of the class *Object_State*. An *Object_State* object holds the information about an object in the world at that particular timestamp.

So for example to get the `x`-coordinate of an object called `o1` at timestamp `4` from a *World_Trace* object called `world` we would write:

```
world.trace[4].objects['o1'].x
```

Note: *World_Trace* should not be confused with the *QSRlib request message*.

Output: *World_QSR_Trace*

The standard output data structure is an object of type *World_QSR_Trace*.

The main member of *World_QSR_Trace* is `trace`, which is a dictionary with keys being the timestamps of the QSRs and usually matching those of *World_Trace* (depends on QSR type, request parameters, missing values, etc.), and values being objects of the class *World_QSR_State*. In a *World_QSR_State* object the main member is `qsrs`, which is a dictionary where the keys are unique combinations of the objects for which the QSR is, and values being objects of the class *QSR*. The QSR object holds, among other information, the QSR value.

For example, for readinging the *RCC8* relation at timestamp `4` between objects `'o1,o2'` of a `world_qsr` object we would do:

```
world_qsr.trace[4].qsrs['o1,o2'].qsr['rcc8']
```

A number of convenience slicing functions exist in the *class*.

Note: *World_QSR_Trace* should not be confused with the *QSRlib response message*.

Request/Response messages

Request message

Once we have our input data in the standard QSRlib input format, i.e. as a *World_Trace* object, the next step is to create a request message that is passed as argument in the QSRlib request call (as also explained in the *MWE example*).

The request message is an object of the class `QSRlib_Request_Message`. The compulsory arguments are *input_data* which is your *World_Trace* object that you created before and the *which_qsr* which is your requested QSR. If you want only one QSR to be computed it is a string, otherwise if you want to compute multiple QSRs in one call pass their names as a list. The optional argument *req_made_at* can be safely ignored. The second optional argument *dynamic_args* is in brief a dictionary with your call and QSR specific parameters.

Response message

The response message of QSRlib is an object of the class `QSRlib_Response_Message`. The main field of it is the *qsrs* one that holds your computed QSRs as a *World_QSR_Trace* object. The remaining ones are simply timestamps of the process and can be safely ignored.

dynamic_args

Requesting QSRs for specific objects only

Each QSR has a default behavior for which objects to compute QSRs for. Typically, this is for all valid combinations of the objects in the *World_Trace*. One way to compute QSRs for specific objects is to use the slicing utilities and subsample the *World_Trace*. Still, this might not have the desired effect as most QSRs will also create relations for mirror pairs as well, e.g. the RCC relations computed for two objects o1 and o2 will be for both o1,o2 as well as o2,o1.

For these reasons QSRlib allows the user to specify valid objects that are passed in the request in the *dynamic_args* field. It is easier to explain the usage with the examples shown below.

For all cases assument that we have a *World_Trace* of three objects o1, o2 and o3, and we want to compute two dyadic QSRs *CARDIR* and *RCC8*, and one monadic *MOS*.

By default the dyadic QSRs will be computed for o1,o2; o1,o3; o2,o3; o3,o1; and o3,o2; and mos relations will be computed for o1;o2; and o3.

Example 1: Suppose we want to compute QSR relations for o1,o2 for the dyadic QSRs and for o1 and o2 for MOS. All we need to do is define a *dynamic_args* as follows (and then pass it to our request).

```
dynamic_args = {'for_all_qsrs': {'qsrs_for': [('o1', 'o2'), 'o1', 'o2']}}
```

Example 2: Now, suppose that we want to compute CARDIR relations for o1,o2 and o2,o3, RCC8 relations for o1,o3 and MOS relations for o1 only. This is possible by defining the following *dynamic_args*:

```
dynamic_args = {'cardir': {'qsrs_for': [('o1', 'o2'), ('o2', 'o3')]},
                'rcc8': {'qsrs_for': [('o1', 'o3')]},
                'mos': {'qsrs_for': ['o1']}}
```

Note: We can mix the global namespace *'for_all_qsrs'* with the QSR specific namespace, but note that parameters in the QSR namespace always take precedence over the global one.

QSR specific parameters

Further to the *qsrs_for* option of dynamic args which is common for all QSRs, some of the QSRs allow some form of customization during the request call via their namespace in *dynamic_args*. What options are available depends on each QSR and is the options are given and described in their own API pages.

An example is shown below using *MOS*, which can take a parameter called *'quantisation_factor'* that determines the minimum distance of an object between two frames in order to be considered that it has moved.

```
dynamic_args = {'mos': {'quantisation_factor': 1.0}}
```

Of course we can still mix QSR specific parameters with common ones. So we wanted to compute MOS relations with the above quantisation factor for only object o1 when there are more we could do:

```
dynamic_args = {'mos': {'quantisation_factor': 1.0,
                        'qsrs_for': ['o1']}}
```

Graph representation

QSRlib provides also functionalities to represent time-series QSRs as a graph structure, called *Qualitative Spatio-Temporal Activity Graphs* (QSTAG). For details, please refer to its *documentation*.

ROS calls

The example of a ROS call in the *MWE* provides a good summary of the usage. For further details have a look in the API of the *ROS QSRlib client*.

CHAPTER 5

For developers

This section provides information for developers that wish extend QSRlib with new QSRs. This process consists of two steps:

1. Implement a new QSR
2. Register the new QSR with QSRlib

Howto: New QSRs

Implementation

Find below a minimally working example:

```
from __future__ import print_function, division
from qsrlib_qsrs.qsr_dyadic_abstractclass import QSR_Dyadic_1t_Abstractclass

class QSR_MWE(QSR_Dyadic_1t_Abstractclass):
    _unique_id = "mwe"
    _all_possible_relations = ("left", "together", "right")
    _dtype = "points"

    def __init__(self):
        super(QSR_MWE, self).__init__()

    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). For now, we need to define the following abstract properties.

```
_unique_id = "mwe"
_all_possible_relations = ("left", "together", "right")
_dtype = "points"
```

- *_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.
- *all_possible_relations*: A list (or tuple) of all the possible values that the QSR can take. It can be anything you like.
- *_dtype*: With what type of data your QSR works with. For example, they might be points, or they might be bounding boxes. For what you can use look in the *QSR abstractclass self._dtype_map*.

Then you need to write one function that computes the QSR.

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

Note: There are different types of parent classes that you can inherit from. You can see them in the *qsr_monadic_abstractclass.py*, *qsr_dyadic_abstractclass.py*, and, *qsr_triadic_abstractclass.py* module 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 one level higher, i.e. from *QSR_Dyadic_Abstractclass* (or from *QSR_Monadic_Abstractclass*). In this case you will also have to provide your own *make_world_qsr_trace* (see the special cases for some example ideas).

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 and we will consider implementing it the quickest possible.

Registration

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)
```

Advanced Topics

QSR specific parameters

It is possible to change the behavior of a QSR via passing dynamically during the request call argument parameters in one of its fields that is called *dynamic_args*. It is recommended to read first the documentation on how it is used in *this page*.

In order use QSR specific parameters you will have to overwrite the method `_process_qsr_parameters_from_request_parameters(self, req_params, **kwargs)` in your QSR implementation.

Below is an example on how to do it from *MOS* QSR.

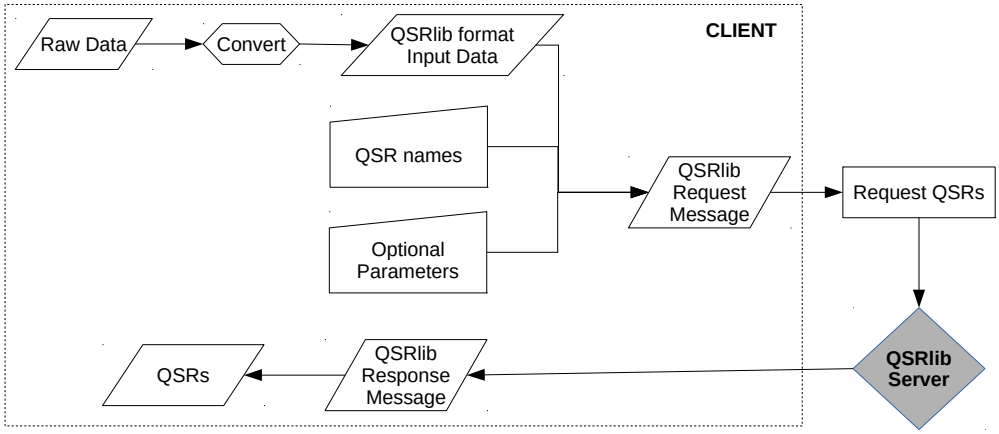
```
def _process_qsr_parameters_from_request_parameters(self, req_params, **kwargs):
    """Extract QSR specific parameters from the QSRlib request call parameters.

    :param req_params: QSRlib request call parameters.
    :type req_params: dict
    :param kwargs: kwargs arguments.
    :return: QSR specific parameter settings.
    :rtype: dict
    """
    qsr_params = self.__qsr_params_defaults.copy()
    try:
        qsr_params["quantisation_factor"] = float(req_params["dynamic_args"][self._
↪unique_id]["quantisation_factor"])
    except (KeyError, TypeError):
        try:
            qsr_params["quantisation_factor"] = float(req_params["dynamic_args"]["for_
↪all_qsrs"]["quantisation_factor"])
        except (TypeError, KeyError):
            pass
    return qsr_params
```

Note: Make sure that the QSR namespace **has precedence** over the global *'for_all_qsrs'* one.

Software Architecure

General overview

QSRlib is based on a client-server architecture implemented in python 2.7 although measures for compatibility with 3.x have been adopted. Furthermore, the library is seamlessly exposed to ROS, via a provided interface. The following figures presents a flowchart with the main step processes for computing QSRs via the library. Raw data first needs to be converted into the common input data format of QSRlib, which represents a timeseries of the states of the perceived objects, such Cartesian position and rotation, size of the object in each dimension, and allows other custom information about the objects to be kept on a per QSR-need basis. Utility functions are provided that allow easy conversion of the raw data to this standard input data structure. This input data structure, the names of the requested QSRs to be computed and other options that control their behaviours are used to create a request message to the QSRlib server, which upon computation returns a response message that includes the computed QSRs as an output data structure similar to the input one, i.e. a timeseries of the QSRs between the objects, as well as other information. This was explained in detail in the *for users* section.



The following tree shows the list of the main packages and classes that provide the key functionalities.

src/
- *qsrlib*: Provides the client-server architecture
  - *QSRlib*: Provides the server side functionalities
- *qsrlib_io*: Provides the standard IO data structures
  - *QSRlib_Request_Message*: The input to the server request
  - *QSRlib_Response_Message*: The return response from the server
  - *World_Trace*: Standard input format that QSRs are computed upon
  - *World_QSR_Trace*: Standard output format that holds the computed QSRs from a *World_Trace*
- *qsrlib_qsrs*: Provides the implementations of the QSRs
  - QSR classes: Each QSR is implemented as a separate class inheriting from one of the standard prototypes
- *qsrlib_qstag*: Provides spatio-temporal graph representations
  - *Activity_Graph*: Implements a spatio-temporal graph representation for a timeseries of QSRs
- *qsrlib_ros*: Provides the ROS client module
- *qsrlib_utils*: Provides common utilities used by the other packages

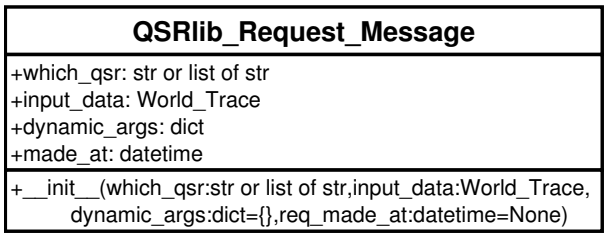
Packages and classes

QSRlib class

QSRlib
-qsrs_registry: dict
+__init__(help:bool=False) +request_qsrs(req_msg:QSRlib_Request_Message): QSRlib_Response_Message +help()

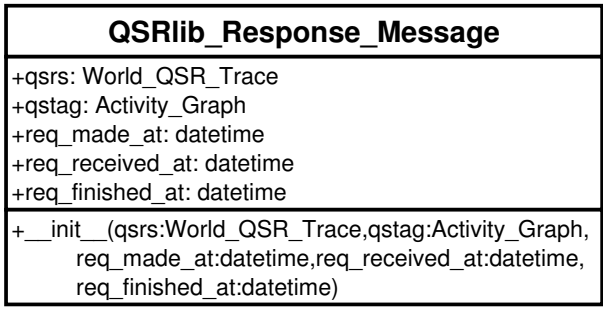
The `QSRlib` class is responsible for handling the requests via the *request_qsrs* method, which takes a *QSRlib_Request_Message* object as an argument. It also holds a registry of the QSRs included in the library through a dictionary whose keys are the unique names of the QSRs and values are class instantiations of the corresponding QSR classes.

Request message class



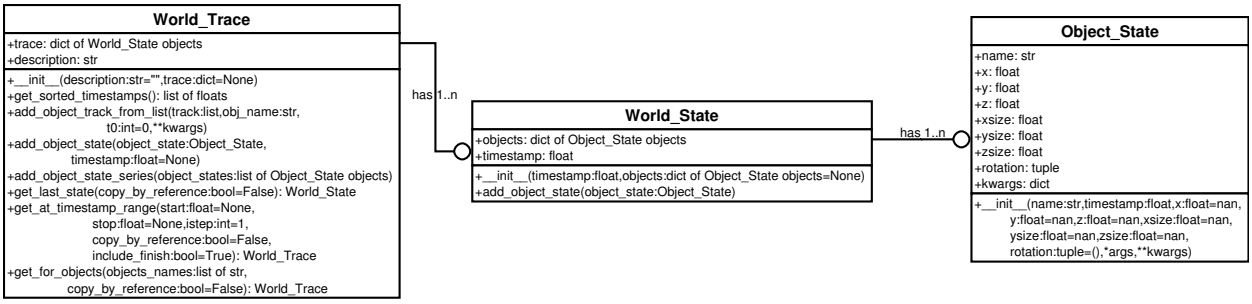
The information needed for QSRlib to process a request is packed in the `QSRlib_Request_Message` class. The minimum information needed is the input data (*input_data*) in the format of a *World_Trace* object and which QSR(s) are to be computed (*which_qsr*) in the form of a unique string QSR identifier for computing a single QSR, or a list of string identifiers for multiple ones. Optionally, it is possible to change the default behaviours of the requested QSRs by passing a dictionary of appropriate values in the *dynamic_args* argument.

Response message class



The information returned by QSRlib is an object of the class `QSRlib_Request_Message`, which mainly consists of the computed QSRs in the *qsrs* member and is a *World_QSR_Trace* object. If requested, a graph representation of QSRs, which contains additional temporal information about them is also returned in the *qstag* member.

Input data structure classes



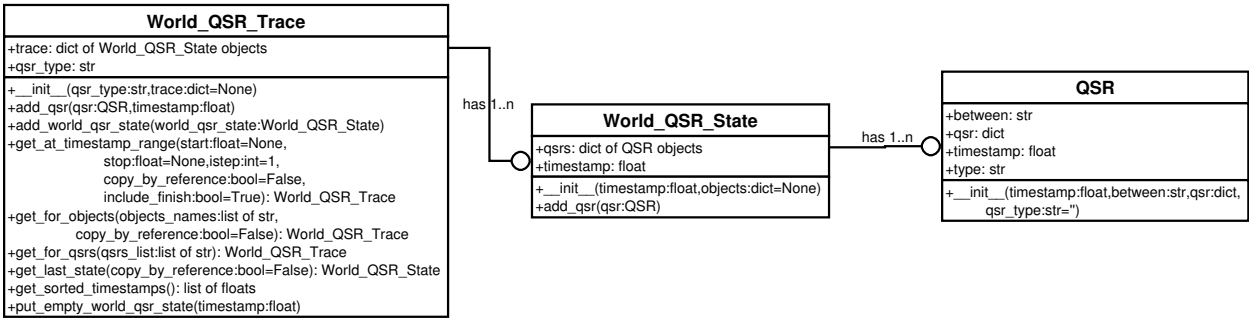
QSRs usually require input Cartesian poses of some objects (e.g. TPCC and distance-based), size of the objects as they work with regions (e.g. RCC and RA), time-series of poses (e.g. QTC). As such, a common and complete representation is needed in order to be able to re-use the data easily and transparently with a number of different QSRs.

For this reason, QSRlib uses its own input format, which is an object of the `World_Trace` class. This allows to re-use the input data, once the raw data are converted into this standard input format, without the need to do each time

specific pre-processing depending on the QSRs' requirements. A further advantage is that developers of new QSRs can expect that the input will always have the same structure.

The variable that holds the objects data is the *World_Trace.trace* member, which is a python dictionary with keys being float timestamps and values being objects of the class `World_State`. The *World_State* class describes the state of the world at one particular time. Its main members are *timestamp* which is a float variable representing the time of this world state and is the same as the corresponding key in *World_Trace.trace* dictionary, and, *objects* which holds the information about the objects. Like *trace*, *objects* is a dictionary with keys being the unique name of the object and values being objects of the class `Object_State`. Finally, an *Object_State* object holds the information about an object in the world at that particular timestamp. *Object_State* has already members for the most common spatial information about an object, such as coordinates, size and rotation, and allows dynamic expandability if needed by a QSR via dynamic arguments in its constructor.

Output data structure class

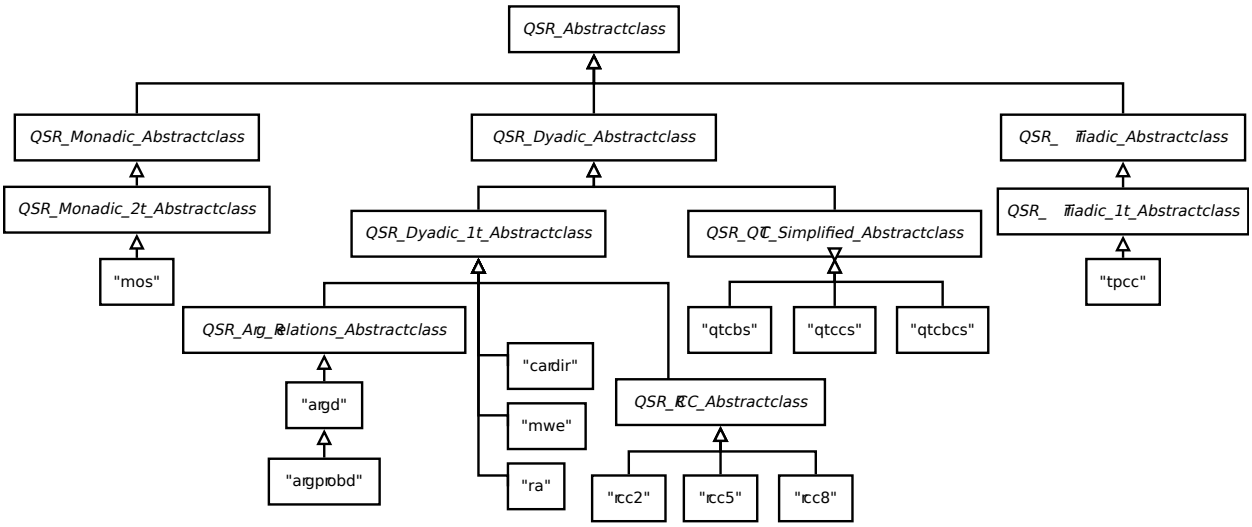


QSRs are commonly represented symbolically, for example *RCC8* relations are denoted as 'dc' for 'disconnected', 'po' for 'partially overlapping', 'eq' for 'is equal to', etc. The output of QSRlib is in a standard format similar to the input one. It is an object of the class `World_QSR_Trace`. The main member is *trace* which is a dictionary where typically with the exception of specific QSRs (e.g. validated QTC series) the keys are the same timestamps as in the *World_Trace.trace* dictionary, and the values are objects of `World_QSR_State`. A *World_QSR_State* object holds the QSRs at a particular time in the form of a dictionary, called *qsrs*, which has as keys unique string identifiers obtained from the object(s), and values are objects of the class `QSR`. The main member of a *QSR* object is a dictionary, called *qsr*, that has as keys the unique names of the QSRs and values the corresponding computed QSR strings.

QSR classes

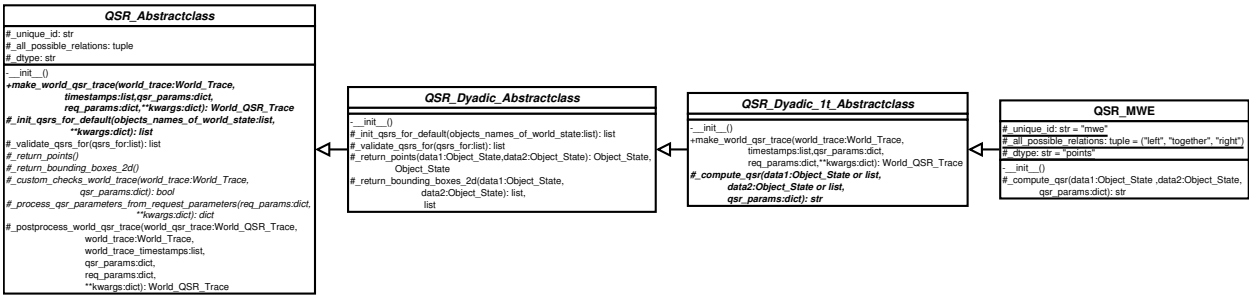
Each QSR is implemented in its own class which inherits from one of the pre-defined abstract prototype classes. The lower level, with respect to inheritance hierachy, prototypes aim to make the implementation of new QSRs as quick and easy as possible by providing interfaces that hide most of the common QSRlib backend code. Higher level prototypes provide incremental freedom at the cost of writing more code, with the top level abstract class giving complete flexibility to implement any type of QSR.

The following diagram shows a sample of the existing QSRs and their inheritance hierarchy.



Typically a QSR class should not need to inherit directly from the top level prototype (*QSR_Abstractclass*). Level 2 abstract classes specify generic prototypes for computing QSRs over a single object (*QSR_Monadic_Abstractclass*), a pair of objects (*QSR_Dyadic_Abstractclass*) or three objects (*QSR_Triadic_Abstractclass*). If a new QSR requires four objects or more then it can inherit directly from the top level (*QSR_Abstractclass*). Alternatively, it can firstly implement a level~2 abstract class in a similar manner to the others (and optionally a level~3 also), and inherit directly from it; this is the recommended approach. Level~3 abstract classes are common time-specific special cases. For example, *QSR_Monadic_2t_Abstractclass* implements the interface for QSRs that require input data of a single object over two different time points; *QSR_Dyadic_1t_Abstractclass* requires input data from two different objects for a single time point, etc.

A specific example of a QSR inheritance is shown in the following UML diagram.



CHAPTER 6

License

CHAPTER 7

People

- Yiannis Gatsoulis <y.gatsoulis@leeds.ac.uk>, University of Leeds
- Christian Dondrup <cdondrup@lincoln.ac.uk>, University of Lincoln
- Chris Burbridge <c.j.c.burbridge@cs.bham.ac.uk>, University of Birmigham
- Peter Lightbody <pet1330@gmail.com>, University of Lincoln
- Paul Duckworth <scpd@leeds.ac.uk>, University of Leeds
- Muhannad Alomari <scmara@leeds.ac.uk>, University of Leeds
- Marc Hanheide <marc@hanheide.net>, University of Lincoln
- Nick Hawes <n.a.hawes@cs.bham.ac.uk>, University of Birmigham
- Tony Cohn <a.g.cogn@leeds.ac.uk>, University of Leeds

We would like to gratefull acknowledge the financial support of the EC under FP7 project STRANDS, 600623.

CHAPTER 8

API

qsrlib package

Submodules

qsrlib.qsrlib module

Module contents

qsrlib_io package

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.

args = None
Optional args.

kwargs = None
Optional kwargs.

name = None
str: 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** (*positive int or float*) – If object has no x-size (i.e. a point) then compute bounding box based on this minimal x-size.
- **ysize_minimal** (*positive int or float*) – If object has no y-size (i.e. a point) then compute bounding box based on this minimal y-size.

Returns The 2D coordinates of the first (closest to origin) and third (farthest from origin) corners of the bounding box.

Return type list of 4 int or float

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

timestamp = None
float: 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.

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: 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.

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.

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

It is capable of handling 2D and 3D points, 2D and 3D bounding boxes.

Basically:

•2D points: tuples have length of 2 (x, y).

•3D points: tuples have length of 3 (x, y, z).

•2D bounding boxes: tuples have length of 4 (x, y, xsize,y_size).

•3D bounding boxes: tuples have length of 6 (x, y, z, xsize, ysize, zsize).

Parameters

- **track** (*list or tuple of list(s) or tuple(s)*) – List of tuples of data.
- **obj_name** (*str*) – Name of the object.
- **t0** (*int or float*) – First timestamp to offset timestamps.
- **kwargs** – kwargs arguments.

description = None
str: Optional description of the world.

get_at_timestamp_range(*start=None*, *stop=None*, *istep=1*, *copy_by_reference=False*, *include_finish=True*)
Return a subsample between start and stop timestamps.

Parameters

- **start** (*int or float*) – Start timestamp.
- **stop** (*int or float*) – Finish timestamp. If empty then stop is set to the last timestamp.
- **istep** (*int*) – subsample based on step measured in timestamps list index
- **copy_by_reference** (*bool*) – Return a copy or by reference.
- **include_finish** (*bool*) – Whether to include or not the world state at the stop timestamp.

Returns Subsample between start and stop.

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 of str*) – Requested objects names.
- **copy_by_reference** (*bool*) – Return a copy or by reference.

Returns Subsample for the requested objects.

Return type *World_Trace*

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

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

Returns The last state in *self.trace*.

Return type *World_State*

get_sorted_timestamps()
Return a sorted list of the timestamps.

Returns Sorted list of the timestamps.

Return type list

trace = None
dict: 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. Multiple objects are comma separated, e.g. "o1,o2".

qsr = None
dict: 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: Timestamp of the QSR, which usually matches the corresponding key *t* in *World_QSR_Trace.trace[t]*.

type = None
str: 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.

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: 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) – QSR object to be added.
- **timestamp** (*float*) – Timestamp of the QSR.

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

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

get_at_timestamp_range(*start=None*, *stop=None*, *istep=1*, *copy_by_reference=False*, *include_finish=True*)
Return a subsample between start and stop timestamps.

Parameters

- **start** (*int or float*) – Sstart timestamp.
- **stop** (*int or float*) – Finish timestamp. If empty then stop is set to the last timestamp.
- **istep** (*int*) – subsample based on istep measured in timestamps list index
- **copy_by_reference** (*bool*) – Return a copy or by reference.
- **include_finish** (*bool*) – Whether to include or not the world state at the stop timestamp.

Returns Subsample between start and stop.

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 of str*) – Requested objects names.
- **copy_by_reference** (*bool*) – Return a copy or by reference.

Returns Subsample for the requested objects.

Return type *World_QSR_Trace*

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

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

Returns 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 a copy or by reference.

Returns Last world QSR state in *self.trace*.

Return type *World_QSR_State*

get_sorted_timestamps()
Return a sorted list of the timestamps.

Returns Sorted list of the timestamps.

Return type list of floats

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: Name of the QSR. For multiple QSRs it is usually a sorted comma separated string.

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

Module contents

qsrlib_qsrs package

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 contains:
  - "points": self._return_points
  - "bounding_boxes": self._return_bounding_boxes_2d
  - "bounding_boxes_2d": self._return_bounding_boxes_2d

all_possible_relations
Getter for all the possible relations of a QSR.

Returns *self._all_possible_relations*

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*) – Request parameters.

Returns Computed world qsr trace.

Return type *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* , etc.) need to provide this function.
- When inheriting from one of the special case meta-abstract classes (e.g. *QSR_Monadic_2t_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** (*World_Trace*) – Input data.
- **timestamps** (*list*) – List of sorted timestamps of *world_trace*.
- **qsr_params** (*dict*) – QSR specific parameters passed in *dynamic_args*.
- **dynamic_args** (*dict*) – Dynamic arguments passed with the request.
- **kwargs** – kwargs arguments.

Returns Computed world QSR trace.

Return type *World_QSR_Trace*

unique_id
Getter for the unique identifier of a QSR.

Returns *self._unique_id*

Return type str

qsrlib_qsrs.qsr_arg_prob_relations_distance module

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

Probabilistic ard-distances.

Values of the abstract properties

- **_unique_id** = "argprobd"
- **_all_possible_relations** = depends on what user has passed
- **_dtype** = "points"

QSR Parameters (for *dynamic_args*)

- **'qsr_relations_and_values'**: A dictionary with keys being the relations labels and values

See also:

For further details, refer to its *description.*

allowed_value_types = None
tuple of datatypes: ?

value_sort_key = None
?

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*

Abstract class of argument relations.

all_possible_values = None
tuple: List of distance thresholds from *qsr_relations_and_values*, corresponding to the order of *self._all_possible_relations*.

allowed_value_types = None
tuple: Allowed types of the thresholds.

qsr_relations_and_values = None
dict: Holds the passed *qsr_relations_and_values* dict in *dynamic_args*.

value_sort_key = None
type depends on implementation: The method that the QSR labels are sorted based on their values.

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*

Argument distance relations.

Note: The relations are defined on the intervals of distance thresholds $[d_k, d_{k+1})$.

Values of the abstract properties

- **_unique_id** = "argd"
- **_all_possible_relations** = depends on what user has passed
- **_dtype** = "points"

QSR specific *dynamic_args*

- **'qsr_relations_and_values'**: A dictionary with keys being the relations labels and values the distance thresholds as an int or a float.

See also:

For further details, refer to its *description.*

allowed_value_types = None
tuple: distance thresholds can only be int or float

value_sort_key = None
operator.itemgetter: Sort keys/values by threshold value.

qsrlib_qsrs.qsr_cardinal_direction module

class qsrlib_qsrs.qsr_cardinal_direction.**QSR_Cardinal_Direction**
Bases: *qsrlib_qsrs.qsr_dyadic_abstractclass.QSR_Dyadic_1t_Abstractclass*

Cardinal direction relations.

Values of the abstract properties

- **_unique_id** = "cardir"
- **_all_possible_relations** = ("n", "ne", "e", "se", "s", "sw", "w", "nw", "eq")
- **_dtype** = "bounding_boxes_2d"

Some explanation about the QSR or better link to a separate webpage explaining it. Maybe a reference if it exists.

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** (*World_Trace*) – Input data.
- **timestamps** (*list*) – List of sorted timestamps of *world_trace*.
- **qsr_params** (*dict*) – QSR specific parameters passed in *dynamic_args*.
- **req_params** (*dict*) – Request parameters.
- **kwargs** – kwargs arguments.

Returns Computed world QSR trace.

Return type *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** (*World_Trace*) – Input data.
- **timestamps** (*list*) – List of sorted timestamps of *world_trace*.
- **qsr_params** (*dict*) – QSR specific parameters passed in *dynamic_args*.
- **req_params** (*dict*) – Request parameters.
- **kwargs** – kwargs arguments.

Returns Computed world QSR trace.

Return type *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.

Values of the abstract properties

- **_unique_id** = "mos"
- **_all_possible_relations** = ("m", "s")
- **_dtype** = "points"

QSR specific *dynamic_args*

- **'quantisation_factor'** (*float*) = 0.0: Threshold that determines minimal Euclidean distance to be considered as moving.

Some explanation about the QSR or better link to a separate webpage explaining it. Maybe a reference if it exists.

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.

Values of the abstract properties

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

See also:

For further details about MWE, refer to its *description.*

qsrlib_qsrs.qsr_qtc_b_simplified module

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

QTCB simplified relations.

Values of the abstract properties

- **_unique_id** = "qtcbs"
- **_all_possible_relations** = ?
- **_dtype** = "points"

Some explanation about the QSR or better link to a separate webpage explaining it. Maybe a reference if it exists.

qtc_to_output_format(*qtc*)
Return QTCBS specific from QTCCS.

Parameters **qtc** (*list or tuple*) – Full QTCC tuple [q1,q2,q4,q5].

Returns {"qtcbs": "q1,q2"}

Return type dict

qtc_type = None
?

qsrlib_qsrs.qsr_qtc_bc_simplified module

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

QTCBC simplified relations.

Values of the abstract properties

- **_unique_id** = "qtcbcs"
- **_all_possible_relations** = ?
- **_dtype** = "points"

Some explanation about the QSR or better link to a separate webpage explaining it. Maybe a reference if it exists.

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

Parameters

- **world_trace** (*World_Trace*) – Input data.
- **timestamps** (*list*) – List of sorted timestamps of *world_trace*.
- **qsr_params** (*dict*) – QSR specific parameters passed in *dynamic_args*.
- **req_params** – Dynamic arguments passed with the request.
- **kwargs** – kwargs arguments.

Returns Computed world QSR trace.

Return type *World_QSR_Trace*

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

Parameters **qtc** (*list or tuple*) – Full QTCC tuple [q1,q2,q4,q5].

Returns {"qtcbcs": "q1,q2,q4,q5"}

Return type dict

qtc_type = None
str: QTC specific type.

qsrlib_qsrs.qsr_qtc_c_simplified module

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

QTCB simplified relations.

Values of the abstract properties

- **_unique_id** = "qtccs"
- **_all_possible_relations** = ?
- **_dtype** = "points"

Some explanation about the QSR or better link to a separate webpage explaining it. Maybe a reference if it exists.

qtc_to_output_format(*qtc*)
Return QTCCS.

Parameters **qtc** (*list or tuple*) – Full QTCC tuple [q1,q2,q4,q5].

Returns {"qtccs": "q1,q2,q4,q5"}

Return type dict

qtc_type = None
str: QTC specific type.

qsrlib_qsrs.qsr_qtc_simplified_abstractclass module

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

QTCS abstract class.

create_qtc_string(*qtc*)

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

Parameters

- **world_trace** (*World_Trace*) – Input data.
- **timestamps** (*list*) – List of sorted timestamps of *world_trace*.
- **qsr_params** (*dict*) – QSR specific parameters passed in *dynamic_args*.
- **req_params** (*dict*) – Request parameters.

- **kwargs** – kwargs arguments.

Returns Computed world QSR trace.

Return type *World_QSR_Trace*

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

qtc_type = None
?

return_all_possible_state_combinations()
Return all possible state combinations for the qtc_type defined for this class instance.

Returns All possible state combinations.

Return type

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

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

?

qsrlib_qsrs.qsr_ra module

class qsrlib_qsrs.qsr_ra.**QSR_RA**
Bases: *qsrlib_qsrs.qsr_dyadic_abstractclass.QSR_Dyadic_1t_Abstractclass*

Rectangle Algebra.

Members:

- _unique_id: "ra"
- _all_possible_relations: ("<", ">", "m", "mi", "o", "oi", "s", "si", "d", "di", "f", "fi", "=")
- _dtype: "bounding_boxes"

See also:

For further details about RA, refer to its *description.*

qsrlib_qsrs.qsr_rcc2 module

class qsrlib_qsrs.qsr_rcc2.**QSR_RCC2**
Bases: *qsrlib_qsrs.qsr_rcc_abstractclass.QSR_RCC_Abstractclass*

Symmetrical RCC2 relations.

Values of the abstract properties

- **_unique_id** = "rcc2"
- **_all_possible_relations** = ("dc", "c")

- **_dtype** = "bounding_boxes_2d"

QSR specific *dynamic_args*

- **'quantisation_factor'** (*float*) = 0.0: Threshold that determines whether two rectangle regions are disconnected.

See also:

For further details about RCC2, refer to its *description.*

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*

Symmetrical RCC5 relations.

> **Warning:** RCC3 relations symbols are under consideration and might change in the near future.

Values of the abstract properties

- **_unique_id** = "rcc3"
- **_all_possible_relations** = ("dc", "po", "o")
- **_dtype** = "bounding_boxes_2d"

QSR specific *dynamic_args*

- **'quantisation_factor'** (*float*) = 0.0: Threshold that determines whether two rectangle regions are disconnected.

See also:

For further details about RCC3, refer to its description.

qsrlib_qsrs.qsr_rcc4 module

class qsrlib_qsrs.qsr_rcc4.**QSR_RCC4**
Bases: *qsrlib_qsrs.qsr_rcc_abstractclass.QSR_RCC_Abstractclass*

Symmetrical RCC4 relations.

Values of the abstract properties

- **_unique_id** = "rcc4"
- **_all_possible_relations** = ("dc", "po", "pp", "ppi")
- **_dtype** = "bounding_boxes_2d"

QSR specific *dynamic_args*

- **'quantisation_factor'** (*float*) = 0.0: Threshold that determines whether two rectangle regions are disconnected.

See also:

For further details about RCC4, refer to its *description.*

qsrlib_qsrs.qsr_rcc5 module

class qsrlib_qsrs.qsr_rcc5.**QSR_RCC5**
Bases: *qsrlib_qsrs.qsr_rcc_abstractclass.QSR_RCC_Abstractclass*

Symmetrical RCC5 relations.

Values of the abstract properties

- **_unique_id** = "rcc5"
- **_all_possible_relations** = ("dr", "po", "pp", "ppi", "eq")
- **_dtype** = "bounding_boxes_2d"

QSR specific *dynamic_args*

- **'quantisation_factor'** (*float*) = 0.0: Threshold that determines whether two rectangle regions are disconnected.

See also:

For further details about RCC5, refer to its *description.*

qsrlib_qsrs.qsr_rcc8 module

class qsrlib_qsrs.qsr_rcc8.**QSR_RCC8**
Bases: *qsrlib_qsrs.qsr_rcc_abstractclass.QSR_RCC_Abstractclass*

Symmetrical RCC5 relations.

Values of the abstract properties

- **_unique_id** = "rcc8"
- **_all_possible_relations** = ("dc", "ec", "po", "eq", "tpp", "ntpp", "tppi", "ntppi")
- **_dtype** = "bounding_boxes_2d"

QSR specific *dynamic_args*

- **'quantisation_factor'** (*float*) = 0.0: Threshold that determines whether two rectangle regions are disconnected.

See also:

For further details about RCC8, refer to its *description.*

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 of RCC relations.

Values of the abstract properties

- **_unique_id** = defined by the RCC variant.
- **_all_possible_relations** = defined by the RCC variant.
- **_dtype** = "bounding_boxes_2d"

QSR specific *dynamic_args*

- **quantisation_factor** (*float*) = 0.0: Threshold that determines whether two rectangle regions are disconnected.

qsrlib_qsrs.qsr_tpcc module

class qsrlib_qsrs.qsr_tpcc.**QSR_TPCC**
Bases: *qsrlib_qsrs.qsr_triadic_abstractclass.QSR_Triadic_1t_Abstractclass*

TPCC QSRs.

See also:

For further details about TPCC, refer to its *description.*

qsrlib_qsrs.qsr_triadic_abstractclass module

class qsrlib_qsrs.qsr_triadic_abstractclass.**QSR_Triadic_1t_Abstractclass**
Bases: *qsrlib_qsrs.qsr_triadic_abstractclass.QSR_Triadic_Abstractclass*

Special case abstract class of triadic QSRs. Works with triadic 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** (*World_Trace*) – Input data.
- **timestamps** (*list*) – List of sorted timestamps of *world_trace*.
- **qsr_params** (*dict*) – QSR specific parameters passed in *dynamic_args*.
- **req_params** (*dict*) – Request parameters.
- **kwargs** – kwargs arguments.

Returns Computed world QSR trace.

Return type *World_QSR_Trace*

class qsrlib_qsrs.qsr_triadic_abstractclass.**QSR_Triadic_Abstractclass**
Bases: *qsrlib_qsrs.qsr_abstractclass.QSR_Abstractclass*

Abstract class of triadic QSRs, i.e. QSRs that are computed over three objects.

Module contents

qsrlib_qstag package

Submodules

qsrlib_qstag.qsr_episodes module

qsrlib_qstag.qstag module

Qualitative Spatio-Temporal Activity Graph module

class qsrlib_qstag.qstag.**Activity_Graph**(*world*, *world_qsr*, *object_types={}*, *params={}*)

Activity Graph class: Lower level is a set of only nodes of type 'object'. Middle level nodes are only of type 'spatial_relation'. Top level nodes are only of type 'temporal_relation'.

Accepts a QSRLib.World_Trace and QSRLib.QSR_World_Trace as input.

abstract_graph

Getter.

Returns *self.abstract_graph*

Return type igraph.Graph

abstract_object_nodes

Getter.

Returns *object_nodes* from abstract_graph

Return type list

episodes

Getter.

Returns *self.__episodes*

Return type list

static get_objects_types(*objects_types*, *world*)

Generates a dictionary of object name and object type pairs Using both the dynamic_args dictionary where key = *objects_types*, and the **kwargs value [object_type] in the World Trace object

Parameters

- **objects_types** (*dictionary*) – Uses the dynamic_args dictionary where key = *objects_types* if provided
- **world** (*World_Trace*) – Otherwise, looks at the **kwargs value [object_type] in the World Trace object

Returns A dictionary with the object name as keys and the generic object type as value.

Return type dict

graphlets = None

Creates a Graphlets object containing lists of, unique graphlets, hashes and histogram of graphlets.

histogram

Getter.

Returns *self.__histogram*

Return type dict

object_nodes

Getter.

Returns *object_nodes*

Return type list

spatial_nodes

Getter.

Returns *spatial_nodes*

Return type list

spatial_obj_edges
Getter.

Returns *self.__spatial_obj_edges*

Return type list

temp_spatial_edges
Getter.

Returns *self.__temp_spatial_edges*

Return type list

temporal_nodes
Getter.

Returns *temporal_nodes*

Return type list

class qsrlib_qstag.qstag.**Graphlets**(*episodes*, *params*, *object_types*)
Graphlet class: Minimal subgraphs of the same structure as the Activity Graph.

code_book = None
list: The list of graphlet hashes (zip with histogram for count of each).

graphlets = None
dict: dictionary of the graphlet hash as key, and the iGraph object as value.

histogram = None
list: The list of graphlet counts (zip with codebook for a hash of each, and check graphlets for the iGraph).

qsrlib_qstag.qstag.**get_graph**(*episodes*, *object_types={}*)
Generates a graph from a set of input episode QSRs.

Parameters

- **episodes** (*list*) – list of episodes, where one episode = [[obj_list], {QSR dict}, (start, end_tuple)]
- **object_types** (*dict*) – a dictionary of object ID and object type.

Returns igraph.Graph: An igraph graph object containing all the object, spatial and temporal nodes.

Return type igraph.Graph

qsrlib_qstag.qstag.**get_graphlet_selections**(*episodes*, *params*, *object_types*, *vis=False*)
This function implements Sridar's validity criteria to select all valid graphlets from an activity graph: see Sridar_AAAI_2010 for more details.

Parameters

- **episodes** (*list*) – list of episodes, where one episode = [[obj_list], {QSR dict}, (start, end_tuple)]
- **params** (*dict*) – a dictionary containing parameters for the generation of the QSTAG. i.e. "min_rows", "max_rows", "max_eps"

Return list_of_graphlets a list of iGraph graphlet objects

Return type list

Return list_of_graphlet_hashes a list of hashes, relating to the graphlets

Return type list

Module contents

qsrlib_ros package

Submodules

qsrlib_ros.qsrlib_ros_client module

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

ROS client of QSRlib.

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

Parameters **qsrlib_request_message** (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*) – Request message.

Returns ROS service response.

Return type qsr_lib.srv.RequestQSRsResponse

service_topic_names = None
dict: Topic names of the services.

Module contents

qsrlib_utils package

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 List of pairs as tuples.

Return type list of tuples of str

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 List of pairs as tuples.

Return type list of tuples of str

qsrlib_utils.combinations_and_permutations.**possible_triplets**(*s*, *mirrors=True*)

Return the possible triplets from the list s.

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*) – Python format datetime.

Returns ROS time.

Return type rospy.Time

qsrlib_utils.utils module

Module contents

CHAPTER 9

Indices and tables

- genindex
- modindex

Python Module Index

q

Index

H

K

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z