igraph library

API Documentation

October 8, 2020

Contents

Contents **1**

1 Package igraph **2**
1.1 Modules . . . 2
1.2 Functions . . . 3
1.3 Variables . . . 5
1.4 Class Vertex . . . 5
1.4.1 Methods . . . 6
1.4.2 Properties . . . 11
1.5 Class Graph . . . 11
1.5.1 Methods . . . 13
1.5.2 Properties . . . 90
1.5.3 Class Variables . . . 90
1.6 Class VertexSeq . . . 91
1.6.1 Methods . . . 92
1.6.2 Properties . . . 98
1.7 Class EdgeSeq . . . 98
1.7.1 Methods . . . 100
1.7.2 Properties . . . 103
1.8 Class ARPACKOptions . . . 103
1.8.1 Methods . . . 104
1.8.2 Properties . . . 104
1.9 Class BFSIter . . . 105
1.9.1 Methods . . . 105
1.9.2 Properties . . . 105
1.10 Class DFSIter . . . 105
1.10.1 Methods . . . 105
1.10.2 Properties . . . 106
1.11 Class Edge . . . 106
1.11.1 Methods . . . 106
1.11.2 Properties . . . 109
1.12 Class GraphBase . . . 109
1.12.1 Methods . . . 110
1.12.2 Properties . . . 225
1.12.3 Class Variables . . . 225

2 Module igraph._igraph **226**

2.1 Functions . . . 226
2.2 Variables . . . 228
2.3 Class InternalError . . . 229
2.3.1 Methods . . . 229
2.3.2 Properties . . . 229

3 Package igraph.app **231**
3.1 Modules . . . 231
3.2 Variables . . . 231

4 Module igraph.app.shell **232**
4.1 Functions . . . 232
4.2 Variables . . . 232
4.3 Class TerminalController . . . 232
4.3.1 Methods . . . 233
4.3.2 Class Variables . . . 233
4.4 Class ProgressBar . . . 235
4.4.1 Methods . . . 235
4.4.2 Class Variables . . . 235
4.4.3 Instance Variables . . . 235
4.5 Class Shell . . . 236
4.5.1 Methods . . . 236
4.5.2 Properties . . . 236
4.6 Class IDLEShell . . . 237
4.6.1 Methods . . . 237
4.6.2 Properties . . . 237
4.7 Class ConsoleProgressBarMixin . . . 238
4.7.1 Methods . . . 238
4.7.2 Properties . . . 238
4.8 Class IPythonShell . . . 238
4.8.1 Methods . . . 239
4.8.2 Properties . . . 239
4.9 Class ClassicPythonShell . . . 239
4.9.1 Methods . . . 240
4.9.2 Properties . . . 240

5 Module igraph.clustering **241**
5.1 Functions . . . 243
5.2 Variables . . . 245
5.3 Class Clustering . . . 245
5.3.1 Methods . . . 246
5.3.2 Properties . . . 248
5.4 Class VertexClustering . . . 248
5.4.1 Methods . . . 249
5.4.2 Properties . . . 254
5.5 Class Dendrogram . . . 254
5.5.1 Methods . . . 255
5.5.2 Properties . . . 256
5.6 Class VertexDendrogram . . . 257
5.6.1 Methods . . . 257
5.6.2 Properties . . . 258
5.7 Class Cover . . . 259

5.7.1 Methods . . . 260
5.7.2 Properties . . . 261
5.8 Class VertexCover . . . 262
5.8.1 Methods . . . 262
5.8.2 Properties . . . 265
5.9 Class CohesiveBlocks . . . 265
5.9.1 Methods . . . 266
5.9.2 Properties . . . 267

6 Module igraph.configuration 269
6.1 Functions . . . 269
6.2 Variables . . . 269
6.3 Class Configuration . . . 270
6.3.1 Methods . . . 272
6.3.2 Properties . . . 273

7 Module igraph.cut 274
7.1 Variables . . . 274
7.2 Class Cut . . . 274
7.2.1 Methods . . . 276
7.2.2 Properties . . . 277
7.3 Class Flow . . . 277
7.3.1 Methods . . . 279
7.3.2 Properties . . . 280

8 Module igraph.datatypes 281
8.1 Variables . . . 281
8.2 Class Matrix . . . 281
8.2.1 Methods . . . 281
8.2.2 Properties . . . 286
8.3 Class DyadCensus . . . 286
8.3.1 Methods . . . 287
8.3.2 Properties . . . 288
8.4 Class TriadCensus . . . 288
8.4.1 Methods . . . 289
8.4.2 Properties . . . 289
8.5 Class UniqueIdGenerator . . . 290
8.5.1 Methods . . . 290
8.5.2 Properties . . . 291

9 Package igraph.drawing 292
9.1 Modules . . . 292
9.2 Functions . . . 294
9.3 Class DefaultGraphDrawer . . . 295
9.3.1 Methods . . . 296
9.3.2 Properties . . . 297
9.4 Class BoundingBox . . . 297
9.4.1 Methods . . . 298
9.4.2 Properties . . . 298
9.5 Class Point . . . 299
9.5.1 Methods . . . 299
9.5.2 Properties . . . 301

9.6 Class Rectangle . . . 301
9.6.1 Methods . . . 302
9.6.2 Properties . . . 306
9.7 Class Plot . . . 307
9.7.1 Methods . . . 309
9.7.2 Properties . . . 311

10 Module igraph.drawing.baseclasses 312
10.1 Variables . . . 312
10.2 Class AbstractDrawer . . . 312
10.2.1 Methods . . . 312
10.2.2 Properties . . . 312
10.3 Class AbstractCairoDrawer . . . 312
10.3.1 Methods . . . 313
10.3.2 Properties . . . 313
10.4 Class AbstractXMLRPCDrawer . . . 314
10.4.1 Methods . . . 314
10.4.2 Properties . . . 314

11 Module igraph.drawing.colors 315
11.1 Functions . . . 315
11.2 Variables . . . 318
11.3 Class Palette . . . 318
11.3.1 Methods . . . 318
11.3.2 Properties . . . 321
11.4 Class GradientPalette . . . 321
11.4.1 Methods . . . 322
11.4.2 Properties . . . 322
11.5 Class AdvancedGradientPalette . . . 322
11.5.1 Methods . . . 323
11.5.2 Properties . . . 323
11.6 Class RainbowPalette . . . 323
11.6.1 Methods . . . 324
11.6.2 Properties . . . 325
11.7 Class PrecalculatedPalette . . . 325
11.7.1 Methods . . . 325
11.7.2 Properties . . . 325
11.8 Class ClusterColoringPalette . . . 326
11.8.1 Methods . . . 326
11.8.2 Properties . . . 327

12 Module igraph.drawing.coord 328
12.1 Variables . . . 328
12.2 Class CoordinateSystem . . . 328
12.2.1 Methods . . . 328
12.2.2 Properties . . . 329
12.3 Class DescartesCoordinateSystem . . . 329
12.3.1 Methods . . . 329
12.3.2 Properties . . . 330

13 Module igraph.drawing.edge 331
13.1 Class AbstractEdgeDrawer . . . 331

13.1.1 Methods . . . 331
13.1.2 Properties . . . 333
13.2 Class ArrowEdgeDrawer . . . 333
13.2.1 Methods . . . 334
13.2.2 Properties . . . 334
13.3 Class TaperedEdgeDrawer . . . 334
13.3.1 Methods . . . 335
13.3.2 Properties . . . 335
13.4 Class AlphaVaryingEdgeDrawer . . . 335
13.4.1 Methods . . . 336
13.4.2 Properties . . . 336
13.5 Class LightToDarkEdgeDrawer . . . 337
13.5.1 Methods . . . 337
13.5.2 Properties . . . 337
13.6 Class DarkToLightEdgeDrawer . . . 338
13.6.1 Methods . . . 338
13.6.2 Properties . . . 338

14 Module igraph.drawing.graph 339
14.1 Class DefaultGraphDrawer . . . 339
14.1.1 Methods . . . 340
14.1.2 Properties . . . 341
14.2 Class UbiGraphDrawer . . . 341
14.2.1 Methods . . . 342
14.2.2 Properties . . . 342
14.3 Class CytoscapeGraphDrawer . . . 343
14.3.1 Methods . . . 344
14.3.2 Properties . . . 345

15 Module igraph.drawing.metamagic 346
15.1 Class AttributeSpecification . . . 347
15.1.1 Methods . . . 347
15.1.2 Properties . . . 347
15.2 Class AttributeCollectorBase . . . 348
15.2.1 Methods . . . 348
15.2.2 Properties . . . 348

16 Module igraph.drawing.shapes 349
16.1 Class ShapeDrawerDirectory . . . 349
16.1.1 Methods . . . 349
16.1.2 Properties . . . 350
16.1.3 Class Variables . . . 350

17 Module igraph.drawing.text 352
17.1 Class TextAlignment . . . 352
17.1.1 Methods . . . 352
17.1.2 Properties . . . 352
17.1.3 Class Variables . . . 352
17.2 Class TextDrawer . . . 352
17.2.1 Methods . . . 353
17.2.2 Properties . . . 355
17.2.3 Class Variables . . . 356

18 Module igraph.drawing.utils **357**
18.1 Class Rectangle . . . 357
18.1.1 Methods . . . 357
18.1.2 Properties . . . 362
18.2 Class BoundingBox . . . 362
18.2.1 Methods . . . 363
18.2.2 Properties . . . 363
18.3 Class FakeModule . . . 364
18.3.1 Methods . . . 364
18.3.2 Properties . . . 364
18.4 Class Point . . . 364
18.4.1 Methods . . . 365
18.4.2 Properties . . . 367

19 Module igraph.drawing.vertex **368**
19.1 Class AbstractVertexDrawer . . . 368
19.1.1 Methods . . . 368
19.1.2 Properties . . . 369
19.2 Class AbstractCairoVertexDrawer . . . 369
19.2.1 Methods . . . 370
19.2.2 Properties . . . 370
19.3 Class DefaultVertexDrawer . . . 371
19.3.1 Methods . . . 371
19.3.2 Properties . . . 372

20 Module igraph.layout **373**
20.1 Variables . . . 373
20.2 Class Layout . . . 373
20.2.1 Methods . . . 374
20.2.2 Properties . . . 379

21 Module igraph.matching **380**
21.1 Variables . . . 380
21.2 Class Matching . . . 380
21.2.1 Methods . . . 381
21.2.2 Properties . . . 382

22 Module igraph.operators **383**
22.1 Functions . . . 383

23 Package igraph.remote **386**
23.1 Modules . . . 386
23.2 Variables . . . 386

24 Module igraph.remote.gephi **387**
24.1 Class GephiConnection . . . 387
24.1.1 Methods . . . 387
24.1.2 Properties . . . 388
24.2 Class GephiGraphStreamingAPIFormat . . . 388
24.2.1 Methods . . . 389
24.2.2 Properties . . . 391
24.3 Class GephiGraphStreamer . . . 391

24.3.1 Methods . . . 392
24.3.2 Properties . . . 393

25 Module igraph.statistics **394**
25.1 Functions . . . 394
25.2 Class FittedPowerLaw . . . 397
25.2.1 Methods . . . 398
25.2.2 Properties . . . 399
25.3 Class Histogram . . . 399
25.3.1 Methods . . . 399
25.3.2 Properties . . . 401
25.4 Class RunningMean . . . 401
25.4.1 Methods . . . 402
25.4.2 Properties . . . 404

26 Module igraph.summary' **405**
26.1 Class GraphSummary . . . 405
26.1.1 Methods . . . 406
26.1.2 Properties . . . 407

27 Module igraph.utils **408**
27.1 Functions . . . 408
27.2 Variables . . . 410
27.3 Class multidict . . . 411
27.3.1 Methods . . . 411
27.3.2 Properties . . . 414
27.3.3 Class Variables . . . 414

28 Module igraph.version **415**
28.1 Variables . . . 415

1 Package igraph

IGraph library.

Version: 0.8.3

1.1 Modules

- **_igraph**: Low-level Python interface for the igraph library.
  (Section 2, p. 226)
- **app**: User interfaces of igraph
  (Section 3, p. 231)
  - **shell**: Command-line user interface of igraph
    (Section 4, p. 232)
- **clustering**: Classes related to graph clustering.
  (Section 5, p. 241)
- **configuration**: Configuration framework for igraph.
  (Section 6, p. 269)
- **cut**: Classes representing cuts and flows on graphs.
  (Section 7, p. 274)
- **datatypes**: Additional auxiliary data types
  (Section 8, p. 281)
- **drawing**: Drawing and plotting routines for IGraph.
  (Section 9, p. 292)
  - **baseclasses**: Abstract base classes for the drawing routines.
    (Section 10, p. 312)
  - **colors**: Color handling functions.
    (Section 11, p. 315)
  - **coord**: Coordinate systems and related plotting routines
    (Section 12, p. 328)
  - **edge**: Drawers for various edge styles in graph plots.
    (Section 13, p. 331)
  - **graph**: Drawing routines to draw graphs.
    (Section 14, p. 339)
  - **metamagic**: Auxiliary classes for the default graph drawer in igraph.
    (Section 15, p. 346)
  - **shapes**: Shape drawing classes for igraph
    (Section 16, p. 349)
  - **text**: Drawers for labels on plots.

(Section 17, p. 352)
  - **utils**: Utility classes for drawing routines.
    (Section 18, p. 357)
  - **vertex**: Drawing routines to draw the vertices of graphs.
    (Section 19, p. 368)
- **formula** *(Section ??, p. ??)*
- **layout**: Layout-related code in the IGraph library.
  (Section 20, p. 373)
- **matching**: Classes representing matchings on graphs.
  (Section 21, p. 380)
- **operators**: Implementation of union, disjoint union and intersection operators.
  (Section 22, p. 383)
- **remote**: Classes that help igraph communicate with remote applications.
  (Section 23, p. 386)
  - **gephi**: Classes that help igraph communicate with Gephi (`http://www.gephi.org`).
    (Section 24, p. 387)
- **statistics**: Statistics related stuff in igraph
  (Section 25, p. 394)
- **summary** *(Section 1.2, p. 4)*
- **summary'**: Summary representation of a graph.
  (Section 26, p. 405)
- **utils**: Utility functions that cannot be categorised anywhere else.
  (Section 27, p. 408)
- **version** *(Section 28, p. 415)*

1.2 Functions

autocurve(*graph*, *attribute*=`'curved'`, *default*=`0`)

Calculates curvature values for each of the edges in the graph to make sure that multiple edges are shown properly on a graph plot.

This function checks the multiplicity of each edge in the graph and assigns curvature values (numbers between -1 and 1, corresponding to CCW (-1), straight (0) and CW (1) curved edges) to them. The assigned values are either stored in an edge attribute or returned as a list, depending on the value of the *attribute* argument.

Parameters

`graph`: the graph on which the calculation will be run

`attribute`: the name of the edge attribute to save the curvature values to. The default value is `curved`, which is the name of the edge attribute the default graph plotter checks to decide whether an edge should be curved on the plot or not. If *attribute* is `None`, the result will not be stored.

`default`: the default curvature for single edges. Zero means that single edges will be straight. If you want single edges to be curved as well, try passing 0.5 or -0.5 here.

Return Value

the list of curvature values if *attribute* is `None`, otherwise `None`.

get_include()

Returns the folder that contains the C API headers of the Python interface of igraph.

read(*filename*, **args*, ***kwds*)

Loads a graph from the given filename.

This is just a convenience function, calls `Graph.Read` directly. All arguments are passed unchanged to `Graph.Read`

Parameters

`filename:` the name of the file to be loaded

load(*filename*, **args*, ***kwds*)

Loads a graph from the given filename.

This is just a convenience function, calls `Graph.Read` directly. All arguments are passed unchanged to `Graph.Read`

Parameters

`filename:` the name of the file to be loaded

write(*graph*, *filename*, **args*, ***kwds*)

Saves a graph to the given file.

This is just a convenience function, calls `Graph.write` directly. All arguments are passed unchanged to `Graph.write`

Parameters

`graph:` the graph to be saved

`filename:` the name of the file to be written

save(*graph*, *filename*, **args*, ***kwds*)

Saves a graph to the given file.

This is just a convenience function, calls `Graph.write` directly. All arguments are passed unchanged to `Graph.write`

Parameters

`graph:` the graph to be saved

`filename:` the name of the file to be written

summary(*obj*, *stream*=`None`, **args*, ***kwds*)

Prints a summary of object o to a given stream

Positional and keyword arguments not explicitly mentioned here are passed on to the underlying `summary()` method of the object if it has any.

Parameters

`obj:` the object about which a human-readable summary is requested.

`stream:` the stream to be used. If `None`, the standard output will be used.

1.3 Variables

Name	Description
config	**Value:** `None`
ADJ_DIRECTED	**Value:** `0`
ADJ_LOWER	**Value:** `3`
ADJ_MAX	**Value:** `1`
ADJ_MIN	**Value:** `4`
ADJ_PLUS	**Value:** `5`
ADJ_UNDIRECTED	**Value:** `1`
ADJ_UPPER	**Value:** `2`
ALL	**Value:** `3`
BLISS_F	**Value:** `0`
BLISS_FL	**Value:** `1`
BLISS_FLM	**Value:** `4`
BLISS_FM	**Value:** `3`
BLISS_FS	**Value:** `2`
BLISS_FSM	**Value:** `5`
GET_ADJACENCY_BOTH	**Value:** `2`
GET_ADJACENCY_LOWER	**Value:** `1`
GET_ADJACENCY_UPPER	**Value:** `0`
IN	**Value:** `2`
OUT	**Value:** `1`
REWIRING_SIMPLE	**Value:** `0`
REWIRING_SIMPLE_LOOPS	**Value:** `1`
STAR_IN	**Value:** `1`
STAR_MUTUAL	**Value:** `3`
STAR_OUT	**Value:** `0`
STAR_UNDIRECTED	**Value:** `2`
STRONG	**Value:** `2`
TRANSITIVITY_NAN	**Value:** `0`
TRANSITIVITY_ZERO	**Value:** `1`
TREE_IN	**Value:** `1`
TREE_OUT	**Value:** `0`
TREE_UNDIRECTED	**Value:** `2`
WEAK	**Value:** `1`
__package__	**Value:** `'igraph'`
arpack_options	**Value:** `<igraph.ARPACKOptions object at 0x10c48ab90>`
name	**Value:** `'write_svg'`

1.4 Class Vertex

object
└ **igraph.Vertex**

Class representing a single vertex in a graph.

The vertex is referenced by its index, so if the underlying graph changes, the semantics of the vertex object

might change as well (if the vertex indices are altered in the original graph).

The attributes of the vertex can be accessed by using the vertex as a hash:

```
>>> v["color"] = "red"                    #doctest: +SKIP
>>> print v["color"]                        #doctest: +SKIP
red
```

1.4.1 Methods

__delitem__(*x*, *y*)

del x[y]

__eq__(*x*, *y*)

x==y

__ge__(*x*, *y*)

x>=y

__getitem__(*x*, *y*)

x[y]

__gt__(*x*, *y*)

x>y

__hash__(*x*)

hash(x)

Overrides: object.__hash__

__le__(*x*, *y*)

x<=y

__len__(*x*)

len(x)

__lt__(*x*, *y*)

x<y

__ne__(*x*, *y*)

x!=y

__repr__(*x*)

repr(x)

Overrides: object.__repr__

__setitem__(*x*, *i*, *y*)

x[i]=y

all_edges(...)

Proxy method to `Graph.incident(..., mode="all")`

This method calls the incident() method of the `Graph` class with this vertex as the first argument and "all" as the mode argument, and returns the result.

See Also: Graph.incident() for details.

attribute_names()

Returns the list of vertex attribute names

Return Value

list

attributes()

Returns a dict of attribute names and values for the vertex

Return Value

dict

betweenness(...)

Proxy method to `Graph.betweenness()`

This method calls the betweenness method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.betweenness() for details.

closeness(...)

Proxy method to `Graph.closeness()`

This method calls the closeness method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.closeness() for details.

constraint(...)

Proxy method to `Graph.constraint()`

This method calls the constraint method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.constraint() for details.

degree(...)

Proxy method to `Graph.degree()`

This method calls the degree method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.degree() for details.

delete(...)

Proxy method to `Graph.delete_vertices()`

This method calls the delete_vertices method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.delete_vertices() for details.

diversity(...)

Proxy method to `Graph.diversity()`

This method calls the diversity method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.diversity() for details.

eccentricity(...)

Proxy method to `Graph.eccentricity()`

This method calls the eccentricity method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.eccentricity() for details.

get_shortest_paths(...)

Proxy method to `Graph.get_shortest_paths()`

This method calls the get_shortest_paths method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.get_shortest_paths() for details.

in_edges(...)

Proxy method to `Graph.incident(..., mode="in")`

This method calls the incident() method of the `Graph` class with this vertex as the first argument and "in" as the mode argument, and returns the result.

See Also: Graph.incident() for details.

incident(...)

Proxy method to `Graph.incident()`

This method calls the incident method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.incident() for details.

indegree(...)

Proxy method to `Graph.indegree()`

This method calls the indegree method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.indegree() for details.

is_minimal_separator(...)

Proxy method to `Graph.is_minimal_separator()`

This method calls the is_minimal_separator method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.is_minimal_separator() for details.

is_separator(...)

Proxy method to `Graph.is_separator()`

This method calls the is_separator method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.is_separator() for details.

neighbors(...)

Proxy method to `Graph.neighbors()`

This method calls the neighbors method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.neighbors() for details.

out_edges(...)

Proxy method to `Graph.incident(..., mode="out")`

This method calls the incident() method of the `Graph` class with this vertex as the first argument and "out" as the mode argument, and returns the result.

See Also: Graph.incident() for details.

outdegree(...)

Proxy method to `Graph.outdegree()`

This method calls the outdegree method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.outdegree() for details.

pagerank(...)

Proxy method to `Graph.pagerank()`

This method calls the pagerank method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.pagerank() for details.

personalized_pagerank(...)

Proxy method to `Graph.personalized_pagerank()`

This method calls the personalized_pagerank method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.personalized_pagerank() for details.

predecessors(...)

Proxy method to `Graph.predecessors()`

This method calls the predecessors method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.predecessors() for details.

shortest_paths(...)

Proxy method to `Graph.shortest_paths()`

This method calls the shortest_paths method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.shortest_paths() for details.

strength(...)

Proxy method to `Graph.strength()`

This method calls the strength method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.strength() for details.

successors(...)

Proxy method to `Graph.successors()`

This method calls the successors method of the `Graph` class with this vertex as the first argument, and returns the result.

See Also: Graph.successors() for details.

update_attributes(*E*, ***F*)

Updates the attributes of the vertex from dict/iterable E and F.

If E has a `keys()` method, it does: `for k in E: self[k] = E[k]`. If E lacks a `keys()` method, it does: `for (k, v) in E: self[k] = v`. In either case, this is followed by: `for k in F: self[k] = F[k]`.

This method thus behaves similarly to the `update()` method of Python dictionaries.

Return Value

None

Inherited from object

__delattr__(), __format__(), __getattribute__(), __init__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

1.4.2 Properties

Name	Description
graph	The graph the vertex belongs to
index	Index of the vertex
Inherited from object __class__	

1.5 Class Graph

object

igraph.GraphBase

igraph.Graph

Generic graph.

This class is built on top of `GraphBase`, so the order of the methods in the Epydoc documentation is a little bit obscure: inherited methods come after the ones implemented directly in the subclass. `Graph` provides many functions that `GraphBase` does not, mostly because these functions are not speed critical and they were easier to implement in Python than in pure C. An example is the attribute handling in the constructor: the constructor of `Graph` accepts three dictionaries corresponding to the graph, vertex and edge attributes while the constructor of `GraphBase` does not. This extension was needed to make `Graph` serializable through the `pickle` module. `Graph` also overrides some functions from `GraphBase` to provide a more convenient interface; e.g., layout functions return a `Layout` instance from `Graph` instead of a list of coordinate pairs.

Graphs can also be indexed by strings or pairs of vertex indices or vertex names. When a graph is indexed by a string, the operation translates to the retrieval, creation, modification or deletion of a graph attribute:

```
>>> g = Graph.Full(3)
>>> g["name"] = "Triangle graph"
>>> g["name"]
'Triangle graph'
>>> del g["name"]
```

When a graph is indexed by a pair of vertex indices or names, the graph itself is treated as an adjacency matrix and the corresponding cell of the matrix is returned:

```
>>> g = Graph.Full(3)
>>> g.vs["name"] = ["A", "B", "C"]
>>> g[1, 2]
1
>>> g["A", "B"]
1
>>> g["A", "B"] = 0
>>> g.ecount()
2
```

Assigning values different from zero or one to the adjacency matrix will be translated to one, unless the graph is weighted, in which case the numbers will be treated as weights:

```
>>> g.is_weighted()
False
>>> g["A", "B"] = 2
>>> g["A", "B"]
1
>>> g.es["weight"] = 1.0
>>> g.is_weighted()
True
```

```
>>> g["A", "B"] = 2
>>> g["A", "B"]
2
>>> g.es["weight"]
[1.0, 1.0, 2]
```

1.5.1 Methods

omega()

Returns the clique number of the graph.

The clique number of the graph is the size of the largest clique.

See Also: `largest_cliques()` for the largest cliques.

alpha()

Returns the independence number of the graph.

The independence number of the graph is the size of the largest independent vertex set.

See Also: `largest_independent_vertex_sets()` for the largest independent vertex sets

shell_index(*mode*=`ALL`)

Finds the coreness (shell index) of the vertices of the network.

The k-core of a graph is a maximal subgraph in which each vertex has at least degree k. (Degree here means the degree in the subgraph of course). The coreness of a vertex is k if it is a member of the k-core but not a member of the $k+1$-core.

Parameters

`mode`: whether to compute the in-corenesses (`IN`), the out-corenesses (`OUT`) or the undirected corenesses (`ALL`). Ignored and assumed to be `ALL` for undirected graphs.

Return Value

the corenesses for each vertex.

Reference: Vladimir Batagelj, Matjaz Zaversnik: *An O(m) Algorithm for Core Decomposition of Networks.*

cut_vertices()

Returns the list of articulation points in the graph.

A vertex is an articulation point if its removal increases the number of connected components in the graph.

evcent(*directed*=`True`, *scale*=`True`, *weights*=`None`, *return_ eigenvalue*=`False`, *arpack_ options*=`None`)

Calculates the eigenvector centralities of the vertices in a graph.

Eigenvector centrality is a measure of the importance of a node in a network. It assigns relative scores to all nodes in the network based on the principle that connections from high-scoring nodes contribute more to the score of the node in question than equal connections from low-scoring nodes. In practice, the centralities are determined by calculating eigenvector corresponding to the largest positive eigenvalue of the adjacency matrix. In the undirected case, this function considers the diagonal entries of the adjacency matrix to be twice the number of self-loops on the corresponding vertex.

In the directed case, the left eigenvector of the adjacency matrix is calculated. In other words, the centrality of a vertex is proportional to the sum of centralities of vertices pointing to it.

Eigenvector centrality is meaningful only for connected graphs. Graphs that are not connected should be decomposed into connected components, and the eigenvector centrality calculated for each separately.

Parameters

`directed:`	whether to consider edge directions in a directed graph. Ignored for undirected graphs.
`scale:`	whether to normalize the centralities so the largest one will always be 1.
`weights:`	edge weights given as a list or an edge attribute. If `None`, all edges have equal weight.
`return_eigenvalue:`	whether to return the actual largest eigenvalue along with the centralities
`arpack_options:`	an `ARPACKOptions` object that can be used to fine-tune the calculation. If it is omitted, the module-level variable called `arpack_options` is used.

Return Value

the eigenvector centralities in a list and optionally the largest eigenvalue (as a second member of a tuple)

vertex_disjoint_paths(*source*=`-1`, *target*=`-1`, *checks*=`True`, *neighbors*=`"error"`)

Calculates the vertex connectivity of the graph or between some vertices.

The vertex connectivity between two given vertices is the number of vertices that have to be removed in order to disconnect the two vertices into two separate components. This is also the number of vertex disjoint directed paths between the vertices (apart from the source and target vertices of course). The vertex connectivity of the graph is the minimal vertex connectivity over all vertex pairs.

This method calculates the vertex connectivity of a given vertex pair if both the source and target vertices are given. If none of them is given (or they are both negative), the overall vertex connectivity is returned.

Parameters

`source`: the source vertex involved in the calculation.

`target`: the target vertex involved in the calculation.

`checks`: if the whole graph connectivity is calculated and this is `True`, igraph performs some basic checks before calculation. If the graph is not strongly connected, then the connectivity is obviously zero. If the minimum degree is one, then the connectivity is also one. These simple checks are much faster than checking the entire graph, therefore it is advised to set this to `True`. The parameter is ignored if the connectivity between two given vertices is computed.

`neighbors`: tells igraph what to do when the two vertices are connected. `"error"` raises an exception, `"infinity"` returns infinity, `"ignore"` ignores the edge.

Return Value

the vertex connectivity

edge_disjoint_paths(*source*=-1, *target*=-1, *checks*=True)

Calculates the edge connectivity of the graph or between some vertices.

The edge connectivity between two given vertices is the number of edges that have to be removed in order to disconnect the two vertices into two separate components. This is also the number of edge disjoint directed paths between the vertices. The edge connectivity of the graph is the minimal edge connectivity over all vertex pairs.

This method calculates the edge connectivity of a given vertex pair if both the source and target vertices are given. If none of them is given (or they are both negative), the overall edge connectivity is returned.

Parameters

`source`: the source vertex involved in the calculation.

`target`: the target vertex involved in the calculation.

`checks`: if the whole graph connectivity is calculated and this is `True`, igraph performs some basic checks before calculation. If the graph is not strongly connected, then the connectivity is obviously zero. If the minimum degree is one, then the connectivity is also one. These simple checks are much faster than checking the entire graph, therefore it is advised to set this to `True`. The parameter is ignored if the connectivity between two given vertices is computed.

Return Value

the edge connectivity

cohesion(*source*=`-1`, *target*=`-1`, *checks*=`True`, *neighbors*=`"error"`)

Calculates the vertex connectivity of the graph or between some vertices.

The vertex connectivity between two given vertices is the number of vertices that have to be removed in order to disconnect the two vertices into two separate components. This is also the number of vertex disjoint directed paths between the vertices (apart from the source and target vertices of course). The vertex connectivity of the graph is the minimal vertex connectivity over all vertex pairs.

This method calculates the vertex connectivity of a given vertex pair if both the source and target vertices are given. If none of them is given (or they are both negative), the overall vertex connectivity is returned.

Parameters

`source`: the source vertex involved in the calculation.

`target`: the target vertex involved in the calculation.

`checks`: if the whole graph connectivity is calculated and this is `True`, igraph performs some basic checks before calculation. If the graph is not strongly connected, then the connectivity is obviously zero. If the minimum degree is one, then the connectivity is also one. These simple checks are much faster than checking the entire graph, therefore it is advised to set this to `True`. The parameter is ignored if the connectivity between two given vertices is computed.

`neighbors`: tells igraph what to do when the two vertices are connected. `"error"` raises an exception, `"infinity"` returns infinity, `"ignore"` ignores the edge.

Return Value

the vertex connectivity

adhesion(*source*=`-1`, *target*=`-1`, *checks*=`True`)

Calculates the edge connectivity of the graph or between some vertices.

The edge connectivity between two given vertices is the number of edges that have to be removed in order to disconnect the two vertices into two separate components. This is also the number of edge disjoint directed paths between the vertices. The edge connectivity of the graph is the minimal edge connectivity over all vertex pairs.

This method calculates the edge connectivity of a given vertex pair if both the source and target vertices are given. If none of them is given (or they are both negative), the overall edge connectivity is returned.

Parameters

`source`: the source vertex involved in the calculation.

`target`: the target vertex involved in the calculation.

`checks`: if the whole graph connectivity is calculated and this is `True`, igraph performs some basic checks before calculation. If the graph is not strongly connected, then the connectivity is obviously zero. If the minimum degree is one, then the connectivity is also one. These simple checks are much faster than checking the entire graph, therefore it is advised to set this to `True`. The parameter is ignored if the connectivity between two given vertices is computed.

Return Value

the edge connectivity

shortest_paths_dijkstra(*source*=`None`, *target*=`None`, *weights*=`None`, *mode*=`OUT`)

Calculates shortest path lengths for given vertices in a graph.

The algorithm used for the calculations is selected automatically: a simple BFS is used for unweighted graphs, Dijkstra's algorithm is used when all the weights are positive. Otherwise, the Bellman-Ford algorithm is used if the number of requested source vertices is larger than 100 and Johnson's algorithm is used otherwise.

Parameters

`source`: a list containing the source vertex IDs which should be included in the result. If `None`, all vertices will be considered.

`target`: a list containing the target vertex IDs which should be included in the result. If `None`, all vertices will be considered.

`weights`: a list containing the edge weights. It can also be an attribute name (edge weights are retrieved from the given attribute) or `None` (all edges have equal weight).

`mode`: the type of shortest paths to be used for the calculation in directed graphs. `OUT` means only outgoing, `IN` means only incoming paths. `ALL` means to consider the directed graph as an undirected one.

Return Value

the shortest path lengths for given vertices in a matrix

subgraph(*vertices*, *implementation*=`"auto"`)

Returns a subgraph spanned by the given vertices.

Parameters

`vertices:` a list containing the vertex IDs which should be included in the result.

`implementation:` the implementation to use when constructing the new subgraph. igraph includes two implementations at the moment. `"copy_and_delete"` copies the original graph and removes those vertices that are not in the given set. This is more efficient if the size of the subgraph is comparable to the original graph. The other implementation (`"create_from_scratch"`) constructs the result graph from scratch and then copies the attributes accordingly. This is a better solution if the subgraph is relatively small, compared to the original graph. `"auto"` selects between the two implementations automatically, based on the ratio of the size of the subgraph and the size of the original graph.

Return Value

the subgraph

__init__(*n*=0, *edges*=None, *directed*=False, *graph_attrs*=None, *vertex_attrs*=None, *edge_attrs*=None)

Constructs a graph from scratch.

Parameters

n:	the number of vertices. Can be omitted, the default is zero. Note that if the edge list contains vertices with indexes larger than or equal to *m*, then the number of vertices will be adjusted accordingly.
edges:	the edge list where every list item is a pair of integers. If any of the integers is larger than *n-1*, the number of vertices is adjusted accordingly. None means no edges.
directed:	whether the graph should be directed
graph_attrs:	the attributes of the graph as a dictionary.
vertex_attrs:	the attributes of the vertices as a dictionary. Every dictionary value must be an iterable with exactly *n* items.
edge_attrs:	the attributes of the edges as a dictionary. Every dictionary value must be an iterable with exactly *m* items where *m* is the number of edges.

Overrides: object.__init__

add_edge(*source*, *target*, ***kwds*)

Adds a single edge to the graph.

Keyword arguments (except the source and target arguments) will be assigned to the edge as attributes.

Parameters

source: the source vertex of the edge or its name.

target: the target vertex of the edge or its name.

Return Value

the newly added edge as an Edge object. Use add_edges([(source, target)]) if you don't need the Edge object and want to avoid the overhead of creating t.

add_edges(*es*)

Adds some edges to the graph.

Parameters

`es`: the list of edges to be added. Every edge is represented with a tuple containing the vertex IDs or names of the two endpoints. Vertices are enumerated from zero.

Overrides: igraph.GraphBase.add_edges

add_vertex(*name*=`None`, ***kwds*)

Adds a single vertex to the graph. Keyword arguments will be assigned as vertex attributes. Note that `name` as a keyword argument is treated specially; if a graph has `name` as a vertex attribute, it allows one to refer to vertices by their names in most places where igraph expects a vertex ID.

Return Value

the newly added vertex as a `Vertex` object. Use `add_vertices(1)` if you don't need the `Vertex` object and want to avoid the overhead of creating t.

add_vertices(*n*)

```
Adds some vertices to the graph.

@param n: the number of vertices to be added, or the name of a single
  vertex to be added, or a sequence of strings, each corresponding to the
  name of a vertex to be added. Names will be assigned to the C{name}
  vertex attribute.
@params attributes: dict of sequences, all of length equal to the
  number of vertices to be added, containing the attributes of the new
  vertices. If n is a string (so a single vertex is added), then the
  values of this dict are the attributes themselves, but if n=1 then
  they have to be lists of length 1.

Note that if n is a sequence of strings, indicating the names of the
new vertices, and attributes has a key 'name', the two conflict. In
that case the attribute will be applied.
```

Parameters

`n`: the number of vertices to be added

Overrides: igraph.GraphBase.add_vertices

adjacent(*vertex*, *mode*=`OUT`)

Returns the edges a given vertex is incident on.

Deprecated: replaced by `Graph.incident()` since igraph 0.6

as_directed(**args*, ***kwds*)

Returns a directed copy of this graph. Arguments are passed on to `Graph.to_directed()` that is invoked on the copy.

as_undirected(**args*, ***kwds*)

Returns an undirected copy of this graph. Arguments are passed on to `Graph.to_undirected()` that is invoked on the copy.

delete_edges(*self*, **args*, ***kwds*)

Deletes some edges from the graph.

The set of edges to be deleted is determined by the positional and keyword arguments. If the function is called without any arguments, all edges are deleted. If any keyword argument is present, or the first positional argument is callable, an edge sequence is derived by calling `EdgeSeq.select` with the same positional and keyword arguments. Edges in the derived edge sequence will be removed. Otherwise the first positional argument is considered as follows:

- `None` - deletes all edges (deprecated since 0.8.3)
- a single integer - deletes the edge with the given ID
- a list of integers - deletes the edges denoted by the given IDs
- a list of 2-tuples - deletes the edges denoted by the given source-target vertex pairs. When multiple edges are present between a given source-target vertex pair, only one is removed.

Parameters

`es`: the list of edges to be removed. Edges are identifed by edge IDs. `EdgeSeq` objects are also accepted here. No argument deletes all edges.

Overrides: igraph.GraphBase.delete_edges

Deprecated: `Graph.delete_edges(None)` has been replaced by `Graph.delete_edges()` - with no arguments - since igraph 0.8.3.

indegree(*self*, **args*, ***kwds*)

Returns the in-degrees in a list.

See `degree` for possible arguments.

outdegree(*self*, **args*, ***kwds*)

Returns the out-degrees in a list.

See `degree` for possible arguments.

all_st_cuts(*self*, *source*, *target*)

Returns all the cuts between the source and target vertices in a directed graph.

This function lists all edge-cuts between a source and a target vertex. Every cut is listed exactly once.

Parameters

`source`: the source vertex ID

`target`: the target vertex ID

Return Value

a list of `Cut` objects.

Overrides: igraph.GraphBase.all_st_cuts

Reference: JS Provan and DR Shier: A paradigm for listing (s,t)-cuts in graphs. Algorithmica 15, 351–372, 1996.

all_st_mincuts(*self*, *source*, *target*, *capacity*=`None`)

Returns all the mincuts between the source and target vertices in a directed graph.

This function lists all minimum edge-cuts between a source and a target vertex.

Parameters

`source`: the source vertex ID

`target`: the target vertex ID

`capacity`: the edge capacities (weights). If `None`, all edges have equal weight. May also be an attribute name.

Return Value

a list of `Cut` objects.

Overrides: igraph.GraphBase.all_st_mincuts

Reference: JS Provan and DR Shier: A paradigm for listing (s,t)-cuts in graphs. Algorithmica 15, 351–372, 1996.

biconnected_components(*self*, *return_articulation_points*=`False`)

Calculates the biconnected components of the graph.

Parameters

`return_articulation_points`: whether to return the articulation points as well

Return Value

a `VertexCover` object describing the biconnected components, and optionally the list of articulation points as well

Overrides: igraph.GraphBase.biconnected_components

blocks(*self*, *return_articulation_points*=`False`)

Calculates the biconnected components of the graph.

Parameters

`return_articulation_points`: whether to return the articulation points as well

Return Value

a `VertexCover` object describing the biconnected components, and optionally the list of articulation points as well

clear()

Clears the graph, deleting all vertices, edges, and attributes.

See Also: `Graph.delete_vertices` and `Graph.delete_edges`.

cohesive_blocks()

Calculates the cohesive block structure of the graph.

Cohesive blocking is a method of determining hierarchical subsets of graph vertices based on their structural cohesion (i.e. vertex connectivity). For a given graph G, a subset of its vertices S is said to be maximally k-cohesive if there is no superset of S with vertex connectivity greater than or equal to k. Cohesive blocking is a process through which, given a k-cohesive set of vertices, maximally l-cohesive subsets are recursively identified with $l > k$. Thus a hierarchy of vertex subsets is obtained in the end, with the entire graph G at its root.

Return Value

an instance of `CohesiveBlocks`. See the documentation of `CohesiveBlocks` for more information.

Overrides: igraph.GraphBase.cohesive_blocks

See Also: `CohesiveBlocks`

clusters(*mode*=`STRONG`)

Calculates the (strong or weak) clusters (connected components) for a given graph.

Parameters

`mode`: must be either `STRONG` or `WEAK`, depending on the clusters being sought. Optional, defaults to `STRONG`.

Return Value

a `VertexClustering` object

Overrides: igraph.GraphBase.clusters

components(*mode*=`STRONG`)

Calculates the (strong or weak) clusters (connected components) for a given graph.

Parameters

`mode`: must be either `STRONG` or `WEAK`, depending on the clusters being sought. Optional, defaults to `STRONG`.

Return Value

a `VertexClustering` object

degree_distribution(*bin_width*=`1`, ...)

Calculates the degree distribution of the graph.

Unknown keyword arguments are directly passed to `degree()`.

Parameters

`bin_width`: the bin width of the histogram

Return Value

a histogram representing the degree distribution of the graph.

dyad_census()

Calculates the dyad census of the graph.

Dyad census means classifying each pair of vertices of a directed graph into three categories: mutual (there is an edge from a to b and also from b to a), asymmetric (there is an edge from a to b or from b to a but not the other way round) and null (there is no connection between a and b).

Return Value

a `DyadCensus` object.

Overrides: igraph.GraphBase.dyad_census

Reference: Holland, P.W. and Leinhardt, S. (1970). A Method for Detecting Structure in Sociometric Data. American Journal of Sociology, 70, 492-513.

get_adjacency(*self*, *type*=2, *attribute*=None, *default*=0, *eids*=False)

Returns the adjacency matrix of a graph.

Parameters

`type`: either `GET_ADJACENCY_LOWER` (uses the lower triangle of the matrix) or `GET_ADJACENCY_UPPER` (uses the upper triangle) or `GET_ADJACENCY_BOTH` (uses both parts). Ignored for directed graphs.

`attribute`: if `None`, returns the ordinary adjacency matrix. When the name of a valid edge attribute is given here, the matrix returned will contain the default value at the places where there is no edge or the value of the given attribute where there is an edge. Multiple edges are not supported, the value written in the matrix in this case will be unpredictable. This parameter is ignored if *eids* is `True`

`default`: the default value written to the cells in the case of adjacency matrices with attributes.

`eids`: specifies whether the edge IDs should be returned in the adjacency matrix. Since zero is a valid edge ID, the cells in the matrix that correspond to unconnected vertex pairs will contain -1 instead of 0 if *eids* is `True`. If *eids* is `False`, the number of edges will be returned in the matrix for each vertex pair.

Return Value

the adjacency matrix as a `Matrix`.

Overrides: igraph.GraphBase.get_adjacency

get_adjacency_sparse(*self*, *attribute*=None)

Returns the adjacency matrix of a graph as scipy csr matrix.

Parameters

`attribute`: if `None`, returns the ordinary adjacency matrix. When the name of a valid edge attribute is given here, the matrix returned will contain the default value at the places where there is no edge or the value of the given attribute where there is an edge.

Return Value

the adjacency matrix as a `scipy.sparse.csr_matrix`.

get_adjlist(*mode*=`OUT`)

Returns the adjacency list representation of the graph.

The adjacency list representation is a list of lists. Each item of the outer list belongs to a single vertex of the graph. The inner list contains the neighbors of the given vertex.

Parameters

`mode`: if `OUT`, returns the successors of the vertex. If `IN`, returns the predecessors of the vertex. If `ALL`, both the predecessors and the successors will be returned. Ignored for undirected graphs.

get_adjedgelist(*mode*=`OUT`)

Returns the incidence list representation of the graph.

Deprecated: replaced by `Graph.get_inclist()` since igraph 0.6

See Also: Graph.get_inclist()

get_all_simple_paths(*v*, *to*=None, *mode*=OUT)

Calculates all the simple paths from a given node to some other nodes (or all of them) in a graph.

A path is simple if its vertices are unique, i.e. no vertex is visited more than once.

Note that potentially there are exponentially many paths between two vertices of a graph, especially if your graph is lattice-like. In this case, you may run out of memory when using this function.

Parameters

`v`: the source for the calculated paths

`to`: a vertex selector describing the destination for the calculated paths. This can be a single vertex ID, a list of vertex IDs, a single vertex name, a list of vertex names or a `VertexSeq` object. `None` means all the vertices.

`cutoff`: maximum length of path that is considered. If negative, paths of all lengths are considered.

`mode`: the directionality of the paths. `IN` means to calculate incoming paths, `OUT` means to calculate outgoing paths, `ALL` means to calculate both ones.

Return Value

all of the simple paths from the given node to every other reachable node in the graph in a list. Note that in case of mode=`IN`, the vertices in a path are returned in reversed order!

get_inclist(*mode*=OUT)

Returns the incidence list representation of the graph.

The incidence list representation is a list of lists. Each item of the outer list belongs to a single vertex of the graph. The inner list contains the IDs of the incident edges of the given vertex.

Parameters

`mode`: if `OUT`, returns the successors of the vertex. If `IN`, returns the predecessors of the vertex. If `ALL`, both the predecessors and the successors will be returned. Ignored for undirected graphs.

gomory_hu_tree(*capacity*=`None`, *flow*=`"flow"`)

Calculates the Gomory-Hu tree of an undirected graph with optional edge capacities.

The Gomory-Hu tree is a concise representation of the value of all the maximum flows (or minimum cuts) in a graph. The vertices of the tree correspond exactly to the vertices of the original graph in the same order. Edges of the Gomory-Hu tree are annotated by flow values. The value of the maximum flow (or minimum cut) between an arbitrary (u,v) vertex pair in the original graph is then given by the minimum flow value (i.e. edge annotation) along the shortest path between u and v in the Gomory-Hu tree.

Parameters

`capacity`: the edge capacities (weights). If `None`, all edges have equal weight. May also be an attribute name.

`flow`: the name of the edge attribute in the returned graph in which the flow values will be stored.

Return Value

the Gomory-Hu tree as a `Graph` object.

Overrides: igraph.GraphBase.gomory_hu_tree

is_named()

Returns whether the graph is named, i.e., whether it has a "name" vertex attribute.

is_weighted()

Returns whether the graph is weighted, i.e., whether it has a "weight" edge attribute.

maxflow(*source*, *target*, *capacity*=`None`)

Returns a maximum flow between the given source and target vertices in a graph.

A maximum flow from *source* to *target* is an assignment of non-negative real numbers to the edges of the graph, satisfying two properties:

1. For each edge, the flow (i.e. the assigned number) is not more than the capacity of the edge (see the *capacity* argument)
2. For every vertex except the source and the target, the incoming flow is the same as the outgoing flow.

The value of the flow is the incoming flow of the target or the outgoing flow of the source (which are equal). The maximum flow is the maximum possible such value.

Parameters

`capacity`: the edge capacities (weights). If `None`, all edges have equal weight. May also be an attribute name.

Return Value

a `Flow` object describing the maximum flow

Overrides: igraph.GraphBase.maxflow

mincut(*source*=`None`, *target*=`None`, *capacity*=`None`)

Calculates the minimum cut between the given source and target vertices or within the whole graph.

The minimum cut is the minimum set of edges that needs to be removed to separate the source and the target (if they are given) or to disconnect the graph (if neither the source nor the target are given). The minimum is calculated using the weights (capacities) of the edges, so the cut with the minimum total capacity is calculated.

For undirected graphs and no source and target, the method uses the Stoer-Wagner algorithm. For a given source and target, the method uses the push-relabel algorithm; see the references below.

Parameters

`source`: the source vertex ID. If `None`, the target must also be `None` and the calculation will be done for the entire graph (i.e. all possible vertex pairs).

`target`: the target vertex ID. If `None`, the source must also be `None` and the calculation will be done for the entire graph (i.e. all possible vertex pairs).

`capacity`: the edge capacities (weights). If `None`, all edges have equal weight. May also be an attribute name.

Return Value

a `Cut` object describing the minimum cut

Overrides: igraph.GraphBase.mincut

st_mincut(*source*, *target*, *capacity*=`None`)

Calculates the minimum cut between the source and target vertices in a graph.

Parameters

`source`: the source vertex ID

`target`: the target vertex ID

`capacity`: the capacity of the edges. It must be a list or a valid attribute name or `None`. In the latter case, every edge will have the same capacity.

Return Value

the value of the minimum cut, the IDs of vertices in the first and second partition, and the IDs of edges in the cut, packed in a 4-tuple

Overrides: igraph.GraphBase.st_mincut

modularity(*membership*, *weights*=`None`)

Calculates the modularity score of the graph with respect to a given clustering.

The modularity of a graph w.r.t. some division measures how good the division is, or how separated are the different vertex types from each other. It's defined as *Q=1/(2m)*sum(Aij-ki*kj/(2m)delta(ci,cj),i,j)*. *m* is the number of edges, *Aij* is the element of the *A* adjacency matrix in row *i* and column *j*, *ki* is the degree of node *i*, *kj* is the degree of node *j*, and *Ci* and `cj` are the types of the two vertices (*i* and *j*). *delta(x,y)* is one iff *x=y*, 0 otherwise.

If edge weights are given, the definition of modularity is modified as follows: *Aij* becomes the weight of the corresponding edge, *ki* is the total weight of edges adjacent to vertex *i*, *kj* is the total weight of edges adjacent to vertex *j* and *m* is the total edge weight in the graph.

Parameters

`membership:` a membership list or a `VertexClustering` object

`weights:` optional edge weights or `None` if all edges are weighed equally. Attribute names are also allowed.

Return Value

the modularity score

Overrides: igraph.GraphBase.modularity

Reference: MEJ Newman and M Girvan: Finding and evaluating community structure in networks. Phys Rev E 69 026113, 2004.

path_length_hist(*directed*=`True`)

Returns the path length histogram of the graph

Parameters

`directed:` whether to consider directed paths. Ignored for undirected graphs.

Return Value

a `Histogram` object. The object will also have an `unconnected` attribute that stores the number of unconnected vertex pairs (where the second vertex can not be reached from the first one). The latter one will be of type long (and not a simple integer), since this can be *very* large.

Overrides: igraph.GraphBase.path_length_hist

pagerank(*self*, *vertices*=`None`, *directed*=`True`, *damping*=`0.85`, *weights*=`None`, *arpack_ options*=`None`, *implementation*=`'prpack'`, *niter*=`1000`, *eps*=`0.001`)

Calculates the Google PageRank values of a graph.

Parameters

`vertices:` the indices of the vertices being queried. `None` means all of the vertices.

`directed:` whether to consider directed paths.

`damping:` the damping factor. *1-damping* is the PageRank value for nodes with no incoming links. It is also the probability of resetting the random walk to a uniform distribution in each step.

`weights:` edge weights to be used. Can be a sequence or iterable or even an edge attribute name.

`arpack_options:` an `ARPACKOptions` object used to fine-tune the ARPACK eigenvector calculation. If omitted, the module-level variable called `arpack_options` is used. This argument is ignored if not the ARPACK implementation is used, see the *implementation* argument.

`implementation:` which implementation to use to solve the PageRank eigenproblem. Possible values are:

- `"prpack"`: use the PRPACK library. This is a new implementation in igraph 0.7
- `"arpack"`: use the ARPACK library. This implementation was used from version 0.5, until version 0.7.
- `"power"`: use a simple power method. This is the implementation that was used before igraph version 0.5.

`niter:` The number of iterations to use in the power method implementation. It is ignored in the other implementations

`eps:` The power method implementation will consider the calculation as complete if the difference of PageRank values between iterations change less than this value for every node. It is ignored by the other implementations.

Return Value

a list with the Google PageRank values of the specified vertices.

spanning_tree(*self*, *weights*=`None`, *return_tree*=`True`)

Calculates a minimum spanning tree for a graph.

Parameters

`weights`: a vector containing weights for every edge in the graph. `None` means that the graph is unweighted.

`return_tree`: whether to return the minimum spanning tree (when `return_tree` is `True`) or to return the IDs of the edges in the minimum spanning tree instead (when `return_tree` is `False`). The default is `True` for historical reasons as this argument was introduced in igraph 0.6.

Return Value

the spanning tree as a `Graph` object if `return_tree` is `True` or the IDs of the edges that constitute the spanning tree if `return_tree` is `False`.

Reference: Prim, R.C.: *Shortest connection networks and some generalizations*. Bell System Technical Journal 36:1389-1401, 1957.

transitivity_avglocal_undirected(*self*, *mode*='nan', *weights*=None)

Calculates the average of the vertex transitivities of the graph.

In the unweighted case, the transitivity measures the probability that two neighbors of a vertex are connected. In case of the average local transitivity, this probability is calculated for each vertex and then the average is taken. Vertices with less than two neighbors require special treatment, they will either be left out from the calculation or they will be considered as having zero transitivity, depending on the *mode* parameter. The calculation is slightly more involved for weighted graphs; in this case, weights are taken into account according to the formula of Barrat et al (see the references).

Note that this measure is different from the global transitivity measure (see `transitivity_undirected()`) as it simply takes the average local transitivity across the whole network.

Parameters

`mode`: defines how to treat vertices with degree less than two. If `TRANSITIVITY_ZERO` or `"zero"`, these vertices will have zero transitivity. If `TRANSITIVITY_NAN` or `"nan"`, these vertices will be excluded from the average.

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name.

Overrides: igraph.GraphBase.transitivity_avglocal_undirected

See Also: `transitivity_undirected()`, `transitivity_local_undirected()`

Reference:

- Watts DJ and Strogatz S: *Collective dynamics of small-world networks*. Nature 393(6884):440-442, 1998.
- Barrat A, Barthelemy M, Pastor-Satorras R and Vespignani A: *The architecture of complex weighted networks*. PNAS 101, 3747 (2004). *http://arxiv.org/abs/cond-mat/0311416*.

triad_census()

Calculates the triad census of the graph.

Return Value

a `TriadCensus` object.

Overrides: igraph.GraphBase.triad_census

Reference: Davis, J.A. and Leinhardt, S. (1972). The Structure of Positive Interpersonal Relations in Small Groups. In: J. Berger (Ed.), Sociological Theories in Progress, Volume 2, 218-251. Boston: Houghton Mifflin.

count_automorphisms_vf2(*self*, *color*=None, *edge_color*=None, *node_compat_fn*=None, *edge_compat_fn*=None)

Returns the number of automorphisms of the graph.

This function simply calls `count_isomorphisms_vf2` with the graph itself. See `count_isomorphisms_vf2` for an explanation of the parameters.

Return Value

the number of automorphisms of the graph

See Also: Graph.count_isomorphisms_vf2

get_automorphisms_vf2(*self*, *color*=None, *edge_color*=None, *node_compat_fn*=None, *edge_compat_fn*=None)

Returns all the automorphisms of the graph

This function simply calls `get_isomorphisms_vf2` with the graph itself. See `get_isomorphisms_vf2` for an explanation of the parameters.

Return Value

a list of lists, each item containing a possible mapping of the graph vertices to itself according to the automorphism

See Also: Graph.get_isomorphisms_vf2

community_fastgreedy(*self*, *weights*=None)

Community structure based on the greedy optimization of modularity.

This algorithm merges individual nodes into communities in a way that greedily maximizes the modularity score of the graph. It can be proven that if no merge can increase the current modularity score, the algorithm can be stopped since no further increase can be achieved.

This algorithm is said to run almost in linear time on sparse graphs.

Parameters

`weights:` edge attribute name or a list containing edge weights

Return Value

an appropriate `VertexDendrogram` object.

Overrides: igraph.GraphBase.community_fastgreedy

Reference: A Clauset, MEJ Newman and C Moore: Finding community structure in very large networks. Phys Rev E 70, 066111 (2004).

community_infomap(*self*, *edge_weights*=`None`, *vertex_weights*=`None`, *trials*=`10`)

Finds the community structure of the network according to the Infomap method of Martin Rosvall and Carl T. Bergstrom.

Parameters

`edge_weights`: name of an edge attribute or a list containing edge weights.

`vertex_weights`: name of an vertex attribute or a list containing vertex weights.

`trials`: the number of attempts to partition the network.

Return Value

an appropriate `VertexClustering` object with an extra attribute called `codelength` that stores the code length determined by the algorithm.

Overrides: igraph.GraphBase.community_infomap

Reference:

- M. Rosvall and C. T. Bergstrom: Maps of information flow reveal community structure in complex networks, PNAS 105, 1118 (2008). *http://dx.doi.org/10.1073/pnas.0706851105*, *http://arxiv.org/abs/0707.0609*.
- M. Rosvall, D. Axelsson, and C. T. Bergstrom: The map equation, Eur. Phys. J. Special Topics 178, 13 (2009). *http://dx.doi.org/10.1140/epjst/e2010-01179-1*, *http://arxiv.org/abs/0906.1405*.

community_leading_eigenvector_naive(*clusters*=`None`, *return_merges*=`False`)

A naive implementation of Newman's eigenvector community structure detection. This function splits the network into two components according to the leading eigenvector of the modularity matrix and then recursively takes the given number of steps by splitting the communities as individual networks. This is not the correct way, however, see the reference for explanation. Consider using the correct `community_leading_eigenvector` method instead.

Parameters

`clusters`: the desired number of communities. If `None`, the algorithm tries to do as many splits as possible. Note that the algorithm won't split a community further if the signs of the leading eigenvector are all the same, so the actual number of discovered communities can be less than the desired one.

`return_merges`: whether the returned object should be a dendrogram instead of a single clustering.

Return Value

an appropriate `VertexClustering` or `VertexDendrogram` object.

Reference: MEJ Newman: Finding community structure in networks using the eigenvectors of matrices, arXiv:physics/0605087

community_leading_eigenvector(*clusters*=`None`, *weights*=`None`, *arpack_options*=`None`)

Newman's leading eigenvector method for detecting community structure. This is the proper implementation of the recursive, divisive algorithm: each split is done by maximizing the modularity regarding the original network.

Parameters

`clusters`: the desired number of communities. If `None`, the algorithm tries to do as many splits as possible. Note that the algorithm won't split a community further if the signs of the leading eigenvector are all the same, so the actual number of discovered communities can be less than the desired one.

`weights`: name of an edge attribute or a list containing edge weights.

`arpack_options`: an `ARPACKOptions` object used to fine-tune the ARPACK eigenvector calculation. If omitted, the module-level variable called `arpack_options` is used.

Return Value

an appropriate `VertexClustering` object.

Overrides: igraph.GraphBase.community_leading_eigenvector

Reference: MEJ Newman: Finding community structure in networks using the eigenvectors of matrices, arXiv:physics/0605087

community_label_propagation(*weights*=`None`, *initial*=`None`, *fixed*=`None`)

Finds the community structure of the graph according to the label propagation method of Raghavan et al. Initially, each vertex is assigned a different label. After that, each vertex chooses the dominant label in its neighbourhood in each iteration. Ties are broken randomly and the order in which the vertices are updated is randomized before every iteration. The algorithm ends when vertices reach a consensus. Note that since ties are broken randomly, there is no guarantee that the algorithm returns the same community structure after each run. In fact, they frequently differ. See the paper of Raghavan et al on how to come up with an aggregated community structure.

Parameters

`weights`: name of an edge attribute or a list containing edge weights

`initial`: name of a vertex attribute or a list containing the initial vertex labels. Labels are identified by integers from zero to *n-1* where *n* is the number of vertices. Negative numbers may also be present in this vector, they represent unlabeled vertices.

`fixed`: a list of booleans for each vertex. `True` corresponds to vertices whose labeling should not change during the algorithm. It only makes sense if initial labels are also given. Unlabeled vertices cannot be fixed.

Return Value

an appropriate `VertexClustering` object.

Overrides: igraph.GraphBase.community_label_propagation

Reference: Raghavan, U.N. and Albert, R. and Kumara, S. Near linear time algorithm to detect community structures in large-scale networks. Phys Rev E 76:036106, 2007. *http://arxiv.org/abs/0709.2938*.

community_multilevel(*self*, *weights*=`None`, *return_levels*=`False`)

Community structure based on the multilevel algorithm of Blondel et al.

This is a bottom-up algorithm: initially every vertex belongs to a separate community, and vertices are moved between communities iteratively in a way that maximizes the vertices' local contribution to the overall modularity score. When a consensus is reached (i.e. no single move would increase the modularity score), every community in the original graph is shrank to a single vertex (while keeping the total weight of the adjacent edges) and the process continues on the next level. The algorithm stops when it is not possible to increase the modularity any more after shrinking the communities to vertices.

This algorithm is said to run almost in linear time on sparse graphs.

Parameters

`weights`: edge attribute name or a list containing edge weights

`return_levels`: if `True`, the communities at each level are returned in a list. If `False`, only the community structure with the best modularity is returned.

Return Value

a list of `VertexClustering` objects, one corresponding to each level (if `return_levels` is `True`), or a `VertexClustering` corresponding to the best modularity.

Overrides: igraph.GraphBase.community_multilevel

Reference: VD Blondel, J-L Guillaume, R Lambiotte and E Lefebvre: Fast unfolding of community hierarchies in large networks, J Stat Mech P10008 (2008), http://arxiv.org/abs/0803.0476

community_optimal_modularity(*self*, **args*, ***kwds*)

Calculates the optimal modularity score of the graph and the corresponding community structure.

This function uses the GNU Linear Programming Kit to solve a large integer optimization problem in order to find the optimal modularity score and the corresponding community structure, therefore it is unlikely to work for graphs larger than a few (less than a hundred) vertices. Consider using one of the heuristic approaches instead if you have such a large graph.

Parameters

`weights`: name of an edge attribute or a list containing edge weights.

Return Value

the calculated membership vector and the corresponding modularity in a tuple.

Overrides: igraph.GraphBase.community_optimal_modularity

community_edge_betweenness(*self*, *clusters*=`None`, *directed*=`True`, *weights*=`None`)

Community structure based on the betweenness of the edges in the network.

The idea is that the betweenness of the edges connecting two communities is typically high, as many of the shortest paths between nodes in separate communities go through them. So we gradually remove the edge with the highest betweenness and recalculate the betweennesses after every removal. This way sooner or later the network falls of to separate components. The result of the clustering will be represented by a dendrogram.

Parameters

`clusters`: the number of clusters we would like to see. This practically defines the "level" where we "cut" the dendrogram to get the membership vector of the vertices. If `None`, the dendrogram is cut at the level which maximizes the modularity when the graph is unweighted; otherwise the dendrogram is cut at at a single cluster (because cluster count selection based on modularities does not make sense for this method if not all the weights are equal).

`directed`: whether the directionality of the edges should be taken into account or not.

`weights`: name of an edge attribute or a list containing edge weights.

Return Value

a `VertexDendrogram` object, initally cut at the maximum modularity or at the desired number of clusters.

Overrides: igraph.GraphBase.community_edge_betweenness

community_spinglass(*weights*=None, *spins*=25, *parupdate*=False, *start_temp*=1, *stop_temp*=0.01, *cool_fact*=0.99, *update_rule*="config", *gamma*=1, *implementation*="orig", *lambda_*=1)

Finds the community structure of the graph according to the spinglass community detection method of Reichardt & Bornholdt.

Parameters

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name.

`spins`: integer, the number of spins to use. This is the upper limit for the number of communities. It is not a problem to supply a (reasonably) big number here, in which case some spin states will be unpopulated.

`parupdate`: whether to update the spins of the vertices in parallel (synchronously) or not

`start_temp`: the starting temperature

`stop_temp`: the stop temperature

`cool_fact`: cooling factor for the simulated annealing

`update_rule`: specifies the null model of the simulation. Possible values are `"config"` (a random graph with the same vertex degrees as the input graph) or `"simple"` (a random graph with the same number of edges)

`gamma`: the gamma argument of the algorithm, specifying the balance between the importance of present and missing edges within a community. The default value of 1.0 assigns equal importance to both of them.

`implementation`: currently igraph contains two implementations of the spinglass community detection algorithm. The faster original implementation is the default. The other implementation is able to take into account negative weights, this can be chosen by setting `implementation` to `"neg"`

`lambda_`: the lambda argument of the algorithm, which specifies the balance between the importance of present and missing negatively weighted edges within a community. Smaller values of lambda lead to communities with less negative intra-connectivity. If the argument is zero, the algorithm reduces to a graph coloring algorithm, using the number of spins as colors. This argument is ignored if the original implementation is used. Note the underscore at the end of the argument name; this is due to the fact that lambda is a reserved keyword in Python.

community_walktrap(*self*, *weights*=`None`, *steps*=`4`)

Community detection algorithm of Latapy & Pons, based on random walks.

The basic idea of the algorithm is that short random walks tend to stay in the same community. The result of the clustering will be represented as a dendrogram.

Parameters

`weights`: name of an edge attribute or a list containing edge weights

`steps`: length of random walks to perform

Return Value

a `VertexDendrogram` object, initially cut at the maximum modularity.

Overrides: igraph.GraphBase.community_walktrap

Reference: Pascal Pons, Matthieu Latapy: Computing communities in large networks using random walks, *http://arxiv.org/abs/physics/0512106*.

k_core(*self*, **args*)

Returns some k-cores of the graph.

The method accepts an arbitrary number of arguments representing the desired indices of the k-cores to be returned. The arguments can also be lists or tuples. The result is a single `Graph` object if an only integer argument was given, otherwise the result is a list of `Graph` objects representing the desired k-cores in the order the arguments were specified. If no argument is given, returns all k-cores in increasing order of k.

community_leiden(*objective_function*=`CPM`, *weights*=`None`, *resolution_parameter*=`1.0`, *beta*=`0.01`, *initial_membership*=`None`, *n_iterations*=`2`, *node_weights*=`None`)

Finds the community structure of the graph using the Leiden algorithm of Traag, van Eck & Waltman.

Parameters

`objective_function:`	whether to use the Constant Potts Model (CPM) or modularity. Must be either `"CPM"` or `"modularity"`.
`weights:`	edge weights to be used. Can be a sequence or iterable or even an edge attribute name.
`resolution_parameter:`	the resolution parameter to use. Higher resolutions lead to more smaller communities, while lower resolutions lead to fewer larger communities.
`beta:`	parameter affecting the randomness in the Leiden algorithm. This affects only the refinement step of the algorithm.
`initial_membership:`	if provided, the Leiden algorithm will try to improve this provided membership. If no argument is provided, the aglorithm simply starts from the singleton partition.
`n_iterations:`	the number of iterations to iterate the Leiden algorithm. Each iteration may improve the partition further. Using a negative number of iterations will run until a stable iteration is encountered (i.e. the quality was not increased during that iteration).
`node_weights:`	the node weights used in the Leiden algorithm. If this is not provided, it will be automatically determined on the basis of whether you want to use CPM or modularity. If you do provide this, please make sure that you understand what you are doing.

Return Value

an appropriate `VertexClustering` object.

Overrides: igraph.GraphBase.community_leiden

Reference: Traag, V. A., Waltman, L., & van Eck, N. J. (2019). From Louvain to Leiden: guaranteeing well-connected communities. Scientific reports, 9(1), 5233. doi: 10.1038/s41598-019-41695-z

layout(*self*, *layout*=`None`, **args*, ***kwds*)

Returns the layout of the graph according to a layout algorithm.

Parameters and keyword arguments not specified here are passed to the layout algorithm directly. See the documentation of the layout algorithms for the explanation of these parameters.

Registered layout names understood by this method are:

- `auto`, `automatic`: automatic layout (see `Graph.layout_auto`)
- `bipartite`: bipartite layout (see `Graph.layout_bipartite`)
- `circle`, `circular`: circular layout (see `Graph.layout_circle`)
- `dh`, `davidson_harel`: Davidson-Harel layout (see `Graph.layout_davidson_harel`)
- `drl`: DrL layout for large graphs (see `Graph.layout_drl`)
- `drl_3d`: 3D DrL layout for large graphs (see `Graph.layout_drl`)
- `fr`, `fruchterman_reingold`: Fruchterman-Reingold layout (see `Graph.layout_fruchterman_reingold`).
- `fr_3d`, `fr3d`, `fruchterman_reingold_3d`: 3D Fruchterman- Reingold layout (see `Graph.layout_fruchterman_reingold`).
- `grid`: regular grid layout in 2D (see `Graph.layout_grid`)
- `grid_3d`: regular grid layout in 3D (see `Graph.layout_grid_3d`)
- `graphopt`: the graphopt algorithm (see `Graph.layout_graphopt`)
- `kk`, `kamada_kawai`: Kamada-Kawai layout (see `Graph.layout_kamada_kawai`)
- `kk_3d`, `kk3d`, `kamada_kawai_3d`: 3D Kamada-Kawai layout (see `Graph.layout_kamada_kawai`)
- `lgl`, `large`, `large_graph`: Large Graph Layout (see `Graph.layout_lgl`)
- `mds`: multidimensional scaling layout (see `Graph.layout_mds`)
- `random`: random layout (see `Graph.layout_random`)
- `random_3d`: random 3D layout (see `Graph.layout_random`)
- `rt`, `tree`, `reingold_tilford`: Reingold-Tilford tree layout (see `Graph.layout_reingold_tilford`)
- `rt_circular`, `reingold_tilford_circular`: circular Reingold-Tilford tree layout (see `Graph.layout_reingold_tilford_circular`)
- `sphere`, `spherical`, `circle_3d`, `circular_3d`: spherical layout (see `Graph.layout_circle`)
- `star`: star layout (see `Graph.layout_star`)
- `sugiyama`: Sugiyama layout (see `Graph.layout_sugiyama`)

Parameters

`layout`: the layout to use. This can be one of the registered layout names or a callable which returns either a `Layout` object or a list of lists containing the coordinates. If `None`, uses the value of the `plotting.layout` configuration key.

Return Value

layout_auto(*self*, **args*, ***kwds*)

Chooses and runs a suitable layout function based on simple topological properties of the graph.

This function tries to choose an appropriate layout function for the graph using the following rules:

1. If the graph has an attribute called `layout`, it will be used. It may either be a `Layout` instance, a list of coordinate pairs, the name of a layout function, or a callable function which generates the layout when called with the graph as a parameter.
2. Otherwise, if the graph has vertex attributes called `x` and `y`, these will be used as coordinates in the layout. When a 3D layout is requested (by setting `dim` to 3), a vertex attribute named `z` will also be needed.
3. Otherwise, if the graph is connected and has at most 100 vertices, the Kamada-Kawai layout will be used (see `Graph.layout_kamada_kawai()`).
4. Otherwise, if the graph has at most 1000 vertices, the Fruchterman-Reingold layout will be used (see `Graph.layout_fruchterman_reingold()`).
5. If everything else above failed, the DrL layout algorithm will be used (see `Graph.layout_drl()`).

All the arguments of this function except `dim` are passed on to the chosen layout function (in case we have to call some layout function).

Parameters

`dim`: specifies whether we would like to obtain a 2D or a 3D layout.

Return Value

a `Layout` object.

layout_grid_fruchterman_reingold(**args*, ***kwds*)

Compatibility alias to the Fruchterman-Reingold layout with the grid option turned on.

See Also: Graph.layout_fruchterman_reingold()

layout_sugiyama(*layers*=`None`, *weights*=`None`, *hgap*=`1`, *vgap*=`1`, *maxiter*=`100`, *return_extended_graph*=`False`)

Places the vertices using a layered Sugiyama layout.

This is a layered layout that is most suitable for directed acyclic graphs, although it works on undirected or cyclic graphs as well.

Each vertex is assigned to a layer and each layer is placed on a horizontal line. Vertices within the same layer are then permuted using the barycenter heuristic that tries to minimize edge crossings.

Dummy vertices will be added on edges that span more than one layer. The returned layout therefore contains more rows than the number of nodes in the original graph; the extra rows correspond to the dummy vertices.

Parameters

`layers`: a vector specifying a non-negative integer layer index for each vertex, or the name of a numeric vertex attribute that contains the layer indices. If `None`, a layering will be determined automatically. For undirected graphs, a spanning tree will be extracted and vertices will be assigned to layers using a breadth first search from the node with the largest degree. For directed graphs, cycles are broken by reversing the direction of edges in an approximate feedback arc set using the heuristic of Eades, Lin and Smyth, and then using longest path layering to place the vertices in layers.

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name.

`hgap`: minimum horizontal gap between vertices in the same layer.

`vgap`: vertical gap between layers. The layer index will be multiplied by *vgap* to obtain the Y coordinate.

`maxiter`: maximum number of iterations to take in the crossing reduction step. Increase this if you feel that you are getting too many edge crossings.

`return_extended_graph`: specifies that the extended graph with the added dummy vertices should also be returned. When this is `True`, the result will be a tuple containing the layout and the extended graph. The first |V| nodes of the extended graph will correspond to

maximum_bipartite_matching(*self*, *types*='type', *weights*=None, *eps*=None)

Finds a maximum matching in a bipartite graph.

A maximum matching is a set of edges such that each vertex is incident on at most one matched edge and the number (or weight) of such edges in the set is as large as possible.

Parameters

`types`: vertex types in a list or the name of a vertex attribute holding vertex types. Types should be denoted by zeros and ones (or `False` and `True`) for the two sides of the bipartite graph. If omitted, it defaults to `type`, which is the default vertex type attribute for bipartite graphs.

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name.

`eps`: a small real number used in equality tests in the weighted bipartite matching algorithm. Two real numbers are considered equal in the algorithm if their difference is smaller than this value. This is required to avoid the accumulation of numerical errors. If you pass `None` here, igraph will try to determine an appropriate value automatically.

Return Value

an instance of `Matching`.

to_networkx(*self*)

Converts the graph to networkx format

from_networkx(*klass*, *g*)

Converts the graph from networkx

Vertex names will be converted to "_nx_name" attribute and the vertices will get new ids from 0 up (as standard in igraph).

Parameters

`g`: networkx Graph or DiGraph

to_graph_tool(*self*, *graph_attributes*=None, *vertex_attributes*=None, *edge_attributes*=None)

```
Converts the graph to graph-tool

@param graph_attributes: dictionary of graph attributes to transfer.
  Keys are attributes from the graph, values are data types (see
  below). C{None} means no graph attributes are transferred.
@param vertex_attributes: dictionary of vertex attributes to transfer.
  Keys are attributes from the vertices, values are data types (see
  below). C{None} means no vertex attributes are transferred.
@param edge_attributes: dictionary of edge attributes to transfer.
  Keys are attributes from the edges, values are data types (see
  below). C{None} means no vertex attributes are transferred.

Data types: graph-tool only accepts specific data types. See the
following web page for a list:

https://graph-tool.skewed.de/static/doc/quickstart.html

NOTE: because of the restricted data types in graph-tool, vertex and
edge attributes require to be type-consistent across all vertices or
edges. If you set the property for only some vertices/edges, the other
will be tagged as None in python-igraph, so they can only be converted
to graph-tool with the type 'object' and any other conversion will
fail.
```

from_graph_tool(*klass*, *g*)

Converts the graph from graph-tool

Parameters

`g`: graph-tool Graph

write_adjacency(*self*, *f*, *sep*=' ', *eol*='\n', **args*, ***kwds*)

Writes the adjacency matrix of the graph to the given file

All the remaining arguments not mentioned here are passed intact to `Graph.get_adjacency`.

Parameters

`f`: the name of the file to be written.

`sep`: the string that separates the matrix elements in a row

`eol`: the string that separates the rows of the matrix. Please note that igraph is able to read back the written adjacency matrix if and only if this is a single newline character

Read_Adjacency(*klass*, *f*, *sep*=`None`, *comment_char*='#', *attribute*=`None`, **args*, ***kwds*)

Constructs a graph based on an adjacency matrix from the given file

Additional positional and keyword arguments not mentioned here are passed intact to `Graph.Adjacency`.

Parameters

`f`: the name of the file to be read or a file object

`sep`: the string that separates the matrix elements in a row. `None` means an arbitrary sequence of whitespace characters.

`comment_char`: lines starting with this string are treated as comments.

`attribute`: an edge attribute name where the edge weights are stored in the case of a weighted adjacency matrix. If `None`, no weights are stored, values larger than 1 are considered as edge multiplicities.

Return Value

the created graph

write_dimacs(*self*, *f*, *source*=`None`, *target*=`None`, *capacity*=`'capacity'`)

Writes the graph in DIMACS format to the given file.

Parameters

`f`:	the name of the file to be written or a Python file handle.
`source`:	the source vertex ID. If `None`, igraph will try to infer it from the `source` graph attribute.
`target`:	the target vertex ID. If `None`, igraph will try to infer it from the `target` graph attribute.
`capacity`:	the capacities of the edges in a list or the name of an edge attribute that holds the capacities. If there is no such edge attribute, every edge will have a capacity of 1.

Overrides: igraph.GraphBase.write_dimacs

write_graphmlz(*self*, *f*, *compresslevel*=`9`)

Writes the graph to a zipped GraphML file.

The library uses the gzip compression algorithm, so the resulting file can be unzipped with regular gzip uncompression (like `gunzip` or `zcat` from Unix command line) or the Python `gzip` module.

Uses a temporary file to store intermediate GraphML data, so make sure you have enough free space to store the unzipped GraphML file as well.

Parameters

`f`:	the name of the file to be written.
`compresslevel`:	the level of compression. 1 is fastest and produces the least compression, and 9 is slowest and produces the most compression.

Read_DIMACS(*f*, *directed*=`False`)

Reads a graph from a file conforming to the DIMACS minimum-cost flow file format.

For the exact definition of the format, see *http://lpsolve.sourceforge.net/5.5/DIMACS.htm*.

Restrictions compared to the official description of the format are as follows:

- igraph's DIMACS reader requires only three fields in an arc definition, describing the edge's source and target node and its capacity.
- Source vertices are identified by 's' in the FLOW field, target vertices are identified by 't'.
- Node indices start from 1. Only a single source and target node is allowed.

Parameters

`f`: the name of the file or a Python file handle

`directed`: whether the generated graph should be directed.

Return Value

the generated graph. The indices of the source and target vertices are attached as graph attributes `source` and `target`, the edge capacities are stored in the `capacity` edge attribute.

Overrides: igraph.GraphBase.Read_DIMACS

Read_GraphMLz(*f*, *directed*=`True`, *index*=`0`)

Reads a graph from a zipped GraphML file.

Parameters

`f`: the name of the file

`index`: if the GraphML file contains multiple graphs, specified the one that should be loaded. Graph indices start from zero, so if you want to load the first graph, specify 0 here.

Return Value

the loaded graph object

write_pickle(*self*, *fname*=`None`, *version*=`-1`)

Saves the graph in Python pickled format

Parameters

`fname`: the name of the file or a stream to save to. If `None`, saves the graph to a string and returns the string.

`version`: pickle protocol version to be used. If -1, uses the highest protocol available

Return Value

`None` if the graph was saved successfully to the given file, or a string if `fname` was `None`.

write_picklez(*self*, *fname*=`None`, *version*=`-1`)

Saves the graph in Python pickled format, compressed with gzip.

Saving in this format is a bit slower than saving in a Python pickle without compression, but the final file takes up much less space on the hard drive.

Parameters

`fname`: the name of the file or a stream to save to.

`version`: pickle protocol version to be used. If -1, uses the highest protocol available

Return Value

`None` if the graph was saved successfully to the given file.

Read_Pickle(*klass*, *fname*=`None`)

Reads a graph from Python pickled format

Parameters

`fname`: the name of the file, a stream to read from, or a string containing the pickled data.

Return Value

the created graph object.

Read_Picklez(*klass*, *fname*, **args*, ***kwds*)

Reads a graph from compressed Python pickled format, uncompressing it on-the-fly.

Parameters

`fname`: the name of the file or a stream to read from.

Return Value

the created graph object.

write_svg(*self*, *fname*, *layout*=`'auto'`, *width*=`None`, *height*=`None`, *labels*=`'label'`, *colors*=`'color'`, *shapes*=`'shape'`, *vertex_size*=`10`, *edge_colors*=`'color'`, *edge_stroke_widths*=`'width'`, *font_size*=`16`, **args*, ***kwds*)

Saves the graph as an SVG (Scalable Vector Graphics) file

The file will be Inkscape (http://inkscape.org) compatible. In Inkscape, as nodes are rearranged, the edges auto-update.

Parameters

`fname:`	the name of the file or a Python file handle
`layout:`	the layout of the graph. Can be either an explicitly specified layout (using a list of coordinate pairs) or the name of a layout algorithm (which should refer to a method in the `Graph` object, but without the `layout_` prefix.
`width:`	the preferred width in pixels (default: 400)
`height:`	the preferred height in pixels (default: 400)
`labels:`	the vertex labels. Either it is the name of a vertex attribute to use, or a list explicitly specifying the labels. It can also be `None`.
`colors:`	the vertex colors. Either it is the name of a vertex attribute to use, or a list explicitly specifying the colors. A color can be anything acceptable in an SVG file.
`shapes:`	the vertex shapes. Either it is the name of a vertex attribute to use, or a list explicitly specifying the shapes as integers. Shape 0 means hidden (nothing is drawn), shape 1 is a circle, shape 2 is a rectangle and shape 3 is a rectangle that automatically sizes to the inner text.
`vertex_size:`	vertex size in pixels
`edge_colors:`	the edge colors. Either it is the name of an edge attribute to use, or a list explicitly specifying the colors. A color can be anything acceptable in an SVG file.
`edge_stroke_widths:`	the stroke widths of the edges. Either it is the name of an edge attribute to use, or a list explicitly specifying the stroke widths. The stroke width can be anything acceptable in an SVG file.
`font_size:`	font size. If it is a string, it is written into the SVG file as-is (so you can specify anything which is valid as the value of the `font-size` style). If it is a number, it is

Read(*klass*, *f*, *format*=`None`, **args*, ***kwds*)

Unified reading function for graphs.

This method tries to identify the format of the graph given in the first parameter and calls the corresponding reader method.

The remaining arguments are passed to the reader method without any changes.

Parameters

`f`: the file containing the graph to be loaded

`format`: the format of the file (if known in advance). `None` means auto-detection. Possible values are: `"ncol"` (NCOL format), `"lgl"` (LGL format), `"graphdb"` (GraphDB format), `"graphml"`, `"graphmlz"` (GraphML and gzipped GraphML format), `"gml"` (GML format), `"net"`, `"pajek"` (Pajek format), `"dimacs"` (DIMACS format), `"edgelist"`, `"edges"` or `"edge"` (edge list), `"adjacency"` (adjacency matrix), `"dl"` (DL format used by UCINET), `"pickle"` (Python pickled format), `"picklez"` (gzipped Python pickled format)

Raises

`IOError` if the file format can't be identified and none was given.

Load(*klass*, *f*, *format*=`None`, **args*, ***kwds*)

Unified reading function for graphs.

This method tries to identify the format of the graph given in the first parameter and calls the corresponding reader method.

The remaining arguments are passed to the reader method without any changes.

Parameters

`f:` the file containing the graph to be loaded

`format:` the format of the file (if known in advance). `None` means auto-detection. Possible values are: `"ncol"` (NCOL format), `"lgl"` (LGL format), `"graphdb"` (GraphDB format), `"graphml"`, `"graphmlz"` (GraphML and gzipped GraphML format), `"gml"` (GML format), `"net"`, `"pajek"` (Pajek format), `"dimacs"` (DIMACS format), `"edgelist"`, `"edges"` or `"edge"` (edge list), `"adjacency"` (adjacency matrix), `"dl"` (DL format used by UCINET), `"pickle"` (Python pickled format), `"picklez"` (gzipped Python pickled format)

Raises

`IOError` if the file format can't be identified and none was given.

write(*self*, *f*, *format*=`None`, **args*, ***kwds*)

Unified writing function for graphs.

This method tries to identify the format of the graph given in the first parameter (based on extension) and calls the corresponding writer method.

The remaining arguments are passed to the writer method without any changes.

Parameters

`f:` the file containing the graph to be saved

`format:` the format of the file (if one wants to override the format determined from the filename extension, or the filename itself is a stream). `None` means auto-detection. Possible values are:

- `"adjacency"`: adjacency matrix format
- `"dimacs"`: DIMACS format
- `"dot"`, `"graphviz"`: GraphViz DOT format
- `"edgelist"`, `"edges"` or `"edge"`: numeric edge list format
- `"gml"`: GML format
- `"graphml"` and `"graphmlz"`: standard and gzipped GraphML format
- `"gw"`, `"leda"`, `"lgr"`: LEDA native format
- `"lgl"`: LGL format
- `"ncol"`: NCOL format
- `"net"`, `"pajek"`: Pajek format
- `"pickle"`, `"picklez"`: standard and gzipped Python pickled format
- `"svg"`: SVG format

Raises

`IOError` if the file format can't be identified and none was given.

save(*self*, *f*, *format*=`None`, **args*, ***kwds*)

Unified writing function for graphs.

This method tries to identify the format of the graph given in the first parameter (based on extension) and calls the corresponding writer method.

The remaining arguments are passed to the writer method without any changes.

Parameters

`f`: the file containing the graph to be saved

`format`: the format of the file (if one wants to override the format determined from the filename extension, or the filename itself is a stream). `None` means auto-detection. Possible values are:

- `"adjacency"`: adjacency matrix format
- `"dimacs"`: DIMACS format
- `"dot"`, `"graphviz"`: GraphViz DOT format
- `"edgelist"`, `"edges"` or `"edge"`: numeric edge list format
- `"gml"`: GML format
- `"graphml"` and `"graphmlz"`: standard and gzipped GraphML format
- `"gw"`, `"leda"`, `"lgr"`: LEDA native format
- `"lgl"`: LGL format
- `"ncol"`: NCOL format
- `"net"`, `"pajek"`: Pajek format
- `"pickle"`, `"picklez"`: standard and gzipped Python pickled format
- `"svg"`: SVG format

Raises

`IOError` if the file format can't be identified and none was given.

DictList(*klass*, *vertices*, *edges*, *directed*=`False`, *vertex_name_attr*=`'name'`, *edge_foreign_keys*=`('source', 'target')`, *iterative*=`False`)

Constructs a graph from a list-of-dictionaries representation.

This representation assumes that vertices and edges are encoded in two lists, each list containing a Python dict for each vertex and each edge, respectively. A distinguished element of the vertex dicts contain a vertex ID which is used in the edge dicts to refer to source and target vertices. All the remaining elements of the dict are considered vertex and edge attributes. Note that the implementation does not assume that the objects passed to this method are indeed lists of dicts, but they should be iterable and they should yield objects that behave as dicts. So, for instance, a database query result is likely to be fit as long as it's iterable and yields dict-like objects with every iteration.

Parameters

`vertices:`	the data source for the vertices or `None` if there are no special attributes assigned to vertices and we should simply use the edge list of dicts to infer vertex names.
`edges:`	the data source for the edges.
`directed:`	whether the constructed graph will be directed
`vertex_name_attr:`	the name of the distinguished key in the dicts in the vertex data source that contains the vertex names. Ignored if `vertices` is `None`.
`edge_foreign_keys:`	the name of the attributes in the dicts in the edge data source that contain the source and target vertex names.
`iterative:`	whether to add the edges to the graph one by one, iteratively, or to build a large edge list first and use that to construct the graph. The latter approach is faster but it may not be suitable if your dataset is large. The default is to add the edges in a batch from an edge list.

Return Value

the graph that was constructed

TupleList(*klass*, *edges*, *directed*=`False`, *vertex_name_attr*=`'name'`, *edge_attrs*=`None`, *weights*=`False`)

Constructs a graph from a list-of-tuples representation.

This representation assumes that the edges of the graph are encoded in a list of tuples (or lists). Each item in the list must have at least two elements, which specify the source and the target vertices of the edge. The remaining elements (if any) specify the edge attributes of that edge, where the names of the edge attributes originate from the `edge_attrs` list. The names of the vertices will be stored in the vertex attribute given by `vertex_name_attr`.

The default parameters of this function are suitable for creating unweighted graphs from lists where each item contains the source vertex and the target vertex. If you have a weighted graph, you can use items where the third item contains the weight of the edge by setting `edge_attrs` to `"weight"` or `["weight"]`. If you have even more edge attributes, add them to the end of each item in the `edges` list and also specify the corresponding edge attribute names in `edge_attrs` as a list.

Parameters

`edges`:	the data source for the edges. This must be a list where each item is a tuple (or list) containing at least two items: the name of the source and the target vertex. Note that names will be assigned to the `name` vertex attribute (or another vertex attribute if `vertex_name_attr` is specified), even if all the vertex names in the list are in fact numbers.
`directed`:	whether the constructed graph will be directed
`vertex_name_attr`:	the name of the vertex attribute that will contain the vertex names.
`edge_attrs`:	the names of the edge attributes that are filled with the extra items in the edge list (starting from index 2, since the first two items are the source and target vertices). `None` means that only the source and target vertices will be extracted from each item. If you pass a string here, it will be wrapped in a list for convenience.
`weights`:	alternative way to specify that the graph is weighted. If you set `weights` to `true` and `edge_attrs` is not given, it will be assumed that `edge_attrs` is `["weight"]` and igraph will parse the third element from each item into an edge weight. If you set `weights` to a string, it will be assumed that `edge_attrs` contains that string only, and igraph will store the edge weights in that attribute.

Return Value

Formula(*Graph*, *formula*=`None`, *attr*=`"name"`, *simplify*=`True`)

Generates a graph from a graph formula

A graph formula is a simple string representation of a graph. It is very handy for creating small graphs quickly. The string consists of vertex names separated by edge operators. An edge operator is a sequence of dashes (`-`) that may or may not start with an arrowhead (`<` at the beginning of the sequence or `>` at the end of the sequence). The edge operators can be arbitrarily long, i.e., you may use as many dashes to draw them as you like. This makes a total of four different edge operators:

- `---` makes an undirected edge
- `<--` makes a directed edge pointing from the vertex on the right hand side of the operator to the vertex on the left hand side
- `-->` is the opposite of `<--`
- `<-->` creates a mutual directed edge pair between the two vertices

If you only use the undirected edge operator (`---`), the graph will be undirected. Otherwise it will be directed. Vertex names used in the formula will be assigned to the `name` vertex attribute of the graph.

Some simple examples:

```
>>> from igraph import Graph
>>> print Graph.Formula()             # empty graph
IGRAPH UN-- 0 0 --
+ attr: name (v)
>>> g = Graph.Formula("A-B")          # undirected graph
>>> g.vs["name"]
['A', 'B']
>>> print g
IGRAPH UN-- 2 1 --
+ attr: name (v)
+ edges (vertex names):
A--B
>>> g.get_edgelist()
[(0, 1)]
>>> g2 = Graph.Formula("A-----------B")
>>> g2.isomorphic(g)
True
>>> g = Graph.Formula("A  --->  B") # directed graph
>>> g.vs["name"]
['A', 'B']
>>> print g
IGRAPH DN-- 2 1 --
+ attr: name (v)
+ edges (vertex names):
A->B
```

If you have may disconnected componnets, you can separate them with commas. You can also specify isolated vertices:

Bipartite(*types*, *edges*, *directed*=`False`)

Creates a bipartite graph with the given vertex types and edges. This is similar to the default constructor of the graph, the only difference is that it checks whether all the edges go between the two vertex classes and it assigns the type vector to a `type` attribute afterwards.

Examples:

```
>>> g = Graph.Bipartite([0, 1, 0, 1], [(0, 1), (2, 3), (0, 3)])
>>> g.is_bipartite()
True
>>> g.vs["type"]
[False, True, False, True]
```

Parameters

`types`: the vertex types as a boolean list. Anything that evaluates to `False` will denote a vertex of the first kind, anything that evaluates to `True` will denote a vertex of the second kind.

`edges`: the edges as a list of tuples.

`directed`: whether to create a directed graph. Bipartite networks are usually undirected, so the default is `False`

Return Value

the graph with a binary vertex attribute named `"type"` that stores the vertex classes.

Full_Bipartite(*n1*, *n2*, *directed*=`False`, *mode*=`ALL`)

Generates a full bipartite graph (directed or undirected, with or without loops).

```
>>> g = Graph.Full_Bipartite(2, 3)
>>> g.is_bipartite()
True
>>> g.vs["type"]
[False, False, True, True, True]
```

Parameters

`n1`: the number of vertices of the first kind.

`n2`: the number of vertices of the second kind.

`directed`: whether tp generate a directed graph.

`mode`: if `OUT`, then all vertices of the first kind are connected to the others; `IN` specifies the opposite direction, `ALL` creates mutual edges. Ignored for undirected graphs.

Return Value

the graph with a binary vertex attribute named `"type"` that stores the vertex classes.

Random_Bipartite(*n1*, *n2*, *p*=`None`, *m*=`None`, *directed*=`False`, *neimode*=`ALL`)

Generates a random bipartite graph with the given number of vertices and edges (if m is given), or with the given number of vertices and the given connection probability (if p is given).

If m is given but p is not, the generated graph will have n1 vertices of type 1, n2 vertices of type 2 and m randomly selected edges between them. If p is given but m is not, the generated graph will have n1 vertices of type 1 and n2 vertices of type 2, and each edge will exist between them with probability p.

Parameters

`n1:` the number of vertices of type 1.

`n2:` the number of vertices of type 2.

`p:` the probability of edges. If given, `m` must be missing.

`m:` the number of edges. If given, `p` must be missing.

`directed:` whether to generate a directed graph.

`neimode:` if the graph is directed, specifies how the edges will be generated. If it is `"all"`, edges will be generated in both directions (from type 1 to type 2 and vice versa) independently. If it is `"out"` edges will always point from type 1 to type 2. If it is `"in"`, edges will always point from type 2 to type 1. This argument is ignored for undirected graphs.

GRG(*n*, *radius*, *torus*=`False`, *return_coordinates*=`False`)

Generates a random geometric graph.

The algorithm drops the vertices randomly on the 2D unit square and connects them if they are closer to each other than the given radius. The coordinates of the vertices are stored in the vertex attributes `x` and `y`.

Parameters

`n:` The number of vertices in the graph

`radius:` The given radius

`torus:` This should be `True` if we want to use a torus instead of a square.

Incidence(*matrix*, *directed*=`False`, *mode*=`ALL`, *multiple*=`False`)

Creates a bipartite graph from an incidence matrix.

Example:

```
>>> g = Graph.Incidence([[0, 1, 1], [1, 1, 0]])
```

Parameters

`matrix:` the incidence matrix.

`directed:` whether to create a directed graph.

`mode:` defines the direction of edges in the graph. If `OUT`, then edges go from vertices of the first kind (corresponding to rows of the matrix) to vertices of the second kind (the columns of the matrix). If `IN`, the opposite direction is used. `ALL` creates mutual edges. Ignored for undirected graphs.

`multiple:` defines what to do with non-zero entries in the matrix. If `False`, non-zero entries will create an edge no matter what the value is. If `True`, non-zero entries are rounded up to the nearest integer and this will be the number of multiple edges created.

`weighted:` defines whether to create a weighted graph from the incidence matrix. If it is c{None} then an unweighted graph is created and the multiple argument is used to determine the edges of the graph. If it is a string then for every non-zero matrix entry, an edge is created and the value of the entry is added as an edge attribute named by the weighted argument. If it is `True` then a weighted graph is created and the name of the edge attribute will be 'weight'.

Return Value

the graph with a binary vertex attribute named `"type"` that stores the vertex classes.

Raises

`ValueError` if the weighted and multiple are passed together.

DataFrame(*directed*=`True`, *vertices*=`None`)

Generates a graph from one or two dataframes.

Parameters

`edges:` pandas DataFrame containing edges and metadata

`directed:` bool setting whether the graph is directed

`vertices:` None (default) or pandas DataFrame containing vertex metadata. The first column must contain the unique ids of the vertices and will be set as attribute 'name'. All other columns will be added as vertex attributes by column name.

Return Value

the graph

bipartite_projection(*self*, *types*='type', *multiplicity*=True, *probe1*=-1, *which*='both')

Projects a bipartite graph into two one-mode graphs. Edge directions are ignored while projecting.

Examples:

```
>>> g = Graph.Full_Bipartite(10, 5)
>>> g1, g2 = g.bipartite_projection()
>>> g1.isomorphic(Graph.Full(10))
True
>>> g2.isomorphic(Graph.Full(5))
True
```

Parameters

`types`: an igraph vector containing the vertex types, or an attribute name. Anything that evalulates to `False` corresponds to vertices of the first kind, everything else to the second kind.

`multiplicity`: if `True`, then igraph keeps the multiplicity of the edges in the projection in an edge attribute called `"weight"`. E.g., if there is an A-C-B and an A-D-B triplet in the bipartite graph and there is no other X (apart from X=B and X=D) for which an A-X-B triplet would exist in the bipartite graph, the multiplicity of the A-B edge in the projection will be 2.

`probe1`: this argument can be used to specify the order of the projections in the resulting list. If given and non-negative, then it is considered as a vertex ID; the projection containing the vertex will be the first one in the result.

`which`: this argument can be used to specify which of the two projections should be returned if only one of them is needed. Passing 0 here means that only the first projection is returned, while 1 means that only the second projection is returned. (Note that we use 0 and 1 because Python indexing is zero-based). `False` is equivalent to 0 and `True` is equivalent to 1. Any other value means that both projections will be returned in a tuple.

Return Value

a tuple containing the two projected one-mode graphs if `which` is not 1 or 2, or the projected one-mode graph specified by the `which` argument if its value is 0, 1, `False` or `True`.

Overrides: igraph.GraphBase.bipartite_projection

bipartite_projection_size(*types*="type")

Calculates the number of vertices and edges in the bipartite projections of this graph according to the specified vertex types. This is useful if you have a bipartite graph and you want to estimate the amount of memory you would need to calculate the projections themselves.

Parameters

`types`: an igraph vector containing the vertex types, or an attribute name. Anything that evalulates to `False` corresponds to vertices of the first kind, everything else to the second kind.

Return Value

a 4-tuple containing the number of vertices and edges in the first projection, followed by the number of vertices and edges in the second projection.

Overrides: igraph.GraphBase.bipartite_projection_size

get_incidence(*self*, *types*="type")

Returns the incidence matrix of a bipartite graph. The incidence matrix is an n times m matrix, where n and m are the number of vertices in the two vertex classes.

Parameters

`types`: an igraph vector containing the vertex types, or an attribute name. Anything that evalulates to `False` corresponds to vertices of the first kind, everything else to the second kind.

Return Value

the incidence matrix and two lists in a triplet. The first list defines the mapping between row indices of the matrix and the original vertex IDs. The second list is the same for the column indices.

Overrides: igraph.GraphBase.get_incidence

dfs(*self*, *vid*, *mode*=1)

Conducts a depth first search (DFS) on the graph.

Parameters

`vid`: the root vertex ID

`mode`: either `IN` or `OUT` or `ALL`, ignored for undirected graphs.

Return Value

a tuple with the following items:

- The vertex IDs visited (in order)
- The parent of every vertex in the DFS

__iadd__(*self*, *other*)

In-place addition (disjoint union).

See Also: `__add__`

__add__(*self*, *other*)

Copies the graph and extends the copy depending on the type of the other object given.

Parameters

`other`: if it is an integer, the copy is extended by the given number of vertices. If it is a string, the copy is extended by a single vertex whose `name` attribute will be equal to the given string. If it is a tuple with two elements, the copy is extended by a single edge. If it is a list of tuples, the copy is extended by multiple edges. If it is a `Graph`, a disjoint union is performed.

__and__(*self*, *other*)

Graph intersection operator.

Parameters

`other`: the other graph to take the intersection with.

Return Value

the intersected graph.

__isub__(*self*, *other*)

In-place subtraction (difference).

See Also: `__sub__`

__sub__(*self*, *other*)

Removes the given object(s) from the graph

Parameters

`other`: if it is an integer, removes the vertex with the given ID from the graph (note that the remaining vertices will get re-indexed!). If it is a tuple, removes the given edge. If it is a graph, takes the difference of the two graphs. Accepts lists of integers or lists of tuples as well, but they can't be mixed! Also accepts `Edge` and `EdgeSeq` objects.

__mul__(*self*, *other*)

Copies exact replicas of the original graph an arbitrary number of times.

Parameters

`other`: if it is an integer, multiplies the graph by creating the given number of identical copies and taking the disjoint union of them.

__nonzero__(*self*)

Returns True if the graph has at least one vertex, False otherwise.

__or__(*self*, *other*)

Graph union operator.

Parameters

`other`: the other graph to take the union with.

Return Value

the union graph.

__coerce__(*self*, *other*)

Coercion rules.

This method is needed to allow the graph to react to additions with lists, tuples, integers, strings, vertices, edges and so on.

__reduce__(*self*)

Support for pickling.

Overrides: object.__reduce__

__plot__(*self*, *context*, *bbox*, *palette*, **args*, ***kwds*)

Plots the graph to the given Cairo context in the given bounding box

The visual style of vertices and edges can be modified at three places in the following order of precedence (lower indices override higher indices):

1. Keyword arguments of this function (or of `plot()` which is passed intact to `Graph.__plot__()`.
2. Vertex or edge attributes, specified later in the list of keyword arguments.
3. igraph-wide plotting defaults (see `igraph.config.Configuration`)
4. Built-in defaults.

E.g., if the `vertex_size` keyword attribute is not present, but there exists a vertex attribute named `size`, the sizes of the vertices will be specified by that attribute.

Besides the usual self-explanatory plotting parameters (`context`, `bbox`, `palette`), it accepts the following keyword arguments:

- `autocurve`: whether to use curves instead of straight lines for multiple edges on the graph plot. This argument may be `True` or `False`; when omitted, `True` is assumed for graphs with less than 10.000 edges and `False` otherwise.
- `drawer_factory`: a subclass of `AbstractCairoGraphDrawer` which will be used to draw the graph. You may also provide a function here which takes two arguments: the Cairo context to draw on and a bounding box (an instance of `BoundingBox`). If this keyword argument is missing, igraph will use the default graph drawer which should be suitable for most purposes. It is safe to omit this keyword argument unless you need to use a specific graph drawer.
- `keep_aspect_ratio`: whether to keep the aspect ratio of the layout that igraph calculates to place the nodes. `True` means that the layout will be scaled proportionally to fit into the bounding box where the graph is to be drawn but the aspect ratio will be kept the same (potentially leaving empty space next to, below or above the graph). `False` means that the layout will be scaled independently along the X and Y axis in order to fill the entire bounding box. The default is `False`.
- `layout`: the layout to be used. If not an instance of `Layout`, it will be passed to `Graph.layout` to calculate the layout. Note that if you want a deterministic layout that does not change with every plot, you must either use a deterministic layout function (like `Graph.layout_circle`) or calculate the layout in advance and pass a `Layout` object here.
- `margin`: the top, right, bottom, left margins as a 4-tuple. If it has less than 4 elements or is a single float, the elements will be re-used until the length is at least 4.
- `mark_groups`: whether to highlight some of the vertex groups by colored polygons. This argument can be one of the following:
  - `False`: no groups will be highlighted
  - A dict mapping tuples of vertex indices to color names. The given

__str__(*self*)

Returns a string representation of the graph.

Behind the scenes, this method constructs a `GraphSummary` instance and invokes its `__str__` method with a verbosity of 1 and attribute printing turned off.

See the documentation of `GraphSummary` for more details about the output.

Overrides: object.__str__

summary(*self*, *verbosity*=`0`, *width*=`None`, **args*, ***kwds*)

Returns the summary of the graph.

The output of this method is similar to the output of the `__str__` method. If *verbosity* is zero, only the header line is returned (see `__str__` for more details), otherwise the header line and the edge list is printed.

Behind the scenes, this method constructs a `GraphSummary` object and invokes its `__str__` method.

Parameters

`verbosity`: if zero, only the header line is returned (see `__str__` for more details), otherwise the header line and the full edge list is printed.

`width`: the number of characters to use in one line. If `None`, no limit will be enforced on the line lengths.

Return Value

the summary of the graph.

disjoint_union(*self*, *other*)

Creates the disjoint union of two (or more) graphs.

Parameters

`graphs`: graph or list of graphs to be united with the current one.

Return Value

the disjoint union graph

union(*self*, *other*)

Creates the union of two (or more) graphs.

Parameters

`graphs`: graph or list of graphs to be united with the current one.

`byname`: whether to use vertex names instead of ids. See `igraph.union` for details.

Return Value

the union graph

intersection(*self*, *other*)

Creates the intersection of two (or more) graphs.

Parameters

`other`: graph or list of graphs to be intersected with the current one.

`byname`: whether to use vertex names instead of ids. See `igraph.intersection` for details.

Return Value

the intersection graph

layout_fruchterman_reingold_3d(**args*, ***kwds*)

Alias for `layout_fruchterman_reingold()` with dim=3.

See Also: Graph.layout_fruchterman_reingold()

layout_kamada_kawai_3d(**args*, ***kwds*)

Alias for `layout_kamada_kawai()` with dim=3.

See Also: Graph.layout_kamada_kawai()

layout_random_3d(**args*, ***kwds*)

Alias for `layout_random()` with dim=3.

See Also: Graph.layout_random()

layout_grid_3d(**args*, ***kwds*)

Alias for `layout_grid()` with dim=3.

See Also: Graph.layout_grid()

layout_sphere(**args*, ***kwds*)

Alias for `layout_circle()` with dim=3.

See Also: Graph.layout_circle()

layout_bipartite(*types*=`"type"`, *hgap*=`1`, *vgap*=`1`, *maxiter*=`100`)

Place the vertices of a bipartite graph in two layers.

The layout is created by placing the vertices in two rows, according to their types. The positions of the vertices within the rows are then optimized to minimize the number of edge crossings using the heuristic used by the Sugiyama layout algorithm.

Parameters

`types`: an igraph vector containing the vertex types, or an attribute name. Anything that evalulates to `False` corresponds to vertices of the first kind, everything else to the second kind.

`hgap`: minimum horizontal gap between vertices in the same layer.

`vgap`: vertical gap between the two layers.

`maxiter`: maximum number of iterations to take in the crossing reduction step. Increase this if you feel that you are getting too many edge crossings.

Return Value

the calculated layout.

Overrides: igraph.GraphBase.layout_bipartite

layout_circle(*dim*=`2`, *order*=`None`)

Places the vertices of the graph uniformly on a circle or a sphere.

Parameters

`dim`: the desired number of dimensions for the layout. dim=2 means a 2D layout, dim=3 means a 3D layout.

`order`: the order in which the vertices are placed along the circle. Not supported when *dim* is not equal to 2.

Return Value

the calculated layout.

Overrides: igraph.GraphBase.layout_circle

layout_davidson_harel(*seed*=`None`, *maxiter*=`10`, *fineiter*=`-1`, *cool_fact*=`0.75`, *weight_node_dist*=`1.0`, *weight_border*=`0.0`, *weight_edge_lengths*=`-1`, *weight_edge_crossings*=`-1`, *weight_node_edge_dist*=`-1`)

Places the vertices on a 2D plane according to the Davidson-Harel layout algorithm.

The algorithm uses simulated annealing and a sophisticated energy function, which is unfortunately hard to parameterize for different graphs. The original publication did not disclose any parameter values, and the ones below were determined by experimentation.

The algorithm consists of two phases: an annealing phase and a fine-tuning phase. There is no simulated annealing in the second phase.

Parameters

`seed`:	if `None`, uses a random starting layout for the algorithm. If a matrix (list of lists), uses the given matrix as the starting position.
`maxiter`:	Number of iterations to perform in the annealing phase.
`fineiter`:	Number of iterations to perform in the fine-tuning phase. Negative numbers set up a reasonable default from the base-2 logarithm of the vertex count, bounded by 10 from above.
`cool_fact`:	Cooling factor of the simulated annealing phase.
`weight_node_dist`:	Weight for the node-node distances in the energy function.
`weight_border`:	Weight for the distance from the border component of the energy function. Zero means that vertices are allowed to sit on the border of the area designated for the layout.
`weight_edge_lengths`:	Weight for the edge length component of the energy function. Negative numbers are replaced by the density of the graph divided by 10.
`weight_edge_crossings`:	Weight for the edge crossing component of the energy function. Negative numbers are replaced by one minus the square root of the density of the graph.
`weight_node_edge_dist`:	Weight for the node-edge distance component of the energy function. Negative numbers are replaced by 0.2 minus 0.2 times the density of the graph.

layout_drl(*weights*=None, *fixed*=None, *seed*=None, *options*=None, *dim*=2)

Places the vertices on a 2D plane or in the 3D space ccording to the DrL layout algorithm.

This is an algorithm suitable for quite large graphs, but it can be surprisingly slow for small ones (where the simpler force-based layouts like `layout_kamada_kawai()` or `layout_fruchterman_reingold()` are more useful.

Parameters

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name.

`seed`: if `None`, uses a random starting layout for the algorithm. If a matrix (list of lists), uses the given matrix as the starting position.

`fixed`: if a seed is given, you can specify some vertices to be kept fixed at their original position in the seed by passing an appropriate list here. The list must have exactly as many items as the number of vertices in the graph. Items of the list that evaluate to `True` denote vertices that will not be moved.

`options`: if you give a string argument here, you can select from five default preset parameterisations: `default`, `coarsen` for a coarser layout, `coarsest` for an even coarser layout, `refine` for refining an existing layout and `final` for finalizing a layout. If you supply an object that is not a string, the DrL layout parameters are retrieved from the respective keys of the object (so it should be a dict or something else that supports the mapping protocol). The following keys can be used:

- `edge_cut`: edge cutting is done in the late stages of the algorithm in order to achieve less dense layouts. Edges are cut if there is a lot of stress on them (a large value in the objective function sum). The edge cutting parameter is a value between 0 and 1 with 0 representing no edge cutting and 1 representing maximal edge cutting.
- `init_iterations`: number of iterations in the initialization phase
- `init_temperature`: start temperature during initialization
- `init_attraction`: attraction during initialization
- `init_damping_mult`: damping multiplier during initialization
- `liquid_iterations`, `liquid_temperature`, `liquid_attraction`, `liquid_damping_mult`: same parameters for the liquid phase
- `expansion_iterations`, `expansion_temperature`,

layout_fruchterman_reingold(*weights*=None, *niter*=500, *seed*=None, *start_temp*=None, *minx*=None, *maxx*=None, *miny*=None, *maxy*=None, *minz*=None, *maxz*=None, *grid*="auto")

Places the vertices on a 2D plane according to the Fruchterman-Reingold algorithm.

This is a force directed layout, see Fruchterman, T. M. J. and Reingold, E. M.: Graph Drawing by Force-directed Placement. Software – Practice and Experience, 21/11, 1129–1164, 1991

Parameters

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name.

`niter`: the number of iterations to perform. The default is 500.

`start_temp`: Real scalar, the start temperature. This is the maximum amount of movement alloved along one axis, within one step, for a vertex. Currently it is decreased linearly to zero during the iteration. The default is the square root of the number of vertices divided by 10.

`minx`: if not `None`, it must be a vector with exactly as many elements as there are vertices in the graph. Each element is a minimum constraint on the X value of the vertex in the layout.

`maxx`: similar to *minx*, but with maximum constraints

`miny`: similar to *minx*, but with the Y coordinates

`maxy`: similar to *maxx*, but with the Y coordinates

`minz`: similar to *minx*, but with the Z coordinates. Use only for 3D layouts (`dim`=3).

`maxz`: similar to *maxx*, but with the Z coordinates. Use only for 3D layouts (`dim`=3).

`seed`: if `None`, uses a random starting layout for the algorithm. If a matrix (list of lists), uses the given matrix as the starting position.

`grid`: whether to use a faster, but less accurate grid-based implementation of the algorithm. `"auto"` decides based on the number of vertices in the graph; a grid will be used if there are at least 1000 vertices. `"grid"` is equivalent to `True`, `"nogrid"` is equivalent to `False`.

Return Value

the calculated layout.

Overrides: igraph.GraphBase.layout_fruchterman_reingold

layout_graphopt(*niter*=500, *node_charge*=0.001, *node_mass*=30, *spring_length*=0, *spring_constant*=1, *max_sa_movement*=5, *seed*=None)

This is a port of the graphopt layout algorithm by Michael Schmuhl. graphopt version 0.4.1 was rewritten in C and the support for layers was removed.

graphopt uses physical analogies for defining attracting and repelling forces among the vertices and then the physical system is simulated until it reaches an equilibrium or the maximal number of iterations is reached.

See *http://www.schmuhl.org/graphopt/* for the original graphopt.

Parameters

`niter`:	the number of iterations to perform. Should be a couple of hundred in general.
`node_charge`:	the charge of the vertices, used to calculate electric repulsion.
`node_mass`:	the mass of the vertices, used for the spring forces
`spring_length`:	the length of the springs
`spring_constant`:	the spring constant
`max_sa_movement`:	the maximum amount of movement allowed in a single step along a single axis.
`seed`:	a matrix containing a seed layout from which the algorithm will be started. If `None`, a random layout will be used.

Return Value

the calculated layout.

Overrides: igraph.GraphBase.layout_graphopt

layout_grid(*width*=0, *height*=0, *dim*=2)

Places the vertices of a graph in a 2D or 3D grid.

Parameters

`width`: the number of vertices in a single row of the layout. Zero or negative numbers mean that the width should be determined automatically.

`height`: the number of vertices in a single column of the layout. Zero or negative numbers mean that the height should be determined automatically. It must not be given if the number of dimensions is 2.

`dim`: the desired number of dimensions for the layout. dim=2 means a 2D layout, dim=3 means a 3D layout.

Return Value

the calculated layout.

Overrides: igraph.GraphBase.layout_grid

layout_kamada_kawai(*maxiter*=1000, *seed*=None, *maxiter*=1000, *epsilon*=0, *kkconst*=None, *minx*=None, *maxx*=None, *miny*=None, *maxy*=None, *minz*=None, *maxz*=None, *dim*=2)

Places the vertices on a plane according to the Kamada-Kawai algorithm.

This is a force directed layout, see Kamada, T. and Kawai, S.: An Algorithm for Drawing General Undirected Graphs. Information Processing Letters, 31/1, 7–15, 1989.

Parameters

`maxiter`: the maximum number of iterations to perform.

`seed`: if `None`, uses a random starting layout for the algorithm. If a matrix (list of lists), uses the given matrix as the starting position.

`epsilon`: quit if the energy of the system changes less than epsilon. See the original paper for details.

`kkconst`: the Kamada-Kawai vertex attraction constant. `None` means the square of the number of vertices.

`minx`: if not `None`, it must be a vector with exactly as many elements as there are vertices in the graph. Each element is a minimum constraint on the X value of the vertex in the layout.

`maxx`: similar to *minx*, but with maximum constraints

`miny`: similar to *minx*, but with the Y coordinates

`maxy`: similar to *maxx*, but with the Y coordinates

`minz`: similar to *minx*, but with the Z coordinates. Use only for 3D layouts (`dim`=3).

`maxz`: similar to *maxx*, but with the Z coordinates. Use only for 3D layouts (`dim`=3).

`dim`: the desired number of dimensions for the layout. dim=2 means a 2D layout, dim=3 means a 3D layout.

Return Value

the calculated layout.

Overrides: igraph.GraphBase.layout_kamada_kawai

layout_lgl(*maxiter*=`150`, *maxdelta*=`-1`, *area*=`-1`, *coolexp*=`1.5`, *repulserad*=`-1`, *cellsize*=`-1`, *root*=`None`)

Places the vertices on a 2D plane according to the Large Graph Layout.

Parameters

`maxiter`: the number of iterations to perform.

`maxdelta`: the maximum distance to move a vertex in an iteration. If negative, defaults to the number of vertices.

`area`: the area of the square on which the vertices will be placed. If negative, defaults to the number of vertices squared.

`coolexp`: the cooling exponent of the simulated annealing.

`repulserad`: determines the radius at which vertex-vertex repulsion cancels out attraction of adjacent vertices. If negative, defaults to *area* times the number of vertices.

`cellsize`: the size of the grid cells. When calculating the repulsion forces, only vertices in the same or neighboring grid cells are taken into account. Defaults to the fourth root of *area*.

`root`: the root vertex, this is placed first, its neighbors in the first iteration, second neighbors in the second, etc. `None` means that a random vertex will be chosen.

Return Value

the calculated layout.

Overrides: igraph.GraphBase.layout_lgl

layout_mds(*dist*=`None`, *dim*=2, *arpack_options*=`None`)

Places the vertices in an Euclidean space with the given number of dimensions using multidimensional scaling.

This layout requires a distance matrix, where the intersection of row i and column j specifies the desired distance between vertex i and vertex j. The algorithm will try to place the vertices in a way that approximates the distance relations prescribed in the distance matrix. igraph uses the classical multidimensional scaling by Torgerson (see reference below).

For unconnected graphs, the method will decompose the graph into weakly connected components and then lay out the components individually using the appropriate parts of the distance matrix.

Parameters

`dist`: the distance matrix. It must be symmetric and the symmetry is not checked – results are unspecified when a non-symmetric distance matrix is used. If this parameter is `None`, the shortest path lengths will be used as distances. Directed graphs are treated as undirected when calculating the shortest path lengths to ensure symmetry.

`dim`: the number of dimensions. For 2D layouts, supply 2 here; for 3D layouts, supply 3.

`arpack_options`: an `ARPACKOptions` object used to fine-tune the ARPACK eigenvector calculation. If omitted, the module-level variable called `arpack_options` is used.

Return Value

the calculated layout.

Overrides: igraph.GraphBase.layout_mds

Reference: Cox & Cox: Multidimensional Scaling (1994), Chapman and Hall, London.

layout_random(*dim*=2)

Places the vertices of the graph randomly.

Parameters

`dim`: the desired number of dimensions for the layout. dim=2 means a 2D layout, dim=3 means a 3D layout.

Return Value

the coordinate pairs in a list.

Overrides: igraph.GraphBase.layout_random

layout_reingold_tilford(*mode*="out", *root*=None, *rootlevel*=None)

Places the vertices on a 2D plane according to the Reingold-Tilford layout algorithm.

This is a tree layout. If the given graph is not a tree, a breadth-first search is executed first to obtain a possible spanning tree.

Parameters

`mode`: specifies which edges to consider when builing the tree. If it is `OUT` then only the outgoing, if it is `IN` then only the incoming edges of a parent are considered. If it is `ALL` then all edges are used (this was the behaviour in igraph 0.5 and before). This parameter also influences how the root vertices are calculated if they are not given. See the *root* parameter.

`root`: the index of the root vertex or root vertices. if this is a non-empty vector then the supplied vertex IDs are used as the roots of the trees (or a single tree if the graph is connected. If this is `None` or an empty list, the root vertices are automatically calculated based on topological sorting, performed with the opposite of the *mode* argument.

`rootlevel`: this argument is useful when drawing forests which are not trees. It specifies the level of the root vertices for every tree in the forest.

Return Value

the calculated layout.

Overrides: igraph.GraphBase.layout_reingold_tilford

See Also: layout_reingold_tilford_circular

Reference: EM Reingold, JS Tilford: *Tidier Drawings of Trees.* IEEE Transactions on Software Engineering 7:22, 223-228, 1981.

layout_reingold_tilford_circular(*mode*="out", *root*=None, *rootlevel*=None)

Circular Reingold-Tilford layout for trees.

This layout is similar to the Reingold-Tilford layout, but the vertices are placed in a circular way, with the root vertex in the center.

See `layout_reingold_tilford` for the explanation of the parameters.

Return Value

the calculated layout.

Overrides: igraph.GraphBase.layout_reingold_tilford_circular

See Also: layout_reingold_tilford

Reference: EM Reingold, JS Tilford: *Tidier Drawings of Trees.* IEEE Transactions on Software Engineering 7:22, 223-228, 1981.

layout_star(*center*=0, *order*=None)

Calculates a star-like layout for the graph.

Parameters

`center`: the ID of the vertex to put in the center

`order`: a numeric vector giving the order of the vertices (including the center vertex!). If it is `None`, the vertices will be placed in increasing vertex ID order.

Return Value

the calculated layout.

Overrides: igraph.GraphBase.layout_star

Inherited from igraph.GraphBase(Section 1.12)

Adjacency(), Asymmetric_Preference(), Atlas(), Barabasi(), De_Bruijn(), Degree_Sequence(), Erdos_Renyi(), Establishment(), Famous(), Forest_Fire(), Full(), Full_Citation(), Growing_Random(), Isoclass(), K_Regular(), Kautz(), LCF(), Lattice(), Preference(), Read_DL(), Read_Edgelist(), Read_GML(), Read_GraphDB(), Read_GraphML(), Read_Lgl(), Read_Ncol(), Read_Pajek(), Recent_Degree(), Ring(), SBM(), Star(), Static_Fitness(), Static_Power_Law(), Tree(), Watts_Strogatz(), Weighted_Adjacency(), __delitem__(), __getitem__(), __invert__(), __new__(), __setitem__(), all_minimal_st_separators(), are_connected(), articulation_points(), assortativity(), assortativity_degree(), assortativity_nominal(), attributes(), authority_score(), average_path_length(), betweenness(), bfs(), bfsiter(), bibcoupling(), bridges(), canonical_permutation(), clique_number(), cliques(), closeness(), cocitation(), complementer(), compose(), constraint(), contract_vertices(), convergence_degree(), convergence_field_size(), copy(), coreness(), count_isomorphisms_vf2(),

count_multiple(), count_subisomorphisms_vf2(), decompose(), degree(), delete_vertices(), density(), dfsiter(), diameter(), difference(), diversity(), dominator(), eccentricity(), ecount(), edge_attributes(), edge_betweenness(), edge_connectivity(), eigen_adjacency(), eigenvector_centrality(), farthest_points(), feedback_arc_set(), get_all_shortest_paths(), get_diameter(), get_edgelist(), get_eid(), get_eids(), get_isomorphisms_vf2(), get_shortest_paths(), get_subisomorphisms_lad(), get_subisomorphisms_vf2(), girth(), has_multiple(), hub_score(), incident(), independence_number(), independent_vertex_sets(), induced_subgraph(), is_bipartite(), is_connected(), is_dag(), is_directed(), is_loop(), is_minimal_separator(), is_multiple(), is_mutual(), is_separator(), is_simple(), isoclass(), isomorphic(), isomorphic_bliss(), isomorphic_vf2(), knn(), laplacian(), largest_cliques(), largest_independent_vertex_sets(), linegraph(), maxdegree(), maxflow_value(), maximal_cliques(), maximal_independent_vertex_sets(), mincut_value(), minimum_size_separators(), motifs_randesu(), motifs_randesu_estimate(), motifs_randesu_no(), neighborhood(), neighborhood_size(), neighbors(), permute_vertices(), personalized_pagerank(), predecessors(), radius(), random_walk(), reciprocity(), rewire(), rewire_edges(), shortest_paths(), similarity_dice(), similarity_inverse_log_weighted(), similarity_jaccard(), simplify(), strength(), subcomponent(), subgraph_edges(), subisomorphic_lad(), subisomorphic_vf2(), successors(), to_directed(), to_prufer(), to_undirected(), topological_sorting(), transitivity_local_undirected(), transitivity_undirected(), unfold_tree(), vcount(), vertex_attributes(), vertex_connectivity(), write_dot(), write_edgelist(), write_gml(), write_graphml(), write_leda(), write_lgl(), write_ncol(), write_pajek()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

1.5.2 Properties

Name	Description
vs	The vertex sequence of the graph
es	The edge sequence of the graph
_as_parameter_	
Inherited from object __class__	

1.5.3 Class Variables

Name	Description
__iter__	**Value:** `None`
__hash__	**Value:** `None`

1.6 Class VertexSeq

object
└
igraph.drawing.graph.VertexSeq
└
igraph.VertexSeq

Class representing a sequence of vertices in the graph.

This class is most easily accessed by the `vs` field of the `Graph` object, which returns an ordered sequence of all vertices in the graph. The vertex sequence can be refined by invoking the `VertexSeq.select()` method. `VertexSeq.select()` can also be accessed by simply calling the `VertexSeq` object.

An alternative way to create a vertex sequence referring to a given graph is to use the constructor directly:

```
>>> g = Graph.Full(3)
>>> vs = VertexSeq(g)
>>> restricted_vs = VertexSeq(g, [0, 1])
```

The individual vertices can be accessed by indexing the vertex sequence object. It can be used as an iterable as well, or even in a list comprehension:

```
>>> g=Graph.Full(3)
>>> for v in g.vs:
...   v["value"] = v.index ** 2
...
>>> [v["value"] ** 0.5 for v in g.vs]
[0.0, 1.0, 2.0]
```

The vertex set can also be used as a dictionary where the keys are the attribute names. The values corresponding to the keys are the values of the given attribute for every vertex selected by the sequence.

```
>>> g=Graph.Full(3)
>>> for idx, v in enumerate(g.vs):
...   v["weight"] = idx*(idx+1)
...
>>> g.vs["weight"]
[0, 2, 6]
>>> g.vs.select(1,2)["weight"] = [10, 20]
>>> g.vs["weight"]
[0, 10, 20]
```

If you specify a sequence that is shorter than the number of vertices in the VertexSeq, the sequence is reused:

```
>>> g = Graph.Tree(7, 2)
>>> g.vs["color"] = ["red", "green"]
>>> g.vs["color"]
['red', 'green', 'red', 'green', 'red', 'green', 'red']
```

You can even pass a single string or integer, it will be considered as a sequence of length 1:

```
>>> g.vs["color"] = "red"
>>> g.vs["color"]
['red', 'red', 'red', 'red', 'red', 'red', 'red']
```

Some methods of the vertex sequences are simply proxy methods to the corresponding methods in the `Graph` object. One such example is `VertexSeq.degree()`:

```
>>> g=Graph.Tree(7, 2)
>>> g.vs.degree()
[2, 3, 3, 1, 1, 1, 1]
>>> g.vs.degree() == g.degree()
True
```

1.6.1 Methods

attributes(*self*)

Returns the list of all the vertex attributes in the graph associated to this vertex sequence.

find(*self*, **args*, ***kwds*)

Returns the first vertex of the vertex sequence that matches some criteria.

The selection criteria are equal to the ones allowed by `VertexSeq.select`. See `VertexSeq.select` for more details.

For instance, to find the first vertex with name `foo` in graph `g`:

```
>>> g.vs.find(name="foo")              #doctest:+SKIP
```

To find an arbitrary isolated vertex:

```
>>> g.vs.find(_degree=0)               #doctest:+SKIP
```

Return Value

Vertex

Overrides: igraph.drawing.graph.VertexSeq.find

select(*self*, **args*, ***kwds*)

Selects a subset of the vertex sequence based on some criteria

The selection criteria can be specified by the positional and the keyword arguments. Positional arguments are always processed before keyword arguments.

- If the first positional argument is `None`, an empty sequence is returned.
- If the first positional argument is a callable object, the object will be called for every vertex in the sequence. If it returns `True`, the vertex will be included, otherwise it will be excluded.
- If the first positional argument is an iterable, it must return integers and they will be considered as indices of the current vertex set (NOT the whole vertex set of the graph – the difference matters when one filters a vertex set that has already been filtered by a previous invocation of `VertexSeq.select()`. In this case, the indices do not refer directly to the vertices of the graph but to the elements of the filtered vertex sequence.
- If the first positional argument is an integer, all remaining arguments are expected to be integers. They are considered as indices of the current vertex set again.

Keyword arguments can be used to filter the vertices based on their attributes. The name of the keyword specifies the name of the attribute and the filtering operator, they should be concatenated by an underscore (`_`) character. Attribute names can also contain underscores, but operator names don't, so the operator is always the largest trailing substring of the keyword name that does not contain an underscore. Possible operators are:

- `eq`: equal to
- `ne`: not equal to
- `lt`: less than
- `gt`: greater than
- `le`: less than or equal to
- `ge`: greater than or equal to
- `in`: checks if the value of an attribute is in a given list
- `notin`: checks if the value of an attribute is not in a given list

For instance, if you want to filter vertices with a numeric `age` property larger than 200, you have to write:

```
>>> g.vs.select(age_gt=200)                    #doctest: +SKIP
```

Similarly, to filter vertices whose `type` is in a list of predefined types:

```
>>> list_of_types = ["HR", "Finance", "Management"]
>>> g.vs.select(type_in=list_of_types)         #doctest: +SKIP
```

If the operator is omitted, it defaults to `eq`. For instance, the following selector selects vertices whose `cluster` property equals to 2:

__call__(*self*, **args*, ***kwds*)

Shorthand notation to select()

This method simply passes all its arguments to `VertexSeq.select()`.

betweenness(**args*, ***kwds*)

Proxy method to `Graph.betweenness()`

This method calls the `betweenness()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.betweenness() for details.

bibcoupling(**args*, ***kwds*)

Proxy method to `Graph.bibcoupling()`

This method calls the `bibcoupling()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.bibcoupling() for details.

closeness(**args*, ***kwds*)

Proxy method to `Graph.closeness()`

This method calls the `closeness()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.closeness() for details.

cocitation(**args*, ***kwds*)

Proxy method to `Graph.cocitation()`

This method calls the `cocitation()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.cocitation() for details.

constraint(**args*, ***kwds*)

Proxy method to `Graph.constraint()`

This method calls the `constraint()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.constraint() for details.

degree(**args*, ***kwds*)

Proxy method to `Graph.degree()`

This method calls the `degree()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.degree() for details.

delete(**args*, ***kwds*)

Proxy method to `Graph.delete_vertices()`

This method calls the `delete_vertices()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.delete_vertices() for details.

diversity(**args*, ***kwds*)

Proxy method to `Graph.diversity()`

This method calls the `diversity()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.diversity() for details.

eccentricity(**args*, ***kwds*)

Proxy method to `Graph.eccentricity()`

This method calls the `eccentricity()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.eccentricity() for details.

get_shortest_paths(**args*, ***kwds*)

Proxy method to `Graph.get_shortest_paths()`

This method calls the `get_shortest_paths()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.get_shortest_paths() for details.

indegree(**args*, ***kwds*)

Proxy method to `Graph.indegree()`

This method calls the `indegree()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.indegree() for details.

is_minimal_separator(**args*, ***kwds*)

Proxy method to `Graph.is_minimal_separator()`

This method calls the `is_minimal_separator()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.is_minimal_separator() for details.

is_separator(**args*, ***kwds*)

Proxy method to `Graph.is_separator()`

This method calls the `is_separator()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.is_separator() for details.

isoclass(**args*, ***kwds*)

Proxy method to `Graph.isoclass()`

This method calls the `isoclass()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.isoclass() for details.

maxdegree(**args*, ***kwds*)

Proxy method to `Graph.maxdegree()`

This method calls the `maxdegree()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.maxdegree() for details.

outdegree(**args*, ***kwds*)

Proxy method to `Graph.outdegree()`

This method calls the `outdegree()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.outdegree() for details.

pagerank(**args*, ***kwds*)

Proxy method to `Graph.pagerank()`

This method calls the `pagerank()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.pagerank() for details.

personalized_pagerank(**args*, ***kwds*)

Proxy method to `Graph.personalized_pagerank()`

This method calls the `personalized_pagerank()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.personalized_pagerank() for details.

shortest_paths(**args*, ***kwds*)

Proxy method to `Graph.shortest_paths()`

This method calls the `shortest_paths()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.shortest_paths() for details.

similarity_dice(**args*, ***kwds*)

Proxy method to `Graph.similarity_dice()`

This method calls the `similarity_dice()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.similarity_dice() for details.

similarity_jaccard(**args*, ***kwds*)

Proxy method to `Graph.similarity_jaccard()`

This method calls the `similarity_jaccard()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.similarity_jaccard() for details.

subgraph(**args*, ***kwds*)

Proxy method to `Graph.subgraph()`

This method calls the `subgraph()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.subgraph() for details.

Inherited from igraph.drawing.graph.VertexSeq

__delitem__(), __getitem__(), __init__(), __len__(), __new__(), __setitem__(), attribute_names(), get_attribute_values(), set_attribute_values()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

1.6.2 Properties

Name	Description
Inherited from igraph.drawing.graph.VertexSeq graph, indices	
Inherited from object __class__	

1.7 Class EdgeSeq

Class representing a sequence of edges in the graph.

This class is most easily accessed by the `es` field of the `Graph` object, which returns an ordered sequence of all edges in the graph. The edge sequence can be refined by invoking

the `EdgeSeq.select()` method. `EdgeSeq.select()` can also be accessed by simply calling the `EdgeSeq` object.

An alternative way to create an edge sequence referring to a given graph is to use the constructor directly:

```
>>> g = Graph.Full(3)
>>> es = EdgeSeq(g)
>>> restricted_es = EdgeSeq(g, [0, 1])
```

The individual edges can be accessed by indexing the edge sequence object. It can be used as an iterable as well, or even in a list comprehension:

```
>>> g=Graph.Full(3)
>>> for e in g.es:
...   print e.tuple
...
(0, 1)
(0, 2)
(1, 2)
>>> [max(e.tuple) for e in g.es]
[1, 2, 2]
```

The edge sequence can also be used as a dictionary where the keys are the attribute names. The values corresponding to the keys are the values of the given attribute of every edge in the graph:

```
>>> g=Graph.Full(3)
>>> for idx, e in enumerate(g.es):
...   e["weight"] = idx*(idx+1)
...
>>> g.es["weight"]
[0, 2, 6]
>>> g.es["weight"] = range(3)
>>> g.es["weight"]
[0, 1, 2]
```

If you specify a sequence that is shorter than the number of edges in the EdgeSeq, the sequence is reused:

```
>>> g = Graph.Tree(7, 2)
>>> g.es["color"] = ["red", "green"]
>>> g.es["color"]
['red', 'green', 'red', 'green', 'red', 'green']
```

You can even pass a single string or integer, it will be considered as a sequence of length 1:

```
>>> g.es["color"] = "red"
```

```
>>> g.es["color"]
['red', 'red', 'red', 'red', 'red', 'red']
```

Some methods of the edge sequences are simply proxy methods to the corresponding methods in the `Graph` object. One such example is `EdgeSeq.is_multiple()`:

```
>>> g=Graph(3, [(0,1), (1,0), (1,2)])
>>> g.es.is_multiple()
[False, True, False]
>>> g.es.is_multiple() == g.is_multiple()
True
```

1.7.1 Methods

attributes(*self*)

Returns the list of all the edge attributes in the graph associated to this edge sequence.

find(*self*, **args*, ***kwds*)

Returns the first edge of the edge sequence that matches some criteria.

The selection criteria are equal to the ones allowed by `VertexSeq.select`. See `VertexSeq.select` for more details.

For instance, to find the first edge with weight larger than 5 in graph `g`:

```
>>> g.es.find(weight_gt=5)              #doctest:+SKIP
```

Return Value

Edge

Overrides: igraph._igraph.EdgeSeq.find

select(*self*, **args*, ***kwds*)

Selects a subset of the edge sequence based on some criteria

The selection criteria can be specified by the positional and the keyword arguments. Positional arguments are always processed before keyword arguments.

- If the first positional argument is `None`, an empty sequence is returned.
- If the first positional argument is a callable object, the object will be called for every edge in the sequence. If it returns `True`, the edge will be included, otherwise it will be excluded.
- If the first positional argument is an iterable, it must return integers and they will be considered as indices of the current edge set (NOT the whole edge set of the graph – the difference matters when one filters an edge set that has already been filtered by a previous invocation of `EdgeSeq.select()`. In this case, the indices do not refer directly to the edges of the graph but to the elements of the filtered edge sequence.
- If the first positional argument is an integer, all remaining arguments are expected to be integers. They are considered as indices of the current edge set again.

Keyword arguments can be used to filter the edges based on their attributes and properties. The name of the keyword specifies the name of the attribute and the filtering operator, they should be concatenated by an underscore (`_`) character. Attribute names can also contain underscores, but operator names don't, so the operator is always the largest trailing substring of the keyword name that does not contain an underscore. Possible operators are:

- `eq`: equal to
- `ne`: not equal to
- `lt`: less than
- `gt`: greater than
- `le`: less than or equal to
- `ge`: greater than or equal to
- `in`: checks if the value of an attribute is in a given list
- `notin`: checks if the value of an attribute is not in a given list

For instance, if you want to filter edges with a numeric `weight` property larger than 50, you have to write:

```
>>> g.es.select(weight_gt=50)              #doctest: +SKIP
```

Similarly, to filter edges whose `type` is in a list of predefined types:

```
>>> list_of_types = ["inhibitory", "excitatory"]
>>> g.es.select(type_in=list_of_types)     #doctest: +SKIP
```

If the operator is omitted, it defaults to `eq`. For instance, the following selector selects edges whose `type` property is `intracluster`:

__call__(*self*, **args*, ***kwds*)

Shorthand notation to select()

This method simply passes all its arguments to `EdgeSeq.select()`.

count_multiple(**args*, ***kwds*)

Proxy method to `Graph.count_multiple()`

This method calls the `count_multiple()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.count_multiple() for details.

delete(**args*, ***kwds*)

Proxy method to `Graph.delete_edges()`

This method calls the `delete_edges()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.delete_edges() for details.

edge_betweenness(**args*, ***kwds*)

Proxy method to `Graph.edge_betweenness()`

This method calls the `edge_betweenness()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.edge_betweenness() for details.

is_loop(**args*, ***kwds*)

Proxy method to `Graph.is_loop()`

This method calls the `is_loop()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.is_loop() for details.

is_multiple(**args*, ***kwds*)

Proxy method to `Graph.is_multiple()`

This method calls the `is_multiple()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.is_multiple() for details.

is_mutual(**args*, ***kwds*)

Proxy method to `Graph.is_mutual()`

This method calls the `is_mutual()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.is_mutual() for details.

subgraph(**args*, ***kwds*)

Proxy method to `Graph.subgraph_edges()`

This method calls the `subgraph_edges()` method of the `Graph` class restricted to this sequence, and returns the result.

See Also: Graph.subgraph_edges() for details.

Inherited from igraph._ igraph.EdgeSeq

__delitem__(), __getitem__(), __init__(), __len__(), __new__(), __setitem__(), attribute_names(), get_attribute_values(), is_all(), set_attribute_values()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

1.7.2 Properties

Name	Description
Inherited from igraph._ igraph.EdgeSeq	
graph, indices	
Inherited from object	
__class__	

1.8 Class ARPACKOptions

object
└ **igraph.ARPACKOptions**

Class representing the parameters of the ARPACK module.

ARPACK is a Fortran implementation of the implicitly restarted Arnoldi method, an algorithm for calculating some of the eigenvalues and eigenvectors of a given matrix. igraph uses this package occasionally, and this class can be used to fine-tune the behaviour of ARPACK in such cases.

The class has several attributes which are not documented here, since they are usually of marginal use to the ordinary user. See the source code of the original ARPACK Fortran package (especially the file `dsaupd.f`) for a detailed explanation of the parameters. Only the most basic attributes are explained here. Most of them are read only unless stated otherwise.

- `bmat`: type of the eigenproblem solved. `'I'` means standard eigenproblem (A*x = lambda*x), `'G'` means generalized eigenproblem (A*x = lambda*B*x).
- `n`: dimension of the eigenproblem
- `tol`: precision. If less than or equal to zero, the standard machine precision is used as computed by the LAPACK utility called `dlamch`. This can be modified.
- `mxiter`: maximum number of update iterations to take. This can be modified. You can also use `maxiter`.
- `iter`: actual number of update iterations taken
- `numop`: total number of OP*x operations
- `numopb`: total number of B*x operations if `bmat` is `'G'`
- `numreo`: total number of steps of re-orthogonalization

1.8.1 Methods

__new__(*T*, *S*, ...)

Return Value

a new object with type S, a subtype of T

Overrides: object.__new__

__str__(*x*)

str(x)

Overrides: object.__str__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

1.8.2 Properties

Name	Description
Inherited from object	
__class__	

1.9 Class BFSIter

object
└ **igraph.BFSIter**

igraph BFS iterator object

1.9.1 Methods

__iter__(*x*)

iter(x)

next(*x*)

Return Value

the next value, or raise StopIteration

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

1.9.2 Properties

Name	Description
Inherited from object	
__class__	

1.10 Class DFSIter

object
└ **igraph.DFSIter**

igraph DFS iterator object

1.10.1 Methods

__iter__(*x*)

iter(x)

next(*x*)

Return Value

the next value, or raise StopIteration

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

1.10.2 Properties

Name	Description
Inherited from object	
__class__	

1.11 Class Edge

object
└
igraph.Edge

Class representing a single edge in a graph.

The edge is referenced by its index, so if the underlying graph changes, the semantics of the edge object might change as well (if the edge indices are altered in the original graph).

The attributes of the edge can be accessed by using the edge as a hash:

```
>>> e["weight"] = 2                    #doctest: +SKIP
>>> print e["weight"]                 #doctest: +SKIP
2
```

1.11.1 Methods

__delitem__(*x*, *y*)

del x[y]

__eq__(*x*, *y*)

x==y

__ge__(*x*, *y*)

x>=y

__getitem__(*x*, *y*)

x[y]

__gt__(*x*, *y*)

x>y

__hash__(*x*)

hash(x)

Overrides: object.__hash__

__le__(*x*, *y*)

x<=y

__len__(*x*)

len(x)

__lt__(*x*, *y*)

x<y

__ne__(*x*, *y*)

x!=y

__repr__(*x*)

repr(x)

Overrides: object.__repr__

__setitem__(*x*, *i*, *y*)

x[i]=y

attribute_names()

Returns the list of edge attribute names

Return Value

list

attributes()

Returns a dict of attribute names and values for the edge

Return Value

dict

count_multiple(...)

Proxy method to `Graph.count_multiple()`

This method calls the count_multiple method of the `Graph` class with this edge as the first argument, and returns the result.

See Also: Graph.count_multiple() for details.

delete(...)

Proxy method to `Graph.delete_edges()`

This method calls the delete_edges method of the `Graph` class with this edge as the first argument, and returns the result.

See Also: Graph.delete_edges() for details.

is_loop(...)

Proxy method to `Graph.is_loop()`

This method calls the is_loop method of the `Graph` class with this edge as the first argument, and returns the result.

See Also: Graph.is_loop() for details.

is_multiple(...)

Proxy method to `Graph.is_multiple()`

This method calls the is_multiple method of the `Graph` class with this edge as the first argument, and returns the result.

See Also: Graph.is_multiple() for details.

is_mutual(...)

Proxy method to `Graph.is_mutual()`

This method calls the is_mutual method of the `Graph` class with this edge as the first argument, and returns the result.

See Also: Graph.is_mutual() for details.

update_attributes(*E*, ***F*)

Updates the attributes of the edge from dict/iterable E and F.

If E has a `keys()` method, it does: `for k in E: self[k] = E[k]`. If E lacks a `keys()` method, it does: `for (k, v) in E: self[k] = v`. In either case, this is followed by: `for k in F: self[k] = F[k]`.

This method thus behaves similarly to the `update()` method of Python dictionaries.

Return Value

None

Inherited from object

__delattr__(), __format__(), __getattribute__(), __init__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

1.11.2 Properties

Name	Description
graph	The graph the edge belongs to
index	Index of this edge
source	Source vertex index of this edge
source_vertex	Source vertex of this edge
target	Target vertex index of this edge
target_vertex	Target vertex of this edge
tuple	Source and target vertex index of this edge as a tuple
vertex_tuple	Source and target vertex of this edge as a tuple
Inherited from object	
__class__	

1.12 Class GraphBase

object
└ **igraph.GraphBase**

Known Subclasses: igraph.Graph

Low-level representation of a graph.

Don't use it directly, use `igraph.Graph` instead.

1.12.1 Methods

Adjacency(*matrix*, *mode*=ADJ_DIRECTED)

Generates a graph from its adjacency matrix.

Parameters

`matrix`: the adjacency matrix

`mode`: the mode to be used. Possible values are:

- `ADJ_DIRECTED` - the graph will be directed and a matrix element gives the number of edges between two vertex.
- `ADJ_UNDIRECTED` - alias to `ADJ_MAX` for convenience.
- `ADJ_MAX` - undirected graph will be created and the number of edges between vertex i and j is *max(A(i,j), A(j,i))*
- `ADJ_MIN` - like `ADJ_MAX`, but with *min(A(i,j), A(j,i))*
- `ADJ_PLUS` - like `ADJ_MAX`, but with *A(i,j) + A(j,i)*
- `ADJ_UPPER` - undirected graph with the upper right triangle of the matrix (including the diagonal)
- `ADJ_LOWER` - undirected graph with the lower left triangle of the matrix (including the diagonal)

These values can also be given as strings without the `ADJ` prefix.

Asymmetric_Preference(*n*, *type_dist_matrix*, *pref_matrix*, *attribute*=`None`, *loops*=`False`)

Generates a graph based on asymmetric vertex types and connection probabilities.

This is the asymmetric variant of `Graph.Preference`. A given number of vertices are generated. Every vertex is assigned to an "incoming" and an "outgoing" vertex type according to the given joint type probabilities. Finally, every vertex pair is evaluated and a directed edge is created between them with a probability depending on the "outgoing" type of the source vertex and the "incoming" type of the target vertex.

Parameters

`n:`	the number of vertices in the graph
`type_dist_matrix:`	matrix giving the joint distribution of vertex types
`pref_matrix:`	matrix giving the connection probabilities for different vertex types.
`attribute:`	the vertex attribute name used to store the vertex types. If `None`, vertex types are not stored.
`loops:`	whether loop edges are allowed.

Atlas(*idx*)

Generates a graph from the Graph Atlas.

Parameters

`idx:` The index of the graph to be generated. Indices start from zero, graphs are listed:

1. in increasing order of number of vertices;
2. for a fixed number of vertices, in increasing order of the number of edges;
3. for fixed numbers of vertices and edges, in increasing order of the degree sequence, for example 111223 $<$ 112222;
4. for fixed degree sequence, in increasing number of automorphisms.

Reference: *An Atlas of Graphs* by Ronald C. Read and Robin J. Wilson, Oxford University Press, 1998.

Barabasi(*n*, *m*, *outpref*=`False`, *directed*=`False`, *power*=`1`, *zero_ appeal*=`1`, *implementation*=`"psumtree"`, *start_ from*=`None`)

Generates a graph based on the Barabasi-Albert model.

Parameters

`n:` the number of vertices

`m:` either the number of outgoing edges generated for each vertex or a list containing the number of outgoing edges for each vertex explicitly.

`outpref:` `True` if the out-degree of a given vertex should also increase its citation probability (as well as its in-degree), but it defaults to `False`.

`directed:` `True` if the generated graph should be directed (default: `False`).

`power:` the power constant of the nonlinear model. It can be omitted, and in this case the usual linear model will be used.

`zero_appeal:` the attractivity of vertices with degree zero.

`implementation:` the algorithm to use to generate the network. Possible values are:

- `"bag"`: the algorithm that was the default in igraph before 0.6. It works by putting the ids of the vertices into a bag (multiset) exactly as many times as their in-degree, plus once more. The required number of cited vertices are then drawn from the bag with replacement. It works only for *power*=1 and *zero_ appeal*=1.
- `"psumtree"`: this algorithm uses a partial prefix-sum tree to generate the graph. It does not generate multiple edges and it works for any values of *power* and *zero_ appeal*.
- `"psumtree_multiple"`: similar to `"psumtree"`, but it will generate multiple edges as well. igraph before 0.6 used this algorithm for *power*s other than 1.

`start_from:` if given and not `None`, this must be another `Graph` object. igraph will use this graph as a starting point for the preferential attachment model.

Reference: Barabasi, A-L and Albert, R. 1999. Emergence of scaling in random networks. Science, 286 509-512.

De_Bruijn(*m*, *n*)

Generates a de Bruijn graph with parameters (m, n)

A de Bruijn graph represents relationships between strings. An alphabet of m letters are used and strings of length n are considered. A vertex corresponds to every possible string and there is a directed edge from vertex v to vertex w if the string of v can be transformed into the string of w by removing its first letter and appending a letter to it.

Please note that the graph will have $m \hat{} n$ vertices and even more edges, so probably you don't want to supply too big numbers for m and n.

Parameters

`m`: the size of the alphabet

`n`: the length of the strings

Degree_Sequence(*out*, *in*=`None`, *method*=`"simple"`)

Generates a graph with a given degree sequence.

Parameters

`out`: the out-degree sequence for a directed graph. If the in-degree sequence is omitted, the generated graph will be undirected, so this will be the in-degree sequence as well

`in`: the in-degree sequence for a directed graph. If omitted, the generated graph will be undirected.

`method`: the generation method to be used. One of the following:

- `"simple"` – simple generator that sometimes generates loop edges and multiple edges. The generated graph is not guaranteed to be connected.
- `"no_multiple"` – similar to `"simple"` but avoids the generation of multiple and loop edges at the expense of increased time complexity. The method will re-start the generation every time it gets stuck in a configuration where it is not possible to insert any more edges without creating loops or multiple edges, and there is no upper bound on the number of iterations, but it will succeed eventually if the input degree sequence is graphical and throw an exception if the input degree sequence is not graphical.
- `"vl"` – a more sophisticated generator that can sample undirected, connected simple graphs uniformly. It uses Monte-Carlo methods to randomize the graphs. This generator should be favoured if undirected and connected graphs are to be generated and execution time is not a concern. igraph uses the original implementation of Fabien Viger; see the following URL and the paper cited on it for the details of the algorithm: *http://www-rp.lip6.fr/~latapy/FV/generation.html*.

Erdos_Renyi(*n*, *p*, *m*, *directed*=`False`, *loops*=`False`)

Generates a graph based on the Erdos-Renyi model.

Parameters

`n:`	the number of vertices.
`p:`	the probability of edges. If given, `m` must be missing.
`m:`	the number of edges. If given, `p` must be missing.
`directed:`	whether to generate a directed graph.
`loops:`	whether self-loops are allowed.

Establishment(*n*, *k*, *type_dist*, *pref_matrix*, *directed*=`False`)

Generates a graph based on a simple growing model with vertex types.

A single vertex is added at each time step. This new vertex tries to connect to k vertices in the graph. The probability that such a connection is realized depends on the types of the vertices involved.

Parameters

`n:`	the number of vertices in the graph
`k:`	the number of connections tried in each step
`type_dist:`	list giving the distribution of vertex types
`pref_matrix:`	matrix (list of lists) giving the connection probabilities for different vertex types
`directed:`	whether to generate a directed graph.

Famous(*name*)

Generates a famous graph based on its name.

Several famous graphs are known to `igraph` including (but not limited to) the Chvatal graph, the Petersen graph or the Tutte graph. This method generates one of them based on its name (case insensitive). See the documentation of the C interface of `igraph` for the names available: *http://igraph.org/doc/c*.

Parameters

`name:` the name of the graph to be generated.

Forest_Fire(*n*, *fw_prob*, *bw_factor*=`0.0`, *ambs*=`1`, *directed*=`False`)

Generates a graph based on the forest fire model

The forest fire model is a growing graph model. In every time step, a new vertex is added to the graph. The new vertex chooses an ambassador (or more than one if *ambs>1*) and starts a simulated forest fire at its ambassador(s). The fire spreads through the edges. The spreading probability along an edge is given by *fw_prob*. The fire may also spread backwards on an edge by probability *fw_prob * bw_factor*. When the fire ended, the newly added vertex connects to the vertices "burned" in the previous fire.

Parameters

`n:`	the number of vertices in the graph
`fw_prob:`	forward burning probability
`bw_factor:`	ratio of backward and forward burning probability
`ambs:`	number of ambassadors chosen in each step
`directed:`	whether the graph will be directed

Full(*n*, *directed*=`False`, *loops*=`False`)

Generates a full graph (directed or undirected, with or without loops).

Parameters

`n:`	the number of vertices.
`directed:`	whether to generate a directed graph.
`loops:`	whether self-loops are allowed.

Full_Citation(*n*, *directed*=`False`)

Generates a full citation graph

A full citation graph is a graph where the vertices are indexed from 0 to *n-1* and vertex *i* has a directed edge towards all vertices with an index less than *i*.

Parameters

`n:`	the number of vertices.
`directed:`	whether to generate a directed graph.

Growing_Random(*n*, *m*, *directed*=`False`, *citation*=`False`)

Generates a growing random graph.

Parameters

`n:`	The number of vertices in the graph
`m:`	The number of edges to add in each step (after adding a new vertex)
`directed:`	whether the graph should be directed.
`citation:`	whether the new edges should originate from the most recently added vertex.

Isoclass(*n*, *class*, *directed*=`False`)

Generates a graph with a given isomorphism class.

Parameters

`n:`	the number of vertices in the graph (3 or 4)
`class:`	the isomorphism class
`directed:`	whether the graph should be directed.

K_Regular(*n*, *k*, *directed*=`False`, *multiple*=`False`)

Generates a k-regular random graph

A k-regular random graph is a random graph where each vertex has degree k. If the graph is directed, both the in-degree and the out-degree of each vertex will be k.

Parameters

`n:`	The number of vertices in the graph
`k:`	The degree of each vertex if the graph is undirected, or the in-degree and out-degree of each vertex if the graph is directed
`directed:`	whether the graph should be directed.
`multiple:`	whether it is allowed to create multiple edges.

Kautz(*m*, *n*)

Generates a Kautz graph with parameters (m, n)

A Kautz graph is a labeled graph, vertices are labeled by strings of length *n+1* above an alphabet with *m+1* letters, with the restriction that every two consecutive letters in the string must be different. There is a directed edge from a vertex v to another vertex w if it is possible to transform the string of v into the string of w by removing the first letter and appending a letter to it.

Parameters

`m`: the size of the alphabet minus one

`n`: the length of the strings minus one

LCF(*n*, *shifts*, *repeats*)

Generates a graph from LCF notation.

LCF is short for Lederberg-Coxeter-Frucht, it is a concise notation for 3-regular Hamiltonian graphs. It consists of three parameters, the number of vertices in the graph, a list of shifts giving additional edges to a cycle backbone and another integer giving how many times the shifts should be performed. See *http://mathworld.wolfram.com/LCFNotation.html* for details.

Parameters

`n`: the number of vertices

`shifts`: the shifts in a list or tuple

`repeats`: the number of repeats

Lattice(*dim*, *nei*=`1`, *directed*=`False`, *mutual*=`True`, *circular*=`True`)

Generates a regular lattice.

Parameters

`dim`: list with the dimensions of the lattice

`nei`: value giving the distance (number of steps) within which two vertices will be connected.

`directed`: whether to create a directed graph.

`mutual`: whether to create all connections as mutual in case of a directed graph.

`circular`: whether the generated lattice is periodic.

Preference(*n*, *type_dist*, *pref_matrix*, *attribute*=`None`, *directed*=`False`, *loops*=`False`)

Generates a graph based on vertex types and connection probabilities.

This is practically the nongrowing variant of `Graph.Establishment`. A given number of vertices are generated. Every vertex is assigned to a vertex type according to the given type probabilities. Finally, every vertex pair is evaluated and an edge is created between them with a probability depending on the types of the vertices involved.

Parameters

`n:`	the number of vertices in the graph
`type_dist:`	list giving the distribution of vertex types
`pref_matrix:`	matrix giving the connection probabilities for different vertex types.
`attribute:`	the vertex attribute name used to store the vertex types. If `None`, vertex types are not stored.
`directed:`	whether to generate a directed graph.
`loops:`	whether loop edges are allowed.

Read_DIMACS(*f*, *directed*=`False`)

Reads a graph from a file conforming to the DIMACS minimum-cost flow file format.

For the exact description of the format, see *http://lpsolve.sourceforge.net/5.5/DIMACS.htm*

Restrictions compared to the official description of the format:

- igraph's DIMACS reader requires only three fields in an arc definition, describing the edge's source and target node and its capacity.
- Source vertices are identified by 's' in the FLOW field, target vertices are identified by 't'.
- Node indices start from 1. Only a single source and target node is allowed.

Parameters

`f:`	the name of the file or a Python file handle
`directed:`	whether the generated graph should be directed.

Return Value

the generated graph, the source and the target of the flow and the edge capacities in a tuple

Read_DL(*f*, *directed*=`True`)

Reads an UCINET DL file and creates a graph based on it.

Parameters

`f`: the name of the file or a Python file handle

`directed`: whether the generated graph should be directed.

Read_Edgelist(*f*, *directed*=`True`)

Reads an edge list from a file and creates a graph based on it.

Please note that the vertex indices are zero-based. A vertex of zero degree will be created for every integer that is in range but does not appear in the edgelist.

Parameters

`f`: the name of the file or a Python file handle

`directed`: whether the generated graph should be directed.

Read_GML(*f*)

Reads a GML file and creates a graph based on it.

Parameters

`f`: the name of the file or a Python file handle

Read_GraphDB(*f*, *directed*=`False`)

Reads a GraphDB format file and creates a graph based on it.

GraphDB is a binary format, used in the graph database for isomorphism testing (see *http://amalfi.dis.unina.it/graph/*).

Parameters

`f`: the name of the file or a Python file handle

`directed`: whether the generated graph should be directed.

Read_GraphML(*f*, *directed*=`True`, *index*=`0`)

Reads a GraphML format file and creates a graph based on it.

Parameters

`f`: the name of the file or a Python file handle

`index`: if the GraphML file contains multiple graphs, specifies the one that should be loaded. Graph indices start from zero, so if you want to load the first graph, specify 0 here.

Read_Lgl(*f*, *names*=`True`, *weights*=`"if_present"`, *directed*=`True`)

Reads an .lgl file used by LGL.

It is also useful for creating graphs from "named" (and optionally weighted) edge lists.

This format is used by the Large Graph Layout program. See the documentation of LGL[a] regarding the exact format description.

LGL originally cannot deal with graphs containing multiple or loop edges, but this condition is not checked here, as igraph is happy with these.

Parameters

`f`: the name of the file or a Python file handle

`names`: If `True`, the vertex names are added as a vertex attribute called 'name'.

`weights`: If True, the edge weights are added as an edge attribute called 'weight', even if there are no weights in the file. If False, the edge weights are never added, even if they are present. `"auto"` or `"if_present"` means that weights are added if there is at least one weighted edge in the input file, but they are not added otherwise.

`directed`: whether the graph being created should be directed

[a] http://bioinformatics.icmb.utexas.edu/lgl/

Read_Ncol(*f*, *names*=`True`, *weights*=`"if_present"`, *directed*=`True`)

Reads an .ncol file used by LGL.

It is also useful for creating graphs from "named" (and optionally weighted) edge lists.

This format is used by the Large Graph Layout program. See the documentation of LGL[a] regarding the exact format description.

LGL originally cannot deal with graphs containing multiple or loop edges, but this condition is not checked here, as igraph is happy with these.

Parameters

`f:` the name of the file or a Python file handle

`names:` If `True`, the vertex names are added as a vertex attribute called 'name'.

`weights:` If True, the edge weights are added as an edge attribute called 'weight', even if there are no weights in the file. If False, the edge weights are never added, even if they are present. `"auto"` or `"if_present"` means that weights are added if there is at least one weighted edge in the input file, but they are not added otherwise.

`directed:` whether the graph being created should be directed

[a] http://bioinformatics.icmb.utexas.edu/lgl/

Read_Pajek(*f*)

Reads a Pajek format file and creates a graph based on it.

Parameters

`f:` the name of the file or a Python file handle

Recent_Degree(*n*, *m*, *window*, *outpref*=`False`, *directed*=`False`, *power*=`1`)

Generates a graph based on a stochastic model where the probability of an edge gaining a new node is proportional to the edges gained in a given time window.

Parameters

`n:`	the number of vertices
`m:`	either the number of outgoing edges generated for each vertex or a list containing the number of outgoing edges for each vertex explicitly.
`window:`	size of the window in time steps
`outpref:`	`True` if the out-degree of a given vertex should also increase its citation probability (as well as its in-degree), but it defaults to `False`.
`directed:`	`True` if the generated graph should be directed (default: `False`).
`power:`	the power constant of the nonlinear model. It can be omitted, and in this case the usual linear model will be used.

Ring(*n*, *directed*=`False`, *mutual*=`False`, *circular*=`True`)

Generates a ring graph.

Parameters

`n:`	the number of vertices in the ring
`directed:`	whether to create a directed ring.
`mutual:`	whether to create mutual edges in a directed ring.
`circular:`	whether to create a closed ring.

SBM(*n*, *pref_matrix*, *block_sizes*, *directed*=`False`, *loops*=`False`)

Generates a graph based on a stochastic blockmodel.

A given number of vertices are generated. Every vertex is assigned to a vertex type according to the given block sizes. Vertices of the same type will be assigned consecutive vertex IDs. Finally, every vertex pair is evaluated and an edge is created between them with a probability depending on the types of the vertices involved. The probabilities are taken from the preference matrix.

Parameters

`n:`	the number of vertices in the graph
`pref_matrix:`	matrix giving the connection probabilities for different vertex types.
`block_sizes:`	list giving the number of vertices in each block; must sum up to *n*.
`directed:`	whether to generate a directed graph.
`loops:`	whether loop edges are allowed.

Star(*n*, *mode*=`"undirected"`, *center*=`0`)

Generates a star graph.

Parameters

`n:`	the number of vertices in the graph
`mode:`	Gives the type of the star graph to create. Should be either "in", "out", "mutual" or "undirected"
`center:`	Vertex ID for the central vertex in the star.

Static_Fitness(*m*, *fitness_out*, *fitness_in*=`None`, *loops*=`False`, *multiple*=`False`)

Generates a non-growing graph with edge probabilities proportional to node fitnesses.

The algorithm randomly selects vertex pairs and connects them until the given number of edges are created. Each vertex is selected with a probability proportional to its fitness; for directed graphs, a vertex is selected as a source proportional to its out-fitness and as a target proportional to its in-fitness.

Parameters

`m:` the number of edges in the graph

`fitness_out:` a numeric vector with non-negative entries, one for each vertex. These values represent the fitness scores (out-fitness scores for directed graphs). *fitness* is an alias of this keyword argument.

`fitness_in:` a numeric vector with non-negative entries, one for each vertex. These values represent the in-fitness scores for directed graphs. For undirected graphs, this argument must be `None`.

`loops:` whether loop edges are allowed.

`multiple:` whether multiple edges are allowed.

Return Value

a directed or undirected graph with the prescribed power-law degree distributions.

Static_Power_Law(*n*, *m*, *exponent_out*, *exponent_in*=`-1`, *loops*=`False`, *multiple*=`False`, *finite_size_correction*=`True`)

Generates a non-growing graph with prescribed power-law degree distributions.

Parameters

`n:`	the number of vertices in the graph
`m:`	the number of edges in the graph
`exponent_out:`	the exponent of the out-degree distribution, which must be between 2 and infinity (inclusive). When *exponent_in* is not given or negative, the graph will be undirected and this parameter specifies the degree distribution. *exponent* is an alias to this keyword argument.
`exponent_in:`	the exponent of the in-degree distribution, which must be between 2 and infinity (inclusive) It can also be negative, in which case an undirected graph will be generated.
`loops:`	whether loop edges are allowed.
`multiple:`	whether multiple edges are allowed.
`finite_size_correction:`	whether to apply a finite-size correction to the generated fitness values for exponents less than 3. See the paper of Cho et al for more details.

Return Value

a directed or undirected graph with the prescribed power-law degree distributions.

Reference:

- Goh K-I, Kahng B, Kim D: Universal behaviour of load distribution in scale-free networks. Phys Rev Lett 87(27):278701, 2001.
- Cho YS, Kim JS, Park J, Kahng B, Kim D: Percolation transitions in scale-free networks under the Achlioptas process. Phys Rev Lett 103:135702, 2009.

Tree(*n*, *children*, *type*=`TREE_UNDIRECTED`)

Generates a tree in which almost all vertices have the same number of children.

Parameters

`n:`	the number of vertices in the graph
`children:`	the number of children of a vertex in the graph
`type:`	determines whether the tree should be directed, and if this is the case, also its orientation. Must be one of `TREE_IN`, `TREE_OUT` and `TREE_UNDIRECTED`.

Watts_Strogatz(*dim*, *size*, *nei*, *p*, *loops*=`False`, *multiple*=`False`)

Parameters

`dim:`	the dimension of the lattice
`size:`	the size of the lattice along all dimensions
`nei:`	value giving the distance (number of steps) within which two vertices will be connected.
`p:`	rewiring probability
`loops:`	specifies whether loop edges are allowed
`multiple:`	specifies whether multiple edges are allowed

See Also: `Lattice()`, `rewire()`, `rewire_edges()` if more flexibility is needed

Reference: Duncan J Watts and Steven H Strogatz: *Collective dynamics of small world networks*, Nature 393, 440-442, 1998

Weighted_Adjacency(*matrix*, *mode*=ADJ_DIRECTED, *attr*="weight", *loops*=True)

Generates a graph from its adjacency matrix.

Parameters

`matrix`: the adjacency matrix

`mode`: the mode to be used. Possible values are:

- `ADJ_DIRECTED` - the graph will be directed and a matrix element gives the number of edges between two vertex.
- `ADJ_UNDIRECTED` - alias to `ADJ_MAX` for convenience.
- `ADJ_MAX` - undirected graph will be created and the number of edges between vertex i and j is *max(A(i,j), A(j,i))*
- `ADJ_MIN` - like `ADJ_MAX`, but with *min(A(i,j), A(j,i))*
- `ADJ_PLUS` - like `ADJ_MAX`, but with *A(i,j) + A(j,i)*
- `ADJ_UPPER` - undirected graph with the upper right triangle of the matrix (including the diagonal)
- `ADJ_LOWER` - undirected graph with the lower left triangle of the matrix (including the diagonal)

These values can also be given as strings without the `ADJ` prefix.

`attr`: the name of the edge attribute that stores the edge weights.

`loops`: whether to include loop edges. When `False`, the diagonal of the adjacency matrix will be ignored.

__delitem__(*x*, *y*)

del x[y]

__getitem__(*x*, *y*)

x[y]

__init__(...)

x.__init__(...) initializes x; see help(type(x)) for signature

Overrides: object.__init__

__invert__(*x*)

~x

__new__(*T*, *S*, ...)

Return Value

a new object with type S, a subtype of T

Overrides: object.__new__

__setitem__(*x*, *i*, *y*)

x[i]=y

__str__(*x*)

str(x)

Overrides: object.__str__

add_edges(*es*)

Adds edges to the graph.

Parameters

`es`: the list of edges to be added. Every edge is represented with a tuple, containing the vertex IDs of the two endpoints. Vertices are enumerated from zero.

add_vertices(*n*)

Adds vertices to the graph.

Parameters

`n`: the number of vertices to be added

all_minimal_st_separators()

Returns a list containing all the minimal s-t separators of a graph.

A minimal separator is a set of vertices whose removal disconnects the graph, while the removal of any subset of the set keeps the graph connected.

Return Value

a list where each item lists the vertex indices of a given minimal s-t separator.

Reference: Anne Berry, Jean-Paul Bordat and Olivier Cogis: Generating all the minimal separators of a graph. In: Peter Widmayer, Gabriele Neyer and Stephan Eidenbenz (eds.): Graph-theoretic concepts in computer science, 1665, 167–172, 1999. Springer.

all_st_cuts(*source*, *target*)

Returns all the cuts between the source and target vertices in a directed graph.

This function lists all edge-cuts between a source and a target vertex. Every cut is listed exactly once.

Parameters

`source`: the source vertex ID

`target`: the target vertex ID

Return Value

a tuple where the first element is a list of lists of edge IDs representing a cut and the second element is a list of lists of vertex IDs representing the sets of vertices that were separated by the cuts.

Attention: this function has a more convenient interface in class `Graph` which wraps the result in a list of `Cut` objects. It is advised to use that.

all_st_mincuts(*source*, *target*)

Returns all minimum cuts between the source and target vertices in a directed graph.

Parameters

`source`: the source vertex ID

`target`: the target vertex ID

Attention: this function has a more convenient interface in class `Graph` which wraps the result in a list of `Cut` objects. It is advised to use that.

are_connected(*v1*, *v2*)

Decides whether two given vertices are directly connected.

Parameters

`v1`: the ID or name of the first vertex

`v2`: the ID or name of the second vertex

Return Value

`True` if there exists an edge from v1 to v2, `False` otherwise.

articulation_points()

Returns the list of articulation points in the graph.

A vertex is an articulation point if its removal increases the number of connected components in the graph.

assortativity(*types1*, *types2*=`None`, *directed*=`True`)

Returns the assortativity of the graph based on numeric properties of the vertices.

This coefficient is basically the correlation between the actual connectivity patterns of the vertices and the pattern expected from the disribution of the vertex types.

See equation (21) in Newman MEJ: Mixing patterns in networks, Phys Rev E 67:026126 (2003) for the proper definition. The actual calculation is performed using equation (26) in the same paper for directed graphs, and equation (4) in Newman MEJ: Assortative mixing in networks, Phys Rev Lett 89:208701 (2002) for undirected graphs.

Parameters

`types1`: vertex types in a list or the name of a vertex attribute holding vertex types. Types are ideally denoted by numeric values.

`types2`: in directed assortativity calculations, each vertex can have an out-type and an in-type. In this case, *types1* contains the out-types and this parameter contains the in-types in a list or the name of a vertex attribute. If `None`, it is assumed to be equal to *types1*.

`directed`: whether to consider edge directions or not.

Return Value

the assortativity coefficient

Reference:

- Newman MEJ: Mixing patterns in networks, Phys Rev E 67:026126, 2003.
- Newman MEJ: Assortative mixing in networks, Phys Rev Lett 89:208701,

  1.

See Also: `assortativity_degree()` when the types are the vertex degrees

assortativity_degree(*directed*=`True`)

Returns the assortativity of a graph based on vertex degrees.

See `assortativity()` for the details. `assortativity_degree()` simply calls `assortativity()` with the vertex degrees as types.

Parameters

`directed`: whether to consider edge directions for directed graphs or not. This argument is ignored for undirected graphs.

Return Value

the assortativity coefficient

See Also: `assortativity()`

assortativity_nominal(*types*, *directed*=`True`)

Returns the assortativity of the graph based on vertex categories.

Assuming that the vertices belong to different categories, this function calculates the assortativity coefficient, which specifies the extent to which the connections stay within categories. The assortativity coefficient is one if all the connections stay within categories and minus one if all the connections join vertices of different categories. For a randomly connected network, it is asymptotically zero.

See equation (2) in Newman MEJ: Mixing patterns in networks, Phys Rev E 67:026126 (2003) for the proper definition.

Parameters

`types`: vertex types in a list or the name of a vertex attribute holding vertex types. Types should be denoted by numeric values.

`directed`: whether to consider edge directions or not.

Return Value

the assortativity coefficient

Reference: Newman MEJ: Mixing patterns in networks, Phys Rev E 67:026126, 2003.

attributes()

Return Value

the attribute name list of the graph

authority_score(*weights*=`None`, *scale*=`True`, *arpack_options*=`None`, *return_eigenvalue*=`False`)

Calculates Kleinberg's authority score for the vertices of the graph

Parameters

`weights:`	edge weights to be used. Can be a sequence or iterable or even an edge attribute name.
`scale:`	whether to normalize the scores so that the largest one is 1.
`arpack_options:`	an `ARPACKOptions` object used to fine-tune the ARPACK eigenvector calculation. If omitted, the module-level variable called `arpack_options` is used.
`return_eigenvalue:`	whether to return the largest eigenvalue

Return Value

the authority scores in a list and optionally the largest eigenvalue as a second member of a tuple

See Also: hub_score()

average_path_length(*directed*=`True`, *unconn*=`True`)

Calculates the average path length in a graph.

Parameters

`directed:`	whether to consider directed paths in case of a directed graph. Ignored for undirected graphs.
`unconn:`	what to do when the graph is unconnected. If `True`, the average of the geodesic lengths in the components is calculated. Otherwise for all unconnected vertex pairs, a path length equal to the number of vertices is used.

Return Value

the average path length in the graph

betweenness(*vertices*=`None`, *directed*=`True`, *cutoff*=`None`, *weights*=`None`, *nobigint*=`True`)

Calculates or estimates the betweenness of vertices in a graph.

Keyword arguments:

Parameters

`vertices`: the vertices for which the betweennesses must be returned. If `None`, assumes all of the vertices in the graph.

`directed`: whether to consider directed paths.

`cutoff`: if it is an integer, only paths less than or equal to this length are considered, effectively resulting in an estimation of the betweenness for the given vertices. If `None`, the exact betweenness is returned.

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name.

`nobigint`: if `True`, igraph uses the longest available integer type on the current platform to count shortest paths. For some large networks that have a specific structure, the counters may overflow. To prevent this, use `nobigint=False`, which forces igraph to use arbitrary precision integers at the expense of increased computation time.

Return Value

the (possibly estimated) betweenness of the given vertices in a list

bfs(*vid*, *mode*=`OUT`)

Conducts a breadth first search (BFS) on the graph.

Parameters

`vid`: the root vertex ID

`mode`: either `IN` or `OUT` or `ALL`, ignored for undirected graphs.

Return Value

a tuple with the following items:

- The vertex IDs visited (in order)
- The start indices of the layers in the vertex list
- The parent of every vertex in the BFS

bfsiter(*vid*, *mode*=`OUT`, *advanced*=`False`)

Constructs a breadth first search (BFS) iterator of the graph.

Parameters

`vid`: the root vertex ID

`mode`: either `IN` or `OUT` or `ALL`.

`advanced`: if `False`, the iterator returns the next vertex in BFS order in every step. If `True`, the iterator returns the distance of the vertex from the root and the parent of the vertex in the BFS tree as well.

Return Value

the BFS iterator as an `igraph.BFSIter` object.

bibcoupling(*vertices*=`None`)

Calculates bibliographic coupling scores for given vertices in a graph.

Parameters

`vertices`: the vertices to be analysed. If `None`, all vertices will be considered.

Return Value

bibliographic coupling scores for all given vertices in a matrix.

biconnected_components(*return_articulation_points*=`True`)

Calculates the biconnected components of the graph.

Components containing a single vertex only are not considered as being biconnected.

Parameters

`return_articulation_points`: whether to return the articulation points as well

Return Value

a list of lists containing edge indices making up spanning trees of the biconnected components (one spanning tree for each component) and optionally the list of articulation points

bipartite_projection(*types*, *multiplicity*=`True`, *probe1*=`-1`, *which*=`-1`)

Internal function, undocumented.

See Also: Graph.bipartite_projection()

bipartite_projection_size(*types*)

Internal function, undocumented.

See Also: Graph.bipartite_projection_size()

bridges()

Returns the list of bridges in the graph.

An edge is a bridge if its removal increases the number of (weakly) connected components in the graph.

canonical_permutation(*sh*=`"fm"`, *color*=`None`)

Calculates the canonical permutation of a graph using the BLISS isomorphism algorithm.

Passing the permutation returned here to `Graph.permute_vertices()` will transform the graph into its canonical form.

See *http://www.tcs.hut.fi/Software/bliss/index.html* for more information about the BLISS algorithm and canonical permutations.

Parameters

`sh`: splitting heuristics for graph as a case-insensitive string, with the following possible values:

- `"f"`: first non-singleton cell
- `"fl"`: first largest non-singleton cell
- `"fs"`: first smallest non-singleton cell
- `"fm"`: first maximally non-trivially connected non-singleton cell
- `"flm"`: largest maximally non-trivially connected non-singleton cell
- `"fsm"`: smallest maximally non-trivially connected non-singleton cell

`color`: optional vector storing a coloring of the vertices with respect to which the isomorphism is computed.If `None`, all vertices have the same color.

Return Value

a permutation vector containing vertex IDs. Vertex 0 in the original graph will be mapped to an ID contained in the first element of this vector; vertex 1 will be mapped to the second and so on.

clique_number()

Returns the clique number of the graph.

The clique number of the graph is the size of the largest clique.

See Also: `largest_cliques()` for the largest cliques.

cliques(*min*=`0`, *max*=`0`)

Returns some or all cliques of the graph as a list of tuples.

A clique is a complete subgraph – a set of vertices where an edge is present between any two of them (excluding loops)

Parameters

`min`: the minimum size of cliques to be returned. If zero or negative, no lower bound will be used.

`max`: the maximum size of cliques to be returned. If zero or negative, no upper bound will be used.

closeness(*vertices*=`None`, *mode*=`ALL`, *cutoff*=`None`, *weights*=`None`, *normalized*=`True`)

Calculates the closeness centralities of given vertices in a graph.

The closeness centerality of a vertex measures how easily other vertices can be reached from it (or the other way: how easily it can be reached from the other vertices). It is defined as the number of the number of vertices minus one divided by the sum of the lengths of all geodesics from/to the given vertex.

If the graph is not connected, and there is no path between two vertices, the number of vertices is used instead the length of the geodesic. This is always longer than the longest possible geodesic.

Parameters

`vertices:` the vertices for which the closenesses must be returned. If `None`, uses all of the vertices in the graph.

`mode:` must be one of `IN`, `OUT` and `ALL`. `IN` means that the length of the incoming paths, `OUT` means that the length of the outgoing paths must be calculated. `ALL` means that both of them must be calculated.

`cutoff:` if it is an integer, only paths less than or equal to this length are considered, effectively resulting in an estimation of the closeness for the given vertices (which is always an underestimation of the real closeness, since some vertex pairs will appear as disconnected even though they are connected).. If `None`, the exact closeness is returned.

`weights:` edge weights to be used. Can be a sequence or iterable or even an edge attribute name.

`normalized:` Whether to normalize the raw closeness scores by multiplying by the number of vertices minus one.

Return Value

the calculated closenesses in a list

clusters(*mode*=`STRONG`)

Calculates the (strong or weak) clusters for a given graph.

Parameters

`mode`: must be either `STRONG` or `WEAK`, depending on the clusters being sought. Optional, defaults to `STRONG`.

Return Value

the component index for every node in the graph.

Attention: this function has a more convenient interface in class `Graph` which wraps the result in a `VertexClustering` object. It is advised to use that.

cocitation(*vertices*=`None`)

Calculates cocitation scores for given vertices in a graph.

Parameters

`vertices`: the vertices to be analysed. If `None`, all vertices will be considered.

Return Value

cocitation scores for all given vertices in a matrix.

cohesive_blocks()

Calculates the cohesive block structure of the graph.

Attention: this function has a more convenient interface in class `Graph` which wraps the result in a `CohesiveBlocks` object. It is advised to use that.

community_edge_betweenness(*directed*=`True`, *weights*=`None`)

Community structure detection based on the betweenness of the edges in the network. This algorithm was invented by M Girvan and MEJ Newman, see: M Girvan and MEJ Newman: Community structure in social and biological networks, Proc. Nat. Acad. Sci. USA 99, 7821-7826 (2002).

The idea is that the betweenness of the edges connecting two communities is typically high. So we gradually remove the edge with the highest betweenness from the network and recalculate edge betweenness after every removal, as long as all edges are removed.

Parameters

`directed`: whether to take into account the directedness of the edges when we calculate the betweenness values.

`weights`: name of an edge attribute or a list containing edge weights.

Return Value

a tuple with the merge matrix that describes the dendrogram and the modularity scores before each merge. The modularity scores use the weights if the original graph was weighted.

Attention: this function is wrapped in a more convenient syntax in the derived class `Graph`. It is advised to use that instead of this version.

community_fastgreedy(*weights*=`None`)

Finds the community structure of the graph according to the algorithm of Clauset et al based on the greedy optimization of modularity.

This is a bottom-up algorithm: initially every vertex belongs to a separate community, and communities are merged one by one. In every step, the two communities being merged are the ones which result in the maximal increase in modularity.

Parameters

`weights`: name of an edge attribute or a list containing edge weights

Return Value

a tuple with the following elements:

1. The list of merges
2. The modularity scores before each merge

Attention: this function is wrapped in a more convenient syntax in the derived class `Graph`. It is advised to use that instead of this version.

Reference: A. Clauset, M. E. J. Newman and C. Moore: *Finding community structure in very large networks.* Phys Rev E 70, 066111 (2004).

See Also: modularity()

community_infomap(*edge_weights*=`None`, *vertex_weights*=`None`, *trials*=`10`)

Finds the community structure of the network according to the Infomap method of Martin Rosvall and Carl T. Bergstrom.

See *http://www.mapequation.org* for a visualization of the algorithm or one of the references provided below.

Parameters

`edge_weights`: name of an edge attribute or a list containing edge weights.

`vertex_weights`: name of an vertex attribute or a list containing vertex weights.

`trials`: the number of attempts to partition the network.

Return Value

the calculated membership vector and the corresponding codelength in a tuple.

Reference:

- M. Rosvall and C. T. Bergstrom: *Maps of information flow reveal community structure in complex networks*. PNAS 105, 1118 (2008). *http://arxiv.org/abs/0707.0609*
- M. Rosvall, D. Axelsson and C. T. Bergstrom: *The map equation*. Eur Phys J Special Topics 178, 13 (2009). *http://arxiv.org/abs/0906.1405*

community_label_propagation(*weights*=`None`, *initial*=`None`, *fixed*=`None`)

Finds the community structure of the graph according to the label propagation method of Raghavan et al.

Initially, each vertex is assigned a different label. After that, each vertex chooses the dominant label in its neighbourhood in each iteration. Ties are broken randomly and the order in which the vertices are updated is randomized before every iteration. The algorithm ends when vertices reach a consensus.

Note that since ties are broken randomly, there is no guarantee that the algorithm returns the same community structure after each run. In fact, they frequently differ. See the paper of Raghavan et al on how to come up with an aggregated community structure.

Parameters

`weights`: name of an edge attribute or a list containing edge weights

`initial`: name of a vertex attribute or a list containing the initial vertex labels. Labels are identified by integers from zero to *n-1* where *n* is the number of vertices. Negative numbers may also be present in this vector, they represent unlabeled vertices.

`fixed`: a list of booleans for each vertex. `True` corresponds to vertices whose labeling should not change during the algorithm. It only makes sense if initial labels are also given. Unlabeled vertices cannot be fixed. Note that vertex attribute names are not accepted here.

Return Value

the resulting membership vector

Reference: Raghavan, U.N. and Albert, R. and Kumara, S. Near linear time algorithm to detect community structures in large-scale networks. Phys Rev E 76:036106, 2007. *http://arxiv.org/abs/0709.2938*.

community_leading_eigenvector(*n*=`-1`, *arpack_options*=`None`, *weights*=`None`)

A proper implementation of Newman's eigenvector community structure detection. Each split is done by maximizing the modularity regarding the original network. See the reference for details.

Parameters

`n:`	the desired number of communities. If negative, the algorithm tries to do as many splits as possible. Note that the algorithm won't split a community further if the signs of the leading eigenvector are all the same.
`arpack_options:`	an `ARPACKOptions` object used to fine-tune the ARPACK eigenvector calculation. If omitted, the module-level variable called `arpack_options` is used.
`weights:`	name of an edge attribute or a list containing edge weights

Return Value

a tuple where the first element is the membership vector of the clustering and the second element is the merge matrix.

Attention: this function is wrapped in a more convenient syntax in the derived class `Graph`. It is advised to use that instead of this version.

Reference: MEJ Newman: Finding community structure in networks using the eigenvectors of matrices, arXiv:physics/0605087

community_leiden(*edge_weights*=None, *node_weights*=None, *resolution_parameter*=1.0, *normalize_resolution*=False, *beta*=0.01, *initial_membership*=None, *n_iterations*=2)

Finds the community structure of the graph using the Leiden algorithm of Traag, van Eck & Waltman

Parameters

edge_weights:	edge weights to be used. Can be a sequence or iterable or even an edge attribute name.
node_weights:	the node weights used in the Leiden algorithm.
resolution_parameter:	the resolution parameter to use. Higher resolutions lead to more smaller communities, while lower resolutions lead to fewer larger communities.
normalize_resolution:	if set to true, the resolution parameter will be divided by the sum of the node weights. If this is not supplied, it will default to the node degree, or weighted degree in case edge_weights are supplied.
node_weights:	the node weights used in the Leiden algorithm.
beta:	parameter affecting the randomness in the Leiden algorithm. This affects only the refinement step of the algorithm.
initial_membership:	if provided, the Leiden algorithm will try to improve this provided membership. If no argument is provided, the aglorithm simply starts from the singleton partition.
n_iterations:	the number of iterations to iterate the Leiden algorithm. Each iteration may improve the partition further.

Return Value

the community membership vector.

community_multilevel(*weights*=`None`, *return_levels*=`True`)

Finds the community structure of the graph according to the multilevel algorithm of Blondel et al. This is a bottom-up algorithm: initially every vertex belongs to a separate community, and vertices are moved between communities iteratively in a way that maximizes the vertices' local contribution to the overall modularity score. When a consensus is reached (i.e. no single move would increase the modularity score), every community in the original graph is shrank to a single vertex (while keeping the total weight of the incident edges) and the process continues on the next level. The algorithm stops when it is not possible to increase the modularity any more after shrinking the communities to vertices.

Parameters

`weights`: name of an edge attribute or a list containing edge weights

`return_levels`: if `True`, returns the multilevel result. If `False`, only the best level (corresponding to the best modularity) is returned.

Return Value

either a single list describing the community membership of each vertex (if `return_levels` is `False`), or a list of community membership vectors, one corresponding to each level and a list of corresponding modularities (if `return_levels` is `True`).

Attention: this function is wrapped in a more convenient syntax in the derived class `Graph`. It is advised to use that instead of this version.

Reference: VD Blondel, J-L Guillaume, R Lambiotte and E Lefebvre: Fast unfolding of community hierarchies in large networks. J Stat Mech P10008 (2008), http://arxiv.org/abs/0803.0476

See Also: modularity()

community_optimal_modularity(*weights*=`None`)

Calculates the optimal modularity score of the graph and the corresponding community structure.

This function uses the GNU Linear Programming Kit to solve a large integer optimization problem in order to find the optimal modularity score and the corresponding community structure, therefore it is unlikely to work for graphs larger than a few (less than a hundred) vertices. Consider using one of the heuristic approaches instead if you have such a large graph.

Parameters

`weights`: name of an edge attribute or a list containing edge weights.

Return Value

the calculated membership vector and the corresponding modularity in a tuple.

community_spinglass(*weights*=None, *spins*=25, *parupdate*=False, *start_temp*=1, *stop_temp*=0.01, *cool_fact*=0.99, *update_rule*="config", *gamma*=1, *implementation*="orig", *lambda*=1)

Finds the community structure of the graph according to the spinglass community detection method of Reichardt & Bornholdt.

Parameters

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name.

`spins`: integer, the number of spins to use. This is the upper limit for the number of communities. It is not a problem to supply a (reasonably) big number here, in which case some spin states will be unpopulated.

`parupdate`: whether to update the spins of the vertices in parallel (synchronously) or not

`start_temp`: the starting temperature

`stop_temp`: the stop temperature

`cool_fact`: cooling factor for the simulated annealing

`update_rule`: specifies the null model of the simulation. Possible values are `"config"` (a random graph with the same vertex degrees as the input graph) or `"simple"` (a random graph with the same number of edges)

`gamma`: the gamma argument of the algorithm, specifying the balance between the importance of present and missing edges within a community. The default value of 1.0 assigns equal importance to both of them.

`implementation`: currently igraph contains two implementations for the spinglass community detection algorithm. The faster original implementation is the default. The other implementation is able to take into account negative weights, this can be chosen by setting `implementation` to `"neg"`.

`lambda`: the lambda argument of the algorithm, which specifies the balance between the importance of present and missing negatively weighted edges within a community. Smaller values of lambda lead to communities with less negative intra-connectivity. If the argument is zero, the algorithm reduces to a graph coloring algorithm, using the number of spins as colors. This argument is ignored if the original implementation is used.

Return Value

the community membership vector.

community_walktrap(*weights*=`None`, *steps*=`None`)

Finds the community structure of the graph according to the random walk method of Latapy & Pons.

The basic idea of the algorithm is that short random walks tend to stay in the same community. The method provides a dendrogram.

Parameters

`weights`: name of an edge attribute or a list containing edge weights

Return Value

a tuple with the list of merges and the modularity scores corresponding to each merge

Attention: this function is wrapped in a more convenient syntax in the derived class `Graph`. It is advised to use that instead of this version.

Reference: Pascal Pons, Matthieu Latapy: Computing communities in large networks using random walks, *http://arxiv.org/abs/physics/0512106*.

See Also: modularity()

complementer(*loops*=`False`)

Returns the complementer of the graph

Parameters

`loops`: whether to include loop edges in the complementer.

Return Value

the complementer of the graph

compose(*other*)

Returns the composition of two graphs.

constraint(*vertices*=`None`, *weights*=`None`)

Calculates Burt's constraint scores for given vertices in a graph.

Burt's constraint is higher if ego has less, or mutually stronger related (i.e. more redundant) contacts. Burt's measure of constraint, C[i], of vertex i's ego network V[i], is defined for directed and valued graphs as follows:

C[i] = sum(sum((p[i,q] p[q,j])^2, q in V[i], q != i,j), j in V[], j != i)

for a graph of order (ie. number od vertices) N, where proportional tie strengths are defined as follows:

p[i,j]=(a[i,j]+a[j,i]) / sum(a[i,k]+a[k,i], k in V[i], k != i), a[i,j] are elements of A and the latter being the graph adjacency matrix.

For isolated vertices, constraint is undefined.

Parameters

`vertices:` the vertices to be analysed or `None` for all vertices.

`weights:` weights associated to the edges. Can be an attribute name as well. If `None`, every edge will have the same weight.

Return Value

constraint scores for all given vertices in a matrix.

contract_vertices(*mapping*, *combine_attrs*=`None`)

Contracts some vertices in the graph, i.e. replaces groups of vertices with single vertices. Edges are not affected.

Parameters

`mapping`: numeric vector which gives the mapping between old and new vertex IDs. Vertices having the same new vertex ID in this vector will be remapped into a single new vertex. It is safe to pass the membership vector of a `VertexClustering` object here.

`combine_attrs`: specifies how to combine the attributes of the vertices being collapsed into a single one. If it is `None`, all the attributes will be lost. If it is a function, the attributes of the vertices will be collected and passed on to that function which will return the new attribute value that has to be assigned to the single collapsed vertex. It can also be one of the following string constants which define built-in collapsing functions: `sum`, `prod`, `mean`, `median`, `max`, `min`, `first`, `last`, `random`. You can also specify different combination functions for different attributes by passing a dict here which maps attribute names to functions. See `Graph.simplify()` for more details.

Return Value

`None`.

See Also: `Graph.simplify()`

convergence_degree()

Undocumented (yet).

convergence_field_size()

Undocumented (yet).

copy()

Creates a copy of the graph.

Attributes are copied by reference; in other words, if you use mutable Python objects as attribute values, these objects will still be shared between the old and new graph. You can use 'deepcopy()' from the 'copy' module if you need a truly deep copy of the graph.

coreness(*mode*=`ALL`)

Finds the coreness (shell index) of the vertices of the network.

The k-core of a graph is a maximal subgraph in which each vertex has at least degree k. (Degree here means the degree in the subgraph of course). The coreness of a vertex is k if it is a member of the k-core but not a member of the $k+1$-core.

Parameters

`mode`: whether to compute the in-corenesses (`IN`), the out-corenesses (`OUT`) or the undirected corenesses (`ALL`). Ignored and assumed to be `ALL` for undirected graphs.

Return Value

the corenesses for each vertex.

Reference: Vladimir Batagelj, Matjaz Zaversnik: *An O(m) Algorithm for Core Decomposition of Networks.*

count_isomorphisms_vf2(*other*=`None`, *color1*=`None`, *color2*=`None`, *edge_color1*=`None`, *edge_color2*=`None`, *node_compat_fn*=`None`, *edge_compat_fn*=`None`)

Determines the number of isomorphisms between the graph and another one

Vertex and edge colors may be used to restrict the isomorphisms, as only vertices and edges with the same color will be allowed to match each other.

Parameters

`other`: the other graph. If `None`, the number of automorphisms will be returned.

`color1`: optional vector storing the coloring of the vertices of the first graph. If `None`, all vertices have the same color.

`color2`: optional vector storing the coloring of the vertices of the second graph. If `None`, all vertices have the same color.

`edge_color1`: optional vector storing the coloring of the edges of the first graph. If `None`, all edges have the same color.

`edge_color2`: optional vector storing the coloring of the edges of the second graph. If `None`, all edges have the same color.

`node_compat_fn`: a function that receives the two graphs and two node indices (one from the first graph, one from the second graph) and returns `True` if the nodes given by the two indices are compatible (i.e. they could be matched to each other) or `False` otherwise. This can be used to restrict the set of isomorphisms based on node-specific criteria that are too complicated to be represented by node color vectors (i.e. the `color1` and `color2` parameters). `None` means that every node is compatible with every other node.

`edge_compat_fn`: a function that receives the two graphs and two edge indices (one from the first graph, one from the second graph) and returns `True` if the edges given by the two indices are compatible (i.e. they could be matched to each other) or `False` otherwise. This can be used to restrict the set of isomorphisms based on edge-specific criteria that are too complicated to be represented by edge color vectors (i.e. the `edge_color1` and `edge_color2` parameters). `None` means that every edge is compatible with every other node.

Return Value

the number of isomorphisms between the two given graphs (or the number of automorphisms if `other` is `None`.

count_multiple(*edges*=`None`)

Counts the multiplicities of the given edges.

Parameters

`edges`: edge indices for which we want to count their multiplicity. If `None`, all edges are counted.

Return Value

the multiplicities of the given edges as a list.

count_subisomorphisms_vf2(*other*, *color1*=None, *color2*=None, *edge_color1*=None, *edge_color2*=None, *node_compat_fn*=None, *edge_compat_fn*=None)

Determines the number of subisomorphisms between the graph and another one

Vertex and edge colors may be used to restrict the isomorphisms, as only vertices and edges with the same color will be allowed to match each other.

Parameters

`other:` the other graph.

`color1:` optional vector storing the coloring of the vertices of the first graph. If `None`, all vertices have the same color.

`color2:` optional vector storing the coloring of the vertices of the second graph. If `None`, all vertices have the same color.

`edge_color1:` optional vector storing the coloring of the edges of the first graph. If `None`, all edges have the same color.

`edge_color2:` optional vector storing the coloring of the edges of the second graph. If `None`, all edges have the same color.

`node_compat_fn:` a function that receives the two graphs and two node indices (one from the first graph, one from the second graph) and returns `True` if the nodes given by the two indices are compatible (i.e. they could be matched to each other) or `False` otherwise. This can be used to restrict the set of isomorphisms based on node-specific criteria that are too complicated to be represented by node color vectors (i.e. the `color1` and `color2` parameters). `None` means that every node is compatible with every other node.

`edge_compat_fn:` a function that receives the two graphs and two edge indices (one from the first graph, one from the second graph) and returns `True` if the edges given by the two indices are compatible (i.e. they could be matched to each other) or `False` otherwise. This can be used to restrict the set of isomorphisms based on edge-specific criteria that are too complicated to be represented by edge color vectors (i.e. the `edge_color1` and `edge_color2` parameters). `None` means that every edge is compatible with every other node.

Return Value

the number of subisomorphisms between the two given graphs

decompose(*mode*=`STRONG`, *maxcompno*=`None`, *minelements*=`1`)

Decomposes the graph into subgraphs.

Parameters

`mode`: must be either STRONG or WEAK, depending on the clusters being sought.

`maxcompno`: maximum number of components to return. `None` means all possible components.

`minelements`: minimum number of vertices in a component. By setting this to 2, isolated vertices are not returned as separate components.

Return Value

a list of the subgraphs. Every returned subgraph is a copy of the original.

degree(*vertices*, *mode*=`ALL`, *loops*=`True`)

Returns some vertex degrees from the graph.

This method accepts a single vertex ID or a list of vertex IDs as a parameter, and returns the degree of the given vertices (in the form of a single integer or a list, depending on the input parameter).

Parameters

`vertices`: a single vertex ID or a list of vertex IDs

`mode`: the type of degree to be returned (`OUT` for out-degrees, `IN` IN for in-degrees or `ALL` for the sum of them).

`loops`: whether self-loops should be counted.

delete_edges(*es*)

Removes edges from the graph.

All vertices will be kept, even if they lose all their edges. Nonexistent edges will be silently ignored.

Parameters

`es`: the list of edges to be removed. Edges are identifed by edge IDs. `EdgeSeq` objects are also accepted here. No argument deletes all edges.

delete_vertices(*vs*)

Deletes vertices and all its edges from the graph.

Parameters

`vs`: a single vertex ID or the list of vertex IDs to be deleted. No argument deletes all vertices.

density(*loops*=`False`)

Calculates the density of the graph.

Parameters

`loops`: whether to take loops into consideration. If `True`, the algorithm assumes that there might be some loops in the graph and calculates the density accordingly. If `False`, the algorithm assumes that there can't be any loops.

Return Value

the density of the graph.

dfsiter(*vid*, *mode*=`OUT`, *advanced*=`False`)

Constructs a depth first search (DFS) iterator of the graph.

Parameters

`vid`: the root vertex ID

`mode`: either `IN` or `OUT` or `ALL`.

`advanced`: if `False`, the iterator returns the next vertex in DFS order in every step. If `True`, the iterator returns the distance of the vertex from the root and the parent of the vertex in the DFS tree as well.

Return Value

the DFS iterator as an `igraph.DFSIter` object.

diameter(*directed*=`True`, *unconn*=`True`, *weights*=`None`)

Calculates the diameter of the graph.

Parameters

`directed`: whether to consider directed paths.

`unconn`: if `True` and the graph is unconnected, the longest geodesic within a component will be returned. If `False` and the graph is unconnected, the result is the number of vertices if there are no weights or infinity if there are weights.

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name.

Return Value

the diameter

difference(*other*)

Subtracts the given graph from the original

diversity(*vertices*=`None`, *weights*=`None`)

Calculates the structural diversity index of the vertices.

The structural diversity index of a vertex is simply the (normalized) Shannon entropy of the weights of the edges incident on the vertex.

The measure is defined for undirected graphs only; edge directions are ignored.

Parameters

`vertices`: the vertices for which the diversity indices must be returned. If `None`, uses all of the vertices in the graph.

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name.

Return Value

the calculated diversity indices in a list, or a single number if a single vertex was supplied.

Reference: Eagle N, Macy M and Claxton R: Network diversity and economic development, Science 328, 1029–1031, 2010.

dominator(...)

dominator(vid, mode=)

Returns the dominator tree from the given root node@param vid: the root vertex ID

Parameters

`mode:` either `IN` or `OUT`

Return Value

a list containing the dominator tree for the current graph.

dyad_census()

Dyad census, as defined by Holland and Leinhardt

Dyad census means classifying each pair of vertices of a directed graph into three categories: mutual, there is an edge from a to b and also from b to a; asymmetric, there is an edge either from a to b or from b to a but not the other way and null, no edges between a and b.

Return Value

the number of mutual, asymmetric and null connections in a 3-tuple.

Attention: this function has a more convenient interface in class `Graph` which wraps the result in a `DyadCensus` object. It is advised to use that.

eccentricity(*vertices*=`None`, *mode*=`ALL`)

Calculates the eccentricities of given vertices in a graph.

The eccentricity of a vertex is calculated by measuring the shortest distance from (or to) the vertex, to (or from) all other vertices in the graph, and taking the maximum.

Parameters

`vertices:` the vertices for which the eccentricity scores must be returned. If `None`, uses all of the vertices in the graph.

`mode:` must be one of `IN`, `OUT` and `ALL`. `IN` means that edge directions are followed; `OUT` means that edge directions are followed the opposite direction; `ALL` means that directions are ignored. The argument has no effect for undirected graphs.

Return Value

the calculated eccentricities in a list, or a single number if a single vertex was supplied.

ecount()

Counts the number of edges.

Return Value

the number of edges in the graph.

(type=integer)

edge_attributes()

Return Value

the attribute name list of the graph's edges

edge_betweenness(*directed*=`True`, *cutoff*=`None`, *weights*=`None`)

Calculates or estimates the edge betweennesses in a graph.

Parameters

`directed`: whether to consider directed paths.

`cutoff`: if it is an integer, only paths less than or equal to this length are considered, effectively resulting in an estimation of the betweenness values. If `None`, the exact betweennesses are returned.

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name.

Return Value

a list with the (exact or estimated) edge betweennesses of all edges.

edge_connectivity(*source*=-1, *target*=-1, *checks*=True)

Calculates the edge connectivity of the graph or between some vertices.

The edge connectivity between two given vertices is the number of edges that have to be removed in order to disconnect the two vertices into two separate components. This is also the number of edge disjoint directed paths between the vertices. The edge connectivity of the graph is the minimal edge connectivity over all vertex pairs.

This method calculates the edge connectivity of a given vertex pair if both the source and target vertices are given. If none of them is given (or they are both negative), the overall edge connectivity is returned.

Parameters

`source`: the source vertex involved in the calculation.

`target`: the target vertex involved in the calculation.

`checks`: if the whole graph connectivity is calculated and this is `True`, igraph performs some basic checks before calculation. If the graph is not strongly connected, then the connectivity is obviously zero. If the minimum degree is one, then the connectivity is also one. These simple checks are much faster than checking the entire graph, therefore it is advised to set this to `True`. The parameter is ignored if the connectivity between two given vertices is computed.

Return Value

the edge connectivity

eigen_adjacency(...)

eigenvector_centrality(*directed*=`True`, *scale*=`True`, *weights*=`None`, *return_eigenvalue*=`False`, *arpack_options*=`None`)

Calculates the eigenvector centralities of the vertices in a graph.

Eigenvector centrality is a measure of the importance of a node in a network. It assigns relative scores to all nodes in the network based on the principle that connections from high-scoring nodes contribute more to the score of the node in question than equal connections from low-scoring nodes. In practice, the centralities are determined by calculating eigenvector corresponding to the largest positive eigenvalue of the adjacency matrix. In the undirected case, this function considers the diagonal entries of the adjacency matrix to be twice the number of self-loops on the corresponding vertex.

In the directed case, the left eigenvector of the adjacency matrix is calculated. In other words, the centrality of a vertex is proportional to the sum of centralities of vertices pointing to it.

Eigenvector centrality is meaningful only for connected graphs. Graphs that are not connected should be decomposed into connected components, and the eigenvector centrality calculated for each separately.

Parameters

`directed:`	whether to consider edge directions in a directed graph. Ignored for undirected graphs.
`scale:`	whether to normalize the centralities so the largest one will always be 1.
`weights:`	edge weights given as a list or an edge attribute. If `None`, all edges have equal weight.
`return_eigenvalue:`	whether to return the actual largest eigenvalue along with the centralities
`arpack_options:`	an `ARPACKOptions` object that can be used to fine-tune the calculation. If it is omitted, the module-level variable called `arpack_options` is used.

Return Value

the eigenvector centralities in a list and optionally the largest eigenvalue (as a second member of a tuple)

farthest_points(*directed*=`True`, *unconn*=`True`, *weights*=`None`)

Returns two vertex IDs whose distance equals the actual diameter of the graph.

If there are many shortest paths with the length of the diameter, it returns the first one it found.

Parameters

`directed`: whether to consider directed paths.

`unconn`: if `True` and the graph is unconnected, the longest geodesic within a component will be returned. If `False` and the graph is unconnected, the result contains the number of vertices if there are no weights or infinity if there are weights.

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name.

Return Value

a triplet containing the two vertex IDs and their distance. The IDs are `None` if the graph is unconnected and `unconn` is `False`.

feedback_arc_set(*weights*=`None`, *method*=`"eades"`)

Calculates an approximately or exactly minimal feedback arc set.

A feedback arc set is a set of edges whose removal makes the graph acyclic. Since this is always possible by removing all the edges, we are in general interested in removing the smallest possible number of edges, or an edge set with as small total weight as possible. This method calculates one such edge set. Note that the task is trivial for an undirected graph as it is enough to find a spanning tree and then remove all the edges not in the spanning tree. Of course it is more complicated for directed graphs.

Parameters

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name. When given, the algorithm will strive to remove lightweight edges in order to minimize the total weight of the feedback arc set.

`method`: the algorithm to use. `"eades"` uses the greedy cycle breaking heuristic of Eades, Lin and Smyth, which is linear in the number of edges but not necessarily optimal; however, it guarantees that the number of edges to be removed is smaller than |E|/2 - |V|/6. `"ip"` uses an integer programming formulation which is guaranteed to yield an optimal result, but is too slow for large graphs.

Return Value

the IDs of the edges to be removed, in a list.

Reference: Eades P, Lin X and Smyth WF: A fast and effective heuristic for the feedback arc set problem. In: Proc Inf Process Lett 319-323, 1993.

get_adjacency(*type*=`GET_ADJACENCY_BOTH`, *eids*=`False`)

Returns the adjacency matrix of a graph.

Parameters

`type`: either `GET_ADJACENCY_LOWER` (uses the lower triangle of the matrix) or `GET_ADJACENCY_UPPER` (uses the upper triangle) or `GET_ADJACENCY_BOTH` (uses both parts). Ignored for directed graphs.

`eids`: if `True`, the result matrix will contain zeros for non-edges and the ID of the edge plus one for edges in the appropriate cell. If `False`, the result matrix will contain the number of edges for each vertex pair.

Return Value

the adjacency matrix.

get_all_shortest_paths(*v*, *to*=`None`, *weights*=`None`, *mode*=`OUT`)

Calculates all of the shortest paths from/to a given node in a graph.

Parameters

`v`: the source for the calculated paths

`to`: a vertex selector describing the destination for the calculated paths. This can be a single vertex ID, a list of vertex IDs, a single vertex name, a list of vertex names or a `VertexSeq` object. `None` means all the vertices.

`weights`: edge weights in a list or the name of an edge attribute holding edge weights. If `None`, all edges are assumed to have equal weight.

`mode`: the directionality of the paths. `IN` means to calculate incoming paths, `OUT` means to calculate outgoing paths, `ALL` means to calculate both ones.

Return Value

all of the shortest path from the given node to every other reachable node in the graph in a list. Note that in case of mode=`IN`, the vertices in a path are returned in reversed order!

get_diameter(*directed*=`True`, *unconn*=`True`, *weights*=`None`)

Returns a path with the actual diameter of the graph.

If there are many shortest paths with the length of the diameter, it returns the first one it founds.

Parameters

`directed`: whether to consider directed paths.

`unconn`: if `True` and the graph is unconnected, the longest geodesic within a component will be returned. If `False` and the graph is unconnected, the result is the number of vertices if there are no weights or infinity if there are weights.

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name.

Return Value

the vertices in the path in order.

get_edgelist()

Returns the edge list of a graph.

get_eid(*v1*, *v2*, *directed*=`True`, *error*=`True`)

Returns the edge ID of an arbitrary edge between vertices v1 and v2

Parameters

`v1`: the ID or name of the first vertex

`v2`: the ID or name of the second vertex

`directed`: whether edge directions should be considered in directed graphs. The default is `True`. Ignored for undirected graphs.

`error`: if `True`, an exception will be raised when the given edge does not exist. If `False`, -1 will be returned in that case.

Return Value

the edge ID of an arbitrary edge between vertices v1 and v2

get_eids(*pairs*=`None`, *path*=`None`, *directed*=`True`, *error*=`True`)

Returns the edge IDs of some edges between some vertices.

This method can operate in two different modes, depending on which of the keyword arguments `pairs` and `path` are given.

The method does not consider multiple edges; if there are multiple edges between a pair of vertices, only the ID of one of the edges is returned.

Parameters

`pairs`: a list of integer pairs. Each integer pair is considered as a source-target vertex pair; the corresponding edge is looked up in the graph and the edge ID is returned for each pair.

`path`: a list of vertex IDs. The list is considered as a continuous path from the first vertex to the last, passing through the intermediate vertices. The corresponding edge IDs between the first and the second, the second and the third and so on are looked up in the graph and the edge IDs are returned. If both `path` and `pairs` are given, the two lists are concatenated.

`directed`: whether edge directions should be considered in directed graphs. The default is `True`. Ignored for undirected graphs.

`error`: if `True`, an exception will be raised if a given edge does not exist. If `False`, -1 will be returned in that case.

Return Value

the edge IDs in a list

get_incidence(*types*)

Internal function, undocumented.

See Also: Graph.get_incidence()

get_isomorphisms_vf2(*other*=None, *color1*=None, *color2*=None, *edge_color1*=None, *edge_color2*=None, *node_compat_fn*=None, *edge_compat_fn*=None)

Returns all isomorphisms between the graph and another one

Vertex and edge colors may be used to restrict the isomorphisms, as only vertices and edges with the same color will be allowed to match each other.

Parameters

`other`: the other graph. If `None`, the automorphisms will be returned.

`color1`: optional vector storing the coloring of the vertices of the first graph. If `None`, all vertices have the same color.

`color2`: optional vector storing the coloring of the vertices of the second graph. If `None`, all vertices have the same color.

`edge_color1`: optional vector storing the coloring of the edges of the first graph. If `None`, all edges have the same color.

`edge_color2`: optional vector storing the coloring of the edges of the second graph. If `None`, all edges have the same color.

`node_compat_fn`: a function that receives the two graphs and two node indices (one from the first graph, one from the second graph) and returns `True` if the nodes given by the two indices are compatible (i.e. they could be matched to each other) or `False` otherwise. This can be used to restrict the set of isomorphisms based on node-specific criteria that are too complicated to be represented by node color vectors (i.e. the `color1` and `color2` parameters). `None` means that every node is compatible with every other node.

`edge_compat_fn`: a function that receives the two graphs and two edge indices (one from the first graph, one from the second graph) and returns `True` if the edges given by the two indices are compatible (i.e. they could be matched to each other) or `False` otherwise. This can be used to restrict the set of isomorphisms based on edge-specific criteria that are too complicated to be represented by edge color vectors (i.e. the `edge_color1` and `edge_color2` parameters). `None` means that every edge is compatible with every other node.

Return Value

a list of lists, each item of the list containing the mapping from vertices of the second graph to the vertices of the first one

get_shortest_paths(*v*, *to*=None, *weights*=None, *mode*=OUT, *output*="vpath")

Calculates the shortest paths from/to a given node in a graph.

Parameters

`v:` the source/destination for the calculated paths

`to:` a vertex selector describing the destination/source for the calculated paths. This can be a single vertex ID, a list of vertex IDs, a single vertex name, a list of vertex names or a `VertexSeq` object. `None` means all the vertices.

`weights:` edge weights in a list or the name of an edge attribute holding edge weights. If `None`, all edges are assumed to have equal weight.

`mode:` the directionality of the paths. `IN` means to calculate incoming paths, `OUT` means to calculate outgoing paths, `ALL` means to calculate both ones.

`output:` determines what should be returned. If this is `"vpath"`, a list of vertex IDs will be returned, one path for each target vertex. For unconnected graphs, some of the list elements may be empty. Note that in case of mode=`IN`, the vertices in a path are returned in reversed order. If `output="epath"`, edge IDs are returned instead of vertex IDs.

Return Value

see the documentation of the `output` parameter.

get_subisomorphisms_lad(*other*, *domains*=`None`, *induced*=`False`, *time_limit*=`0`)

Returns all subisomorphisms between the graph and another one using the LAD algorithm.

The optional `domains` argument may be used to restrict vertices that may match each other. You can also specify whether you are interested in induced subgraphs only or not.

Parameters

`other:` the pattern graph we are looking for in the graph.

`domains:` a list of lists, one sublist belonging to each vertex in the template graph. Sublist i contains the indices of the vertices in the original graph that may match vertex i in the template graph. `None` means that every vertex may match every other vertex.

`induced:` whether to consider induced subgraphs only.

`time_limit:` an optimal time limit in seconds. Only the integral part of this number is taken into account. If the time limit is exceeded, the method will throw an exception.

Return Value

a list of lists, each item of the list containing the mapping from vertices of the second graph to the vertices of the first one

get_subisomorphisms_vf2(*other*, *color1*=None, *color2*=None, *edge_color1*=None, *edge_color2*=None, *node_compat_fn*=None, *edge_compat_fn*=None)

Returns all subisomorphisms between the graph and another one

Vertex and edge colors may be used to restrict the isomorphisms, as only vertices and edges with the same color will be allowed to match each other.

Parameters

`other`: the other graph.

`color1`: optional vector storing the coloring of the vertices of the first graph. If `None`, all vertices have the same color.

`color2`: optional vector storing the coloring of the vertices of the second graph. If `None`, all vertices have the same color.

`edge_color1`: optional vector storing the coloring of the edges of the first graph. If `None`, all edges have the same color.

`edge_color2`: optional vector storing the coloring of the edges of the second graph. If `None`, all edges have the same color.

`node_compat_fn`: a function that receives the two graphs and two node indices (one from the first graph, one from the second graph) and returns `True` if the nodes given by the two indices are compatible (i.e. they could be matched to each other) or `False` otherwise. This can be used to restrict the set of isomorphisms based on node-specific criteria that are too complicated to be represented by node color vectors (i.e. the `color1` and `color2` parameters). `None` means that every node is compatible with every other node.

`edge_compat_fn`: a function that receives the two graphs and two edge indices (one from the first graph, one from the second graph) and returns `True` if the edges given by the two indices are compatible (i.e. they could be matched to each other) or `False` otherwise. This can be used to restrict the set of isomorphisms based on edge-specific criteria that are too complicated to be represented by edge color vectors (i.e. the `edge_color1` and `edge_color2` parameters). `None` means that every edge is compatible with every other node.

Return Value

a list of lists, each item of the list containing the mapping from vertices of the second graph to the vertices of the first one

girth(*return_shortest_circle*=`False`)

Returns the girth of the graph.

The girth of a graph is the length of the shortest circle in it.

Parameters

`return_shortest_circle`: whether to return one of the shortest circles found in the graph.

Return Value

the length of the shortest circle or (if `return_shortest_circle`) is true, the shortest circle itself as a list

gomory_hu_tree(*capacity*=`None`)

Internal function, undocumented.

See Also: Graph.gomory_hu_tree()

has_multiple()

Checks whether the graph has multiple edges.

Return Value

`True` if the graph has at least one multiple edge, `False` otherwise.

(type=boolean)

hub_score(*weights*=`None`, *scale*=`True`, *arpack_options*=`None`, *return_eigenvalue*=`False`)

Calculates Kleinberg's hub score for the vertices of the graph

Parameters

`weights:`	edge weights to be used. Can be a sequence or iterable or even an edge attribute name.
`scale:`	whether to normalize the scores so that the largest one is 1.
`arpack_options:`	an `ARPACKOptions` object used to fine-tune the ARPACK eigenvector calculation. If omitted, the module-level variable called `arpack_options` is used.
`return_eigenvalue:`	whether to return the largest eigenvalue

Return Value

the hub scores in a list and optionally the largest eigenvalue as a second member of a tuple

See Also: authority_score()

incident(*vertex*, *mode*=`OUT`)

Returns the edges a given vertex is incident on.

Parameters

`vertex:`	a vertex ID
`mode:`	whether to return only successors (`OUT`), predecessors (`IN`) or both (`ALL`). Ignored for undirected graphs.

independence_number()

Returns the independence number of the graph.

The independence number of the graph is the size of the largest independent vertex set.

See Also: `largest_independent_vertex_sets()` for the largest independent vertex sets

independent_vertex_sets(*min*=0, *max*=0)

Returns some or all independent vertex sets of the graph as a list of tuples.

Two vertices are independent if there is no edge between them. Members of an independent vertex set are mutually independent.

Parameters

`min`: the minimum size of sets to be returned. If zero or negative, no lower bound will be used.

`max`: the maximum size of sets to be returned. If zero or negative, no upper bound will be used.

induced_subgraph(*vertices*, *implementation*="auto")

Returns a subgraph spanned by the given vertices.

Parameters

`vertices`: a list containing the vertex IDs which should be included in the result.

`implementation`: the implementation to use when constructing the new subgraph. igraph includes two implementations at the moment. `"copy_and_delete"` copies the original graph and removes those vertices that are not in the given set. This is more efficient if the size of the subgraph is comparable to the original graph. The other implementation (`"create_from_scratch"`) constructs the result graph from scratch and then copies the attributes accordingly. This is a better solution if the subgraph is relatively small, compared to the original graph. `"auto"` selects between the two implementations automatically, based on the ratio of the size of the subgraph and the size of the original graph.

Return Value

the subgraph

is_bipartite(*return_types*=`False`)

Decides whether the graph is bipartite or not.

Vertices of a bipartite graph can be partitioned into two groups A and B in a way that all edges go between the two groups.

Parameters

`return_types`: if `False`, the method will simply return `True` or `False` depending on whether the graph is bipartite or not. If `True`, the actual group assignments are also returned as a list of boolean values. (Note that the group assignment is not unique, especially if the graph consists of multiple components, since the assignments of components are independent from each other).

Return Value

`True` if the graph is bipartite, `False` if not. If `return_types` is `True`, the group assignment is also returned.

is_connected(*mode*=`STRONG`)

Decides whether the graph is connected.

Parameters

`mode`: whether we should calculate strong or weak connectivity.

Return Value

`True` if the graph is connected, `False` otherwise.

is_dag()

Checks whether the graph is a DAG (directed acyclic graph).

A DAG is a directed graph with no directed cycles.

Return Value

`True` if it is a DAG, `False` otherwise.

(type=boolean)

is_directed()

Checks whether the graph is directed.

Return Value

`True` if it is directed, `False` otherwise.

(type=boolean)

is_loop(*edges*=`None`)

Checks whether a specific set of edges contain loop edges

Parameters

`edges`: edge indices which we want to check. If `None`, all edges are checked.

Return Value

a list of booleans, one for every edge given

is_minimal_separator(*vertices*)

Decides whether the given vertex set is a minimal separator.

A minimal separator is a set of vertices whose removal disconnects the graph, while the removal of any subset of the set keeps the graph connected.

Parameters

`vertices`: a single vertex ID or a list of vertex IDs

Return Value

`True` is the given vertex set is a minimal separator, `False` otherwise.

is_multiple(*edges*=`None`)

Checks whether an edge is a multiple edge.

Also works for a set of edges – in this case, every edge is checked one by one. Note that if there are multiple edges going between a pair of vertices, there is always one of them that is *not* reported as multiple (only the others). This allows one to easily detect the edges that have to be deleted in order to make the graph free of multiple edges.

Parameters

`edges`: edge indices which we want to check. If `None`, all edges are checked.

Return Value

a list of booleans, one for every edge given

is_mutual(*edges*=`None`)

Checks whether an edge has an opposite pair.

Also works for a set of edges – in this case, every edge is checked one by one. The result will be a list of booleans (or a single boolean if only an edge index is supplied), every boolean corresponding to an edge in the edge set supplied. `True` is returned for a given edge a –> b if there exists another edge b –> a in the original graph (not the given edge set!). All edges in an undirected graph are mutual. In case there are multiple edges between a and b, it is enough to have at least one edge in either direction to report all edges between them as mutual, so the multiplicity of edges do not matter.

Parameters

`edges:` edge indices which we want to check. If `None`, all edges are checked.

Return Value

a list of booleans, one for every edge given

is_separator(*vertices*)

Decides whether the removal of the given vertices disconnects the graph.

Parameters

`vertices:` a single vertex ID or a list of vertex IDs

Return Value

`True` is the given vertex set is a separator, `False` if not.

is_simple()

Checks whether the graph is simple (no loop or multiple edges).

Return Value

`True` if it is simple, `False` otherwise.

(type=boolean)

isoclass(*vertices*)

Returns the isomorphism class of the graph or its subgraph.

Isomorphy class calculations are implemented only for graphs with 3 or 4 vertices.

Parameters

`vertices`: a list of vertices if we want to calculate the isomorphism class for only a subset of vertices. `None` means to use the full graph.

Return Value

the isomorphism class of the (sub)graph

isomorphic(*other*)

Checks whether the graph is isomorphic to another graph.

The algorithm being used is selected using a simple heuristic:

- If one graph is directed and the other undirected, an exception is thrown.
- If the two graphs does not have the same number of vertices and edges, it returns with `False`
- If the graphs have three or four vertices, then an O(1) algorithm is used with precomputed data.
- Otherwise if the graphs are directed, then the VF2 isomorphism algorithm is used (see `Graph.isomorphic_vf2`).
- Otherwise the BLISS isomorphism algorithm is used, see `Graph.isomorphic_bliss`.

Return Value

`True` if the graphs are isomorphic, `False` otherwise.

isomorphic_bliss(*other*, *return_mapping_12*=`False`, *return_mapping_21*=`False`, *sh1*=`"fm"`, *sh2*=`None`)

Checks whether the graph is isomorphic to another graph, using the BLISS isomorphism algorithm.

See *http://www.tcs.hut.fi/Software/bliss/index.html* for more information about the BLISS algorithm.

Parameters

`other`: the other graph with which we want to compare the graph.

`color1`: optional vector storing the coloring of the vertices of the first graph. If `None`, all vertices have the same color.

`color2`: optional vector storing the coloring of the vertices of the second graph. If `None`, all vertices have the same color.

`return_mapping_12`: if `True`, calculates the mapping which maps the vertices of the first graph to the second.

`return_mapping_21`: if `True`, calculates the mapping which maps the vertices of the second graph to the first.

`sh1`: splitting heuristics for the first graph as a case-insensitive string, with the following possible values:

- `"f"`: first non-singleton cell
- `"fl"`: first largest non-singleton cell
- `"fs"`: first smallest non-singleton cell
- `"fm"`: first maximally non-trivially connected non-singleton cell
- `"flm"`: largest maximally non-trivially connected non-singleton cell
- `"fsm"`: smallest maximally non-trivially connected non-singleton cell

`sh2`: splitting heuristics to be used for the second graph. This must be the same as `sh1`; alternatively, it can be `None`, in which case it will automatically use the same value as `sh1`. Currently it is present for backwards compatibility only.

Return Value

if no mapping is calculated, the result is `True` if the graphs are isomorphic, `False` otherwise. If any or both mappings are calculated, the result is a 3-tuple, the first element being the above mentioned boolean, the second element being the 1 -> 2 mapping and the third element being the 2 -> 1 mapping. If the corresponding mapping was not calculated, `None` is returned in the appropriate element of the 3-tuple.

isomorphic_vf2(*other*=`None`, *color1*=`None`, *color2*=`None`, *edge_color1*=`None`, *edge_color2*=`None`, *return_mapping_12*=`False`, *return_mapping_21*=`False`, *node_compat_fn*=`None`, *edge_compat_fn*=`None`, *callback*=`None`)

Checks whether the graph is isomorphic to another graph, using the VF2 isomorphism algorithm.

Vertex and edge colors may be used to restrict the isomorphisms, as only vertices and edges with the same color will be allowed to match each other.

Parameters

`other`: the other graph with which we want to compare the graph. If `None`, the automorphjisms of the graph will be tested.

`color1`: optional vector storing the coloring of the vertices of the first graph. If `None`, all vertices have the same color.

`color2`: optional vector storing the coloring of the vertices of the second graph. If `None`, all vertices have the same color.

`edge_color1`: optional vector storing the coloring of the edges of the first graph. If `None`, all edges have the same color.

`edge_color2`: optional vector storing the coloring of the edges of the second graph. If `None`, all edges have the same color.

`return_mapping_12`: if `True`, calculates the mapping which maps the vertices of the first graph to the second.

`return_mapping_21`: if `True`, calculates the mapping which maps the vertices of the second graph to the first.

`callback`: if not `None`, the isomorphism search will not stop at the first match; it will call this callback function instead for every isomorphism found. The callback function must accept four arguments: the first graph, the second graph, a mapping from the nodes of the first graph to the second, and a mapping from the nodes of the second graph to the first. The function must return `True` if the search should continue or `False` otherwise.

`node_compat_fn`: a function that receives the two graphs and two node indices (one from the first graph, one from the second graph) and returns `True` if the nodes given by the two indices are compatible (i.e. they could be matched to each other) or `False` otherwise. This can be used to restrict the set of isomorphisms based on node-specific criteria that are too

knn(*vids*=`None`, *weights*=`None`)

Calculates the average degree of the neighbors for each vertex, and the same quantity as the function of vertex degree.

Parameters

`vids`: the vertices for which the calculation is performed. `None` means all vertices.

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name. If this is given, the vertex strength will be used instead of the vertex degree in the calculations, but the "ordinary" vertex degree will be used for the second (degree- dependent) list in the result.

Return Value

two lists in a tuple. The first list contains the average degree of neighbors for each vertex, the second contains the average degree of neighbors as a function of vertex degree. The zeroth element of this list corresponds to vertices of degree 1.

laplacian(*weights*=`None`, *normalized*=`False`)

Returns the Laplacian matrix of a graph.

The Laplacian matrix is similar to the adjacency matrix, but the edges are denoted with -1 and the diagonal contains the node degrees.

Normalized Laplacian matrices have 1 or 0 in their diagonals (0 for vertices with no edges), edges are denoted by 1 / sqrt(d_i * d_j) where d_i is the degree of node i.

Multiple edges and self-loops are silently ignored. Although it is possible to calculate the Laplacian matrix of a directed graph, it does not make much sense.

Parameters

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name. When edge weights are used, the degree of a node is considered to be the weight of its incident edges.

`normalized`: whether to return the normalized Laplacian matrix.

Return Value

the Laplacian matrix.

largest_cliques()

Returns the largest cliques of the graph as a list of tuples.

Quite intuitively a clique is considered largest if there is no clique with more vertices in the whole graph. All largest cliques are maximal (i.e. nonextendable) but not all maximal cliques are largest.

See Also: `clique_number()` for the size of the largest cliques or `maximal_cliques()` for the maximal cliques

largest_independent_vertex_sets()

Returns the largest independent vertex sets of the graph as a list of tuples.

Quite intuitively an independent vertex set is considered largest if there is no other set with more vertices in the whole graph. All largest sets are maximal (i.e. nonextendable) but not all maximal sets are largest.

See Also: `independence_number()` for the size of the largest independent vertex sets or `maximal_independent_vertex_sets()` for the maximal (nonextendable) independent vertex sets

layout_bipartite(*types*=`"type"`, *hgap*=`1`, *vgap*=`1`, *maxiter*=`100`)

Place the vertices of a bipartite graph in two layers.

The layout is created by placing the vertices in two rows, according to their types. The positions of the vertices within the rows are then optimized to minimize the number of edge crossings using the heuristic used by the Sugiyama layout algorithm.

Parameters

`types`: an igraph vector containing the vertex types, or an attribute name. Anything that evalulates to `False` corresponds to vertices of the first kind, everything else to the second kind.

`hgap`: minimum horizontal gap between vertices in the same layer.

`vgap`: vertical gap between the two layers.

`maxiter`: maximum number of iterations to take in the crossing reduction step. Increase this if you feel that you are getting too many edge crossings.

Return Value

the calculated layout.

layout_circle(*dim*=2, *order*=None)

Places the vertices of the graph uniformly on a circle or a sphere.

Parameters

`dim:` the desired number of dimensions for the layout. dim=2 means a 2D layout, dim=3 means a 3D layout.

`order:` the order in which the vertices are placed along the circle. Not supported when *dim* is not equal to 2.

Return Value

the calculated layout.

layout_davidson_harel(*seed*=`None`, *maxiter*=`10`, *fineiter*=`-1`, *cool_fact*=`0.75`, *weight_node_dist*=`1.0`, *weight_border*=`0.0`, *weight_edge_lengths*=`-1`, *weight_edge_crossings*=`-1`, *weight_node_edge_dist*=`-1`)

Places the vertices on a 2D plane according to the Davidson-Harel layout algorithm.

The algorithm uses simulated annealing and a sophisticated energy function, which is unfortunately hard to parameterize for different graphs. The original publication did not disclose any parameter values, and the ones below were determined by experimentation.

The algorithm consists of two phases: an annealing phase and a fine-tuning phase. There is no simulated annealing in the second phase.

Parameters

`seed`:	if `None`, uses a random starting layout for the algorithm. If a matrix (list of lists), uses the given matrix as the starting position.
`maxiter`:	Number of iterations to perform in the annealing phase.
`fineiter`:	Number of iterations to perform in the fine-tuning phase. Negative numbers set up a reasonable default from the base-2 logarithm of the vertex count, bounded by 10 from above.
`cool_fact`:	Cooling factor of the simulated annealing phase.
`weight_node_dist`:	Weight for the node-node distances in the energy function.
`weight_border`:	Weight for the distance from the border component of the energy function. Zero means that vertices are allowed to sit on the border of the area designated for the layout.
`weight_edge_lengths`:	Weight for the edge length component of the energy function. Negative numbers are replaced by the density of the graph divided by 10.
`weight_edge_crossings`:	Weight for the edge crossing component of the energy function. Negative numbers are replaced by one minus the square root of the density of the graph.
`weight_node_edge_dist`:	Weight for the node-edge distance component of the energy function. Negative numbers are replaced by 0.2 minus 0.2 times the density of the graph.

layout_drl(*weights*=None, *fixed*=None, *seed*=None, *options*=None, *dim*=2)

Places the vertices on a 2D plane or in the 3D space ccording to the DrL layout algorithm.

This is an algorithm suitable for quite large graphs, but it can be surprisingly slow for small ones (where the simpler force-based layouts like `layout_kamada_kawai()` or `layout_fruchterman_reingold()` are more useful.

Parameters

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name.

`seed`: if `None`, uses a random starting layout for the algorithm. If a matrix (list of lists), uses the given matrix as the starting position.

`fixed`: if a seed is given, you can specify some vertices to be kept fixed at their original position in the seed by passing an appropriate list here. The list must have exactly as many items as the number of vertices in the graph. Items of the list that evaluate to `True` denote vertices that will not be moved.

`options`: if you give a string argument here, you can select from five default preset parameterisations: `default`, `coarsen` for a coarser layout, `coarsest` for an even coarser layout, `refine` for refining an existing layout and `final` for finalizing a layout. If you supply an object that is not a string, the DrL layout parameters are retrieved from the respective keys of the object (so it should be a dict or something else that supports the mapping protocol). The following keys can be used:

- `edge_cut`: edge cutting is done in the late stages of the algorithm in order to achieve less dense layouts. Edges are cut if there is a lot of stress on them (a large value in the objective function sum). The edge cutting parameter is a value between 0 and 1 with 0 representing no edge cutting and 1 representing maximal edge cutting.
- `init_iterations`: number of iterations in the initialization phase
- `init_temperature`: start temperature during initialization
- `init_attraction`: attraction during initialization
- `init_damping_mult`: damping multiplier during initialization
- `liquid_iterations`, `liquid_temperature`, `liquid_attraction`, `liquid_damping_mult`: same parameters for the liquid phase
- `expansion_iterations`, `expansion_temperature`,

layout_fruchterman_reingold(*weights*=None, *niter*=500, *seed*=None, *start_temp*=None, *minx*=None, *maxx*=None, *miny*=None, *maxy*=None, *minz*=None, *maxz*=None, *grid*="auto")

Places the vertices on a 2D plane according to the Fruchterman-Reingold algorithm.

This is a force directed layout, see Fruchterman, T. M. J. and Reingold, E. M.: Graph Drawing by Force-directed Placement. Software – Practice and Experience, 21/11, 1129–1164, 1991

Parameters

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name.

`niter`: the number of iterations to perform. The default is 500.

`start_temp`: Real scalar, the start temperature. This is the maximum amount of movement alloved along one axis, within one step, for a vertex. Currently it is decreased linearly to zero during the iteration. The default is the square root of the number of vertices divided by 10.

`minx`: if not `None`, it must be a vector with exactly as many elements as there are vertices in the graph. Each element is a minimum constraint on the X value of the vertex in the layout.

`maxx`: similar to *minx*, but with maximum constraints

`miny`: similar to *minx*, but with the Y coordinates

`maxy`: similar to *maxx*, but with the Y coordinates

`minz`: similar to *minx*, but with the Z coordinates. Use only for 3D layouts (`dim`=3).

`maxz`: similar to *maxx*, but with the Z coordinates. Use only for 3D layouts (`dim`=3).

`seed`: if `None`, uses a random starting layout for the algorithm. If a matrix (list of lists), uses the given matrix as the starting position.

`grid`: whether to use a faster, but less accurate grid-based implementation of the algorithm. `"auto"` decides based on the number of vertices in the graph; a grid will be used if there are at least 1000 vertices. `"grid"` is equivalent to `True`, `"nogrid"` is equivalent to `False`.

Return Value

the calculated layout.

layout_graphopt(*niter*=500, *node_charge*=0.001, *node_mass*=30, *spring_length*=0, *spring_constant*=1, *max_sa_movement*=5, *seed*=None)

This is a port of the graphopt layout algorithm by Michael Schmuhl. graphopt version 0.4.1 was rewritten in C and the support for layers was removed.

graphopt uses physical analogies for defining attracting and repelling forces among the vertices and then the physical system is simulated until it reaches an equilibrium or the maximal number of iterations is reached.

See *http://www.schmuhl.org/graphopt/* for the original graphopt.

Parameters

`niter:`	the number of iterations to perform. Should be a couple of hundred in general.
`node_charge:`	the charge of the vertices, used to calculate electric repulsion.
`node_mass:`	the mass of the vertices, used for the spring forces
`spring_length:`	the length of the springs
`spring_constant:`	the spring constant
`max_sa_movement:`	the maximum amount of movement allowed in a single step along a single axis.
`seed:`	a matrix containing a seed layout from which the algorithm will be started. If `None`, a random layout will be used.

Return Value

the calculated layout.

layout_grid(*width*=0, *height*=0, *dim*=2)

Places the vertices of a graph in a 2D or 3D grid.

Parameters

`width:` the number of vertices in a single row of the layout. Zero or negative numbers mean that the width should be determined automatically.

`height:` the number of vertices in a single column of the layout. Zero or negative numbers mean that the height should be determined automatically. It must not be given if the number of dimensions is 2.

`dim:` the desired number of dimensions for the layout. dim=2 means a 2D layout, dim=3 means a 3D layout.

Return Value

the calculated layout.

layout_kamada_kawai(*maxiter*=1000, *seed*=None, *maxiter*=1000, *epsilon*=0, *kkconst*=None, *minx*=None, *maxx*=None, *miny*=None, *maxy*=None, *minz*=None, *maxz*=None, *dim*=2)

Places the vertices on a plane according to the Kamada-Kawai algorithm.

This is a force directed layout, see Kamada, T. and Kawai, S.: An Algorithm for Drawing General Undirected Graphs. Information Processing Letters, 31/1, 7–15, 1989.

Parameters

`maxiter`: the maximum number of iterations to perform.

`seed`: if `None`, uses a random starting layout for the algorithm. If a matrix (list of lists), uses the given matrix as the starting position.

`epsilon`: quit if the energy of the system changes less than epsilon. See the original paper for details.

`kkconst`: the Kamada-Kawai vertex attraction constant. `None` means the square of the number of vertices.

`minx`: if not `None`, it must be a vector with exactly as many elements as there are vertices in the graph. Each element is a minimum constraint on the X value of the vertex in the layout.

`maxx`: similar to *minx*, but with maximum constraints

`miny`: similar to *minx*, but with the Y coordinates

`maxy`: similar to *maxx*, but with the Y coordinates

`minz`: similar to *minx*, but with the Z coordinates. Use only for 3D layouts (`dim`=3).

`maxz`: similar to *maxx*, but with the Z coordinates. Use only for 3D layouts (`dim`=3).

`dim`: the desired number of dimensions for the layout. dim=2 means a 2D layout, dim=3 means a 3D layout.

Return Value

the calculated layout.

layout_lgl(*maxiter*=`150`, *maxdelta*=`-1`, *area*=`-1`, *coolexp*=`1.5`, *repulserad*=`-1`, *cellsize*=`-1`, *root*=`None`)

Places the vertices on a 2D plane according to the Large Graph Layout.

Parameters

`maxiter`: the number of iterations to perform.

`maxdelta`: the maximum distance to move a vertex in an iteration. If negative, defaults to the number of vertices.

`area`: the area of the square on which the vertices will be placed. If negative, defaults to the number of vertices squared.

`coolexp`: the cooling exponent of the simulated annealing.

`repulserad`: determines the radius at which vertex-vertex repulsion cancels out attraction of adjacent vertices. If negative, defaults to *area* times the number of vertices.

`cellsize`: the size of the grid cells. When calculating the repulsion forces, only vertices in the same or neighboring grid cells are taken into account. Defaults to the fourth root of *area*.

`root`: the root vertex, this is placed first, its neighbors in the first iteration, second neighbors in the second, etc. `None` means that a random vertex will be chosen.

Return Value

the calculated layout.

layout_mds(*dist*=None, *dim*=2, *arpack_options*=None)

Places the vertices in an Euclidean space with the given number of dimensions using multidimensional scaling.

This layout requires a distance matrix, where the intersection of row i and column j specifies the desired distance between vertex i and vertex j. The algorithm will try to place the vertices in a way that approximates the distance relations prescribed in the distance matrix. igraph uses the classical multidimensional scaling by Torgerson (see reference below).

For unconnected graphs, the method will decompose the graph into weakly connected components and then lay out the components individually using the appropriate parts of the distance matrix.

Parameters

`dist`: the distance matrix. It must be symmetric and the symmetry is not checked – results are unspecified when a non-symmetric distance matrix is used. If this parameter is `None`, the shortest path lengths will be used as distances. Directed graphs are treated as undirected when calculating the shortest path lengths to ensure symmetry.

`dim`: the number of dimensions. For 2D layouts, supply 2 here; for 3D layouts, supply 3.

`arpack_options`: an `ARPACKOptions` object used to fine-tune the ARPACK eigenvector calculation. If omitted, the module-level variable called `arpack_options` is used.

Return Value

the calculated layout.

Reference: Cox & Cox: Multidimensional Scaling (1994), Chapman and Hall, London.

layout_random(*dim*=2)

Places the vertices of the graph randomly.

Parameters

`dim`: the desired number of dimensions for the layout. dim=2 means a 2D layout, dim=3 means a 3D layout.

Return Value

the coordinate pairs in a list.

layout_reingold_tilford(*mode*="out", *root*=None, *rootlevel*=None)

Places the vertices on a 2D plane according to the Reingold-Tilford layout algorithm.

This is a tree layout. If the given graph is not a tree, a breadth-first search is executed first to obtain a possible spanning tree.

Parameters

`mode:` specifies which edges to consider when builing the tree. If it is `OUT` then only the outgoing, if it is `IN` then only the incoming edges of a parent are considered. If it is `ALL` then all edges are used (this was the behaviour in igraph 0.5 and before). This parameter also influences how the root vertices are calculated if they are not given. See the *root* parameter.

`root:` the index of the root vertex or root vertices. if this is a non-empty vector then the supplied vertex IDs are used as the roots of the trees (or a single tree if the graph is connected. If this is `None` or an empty list, the root vertices are automatically calculated based on topological sorting, performed with the opposite of the *mode* argument.

`rootlevel:` this argument is useful when drawing forests which are not trees. It specifies the level of the root vertices for every tree in the forest.

Return Value

the calculated layout.

See Also: layout_reingold_tilford_circular

Reference: EM Reingold, JS Tilford: *Tidier Drawings of Trees.* IEEE Transactions on Software Engineering 7:22, 223-228, 1981.

layout_reingold_tilford_circular(*mode*="out", *root*=None, *rootlevel*=None)

Circular Reingold-Tilford layout for trees.

This layout is similar to the Reingold-Tilford layout, but the vertices are placed in a circular way, with the root vertex in the center.

See `layout_reingold_tilford` for the explanation of the parameters.

Return Value
the calculated layout.

See Also: layout_reingold_tilford

Reference: EM Reingold, JS Tilford: *Tidier Drawings of Trees.* IEEE Transactions on Software Engineering 7:22, 223-228, 1981.

layout_star(*center*=0, *order*=None)

Calculates a star-like layout for the graph.

Parameters
`center`: the ID of the vertex to put in the center

`order`: a numeric vector giving the order of the vertices (including the center vertex!). If it is `None`, the vertices will be placed in increasing vertex ID order.

Return Value
the calculated layout.

linegraph()

Returns the line graph of the graph.

The line graph *L(G)* of an undirected graph is defined as follows: *L(G)* has one vertex for each edge in G and two vertices in *L(G)* are connected iff their corresponding edges in the original graph share an end point.

The line graph of a directed graph is slightly different: two vertices are connected by a directed edge iff the target of the first vertex's corresponding edge is the same as the source of the second vertex's corresponding edge.

maxdegree(*vertices*=`None`, *mode*=`ALL`, *loops*=`False`)

Returns the maximum degree of a vertex set in the graph.

This method accepts a single vertex ID or a list of vertex IDs as a parameter, and returns the degree of the given vertices (in the form of a single integer or a list, depending on the input parameter).

Parameters

`vertices`: a single vertex ID or a list of vertex IDs, or `None` meaning all the vertices in the graph.

`mode`: the type of degree to be returned (`OUT` for out-degrees, `IN` IN for in-degrees or `ALL` for the sum of them).

`loops`: whether self-loops should be counted.

maxflow(*source*, *target*, *capacity*=`None`)

Returns the maximum flow between the source and target vertices.

Parameters

`source`: the source vertex ID

`target`: the target vertex ID

`capacity`: the capacity of the edges. It must be a list or a valid attribute name or `None`. In the latter case, every edge will have the same capacity.

Return Value

a tuple containing the following: the value of the maximum flow between the given vertices, the flow value on all the edges, the edge IDs that are part of the corresponding minimum cut, and the vertex IDs on one side of the cut. For directed graphs, the flow value vector gives the flow value on each edge. For undirected graphs, the flow value is positive if the flow goes from the smaller vertex ID to the bigger one and negative if the flow goes from the bigger vertex ID to the smaller.

Attention: this function has a more convenient interface in class `Graph` which wraps the result in a `Flow` object. It is advised to use that.

maxflow_value(*source*, *target*, *capacity*=`None`)

Returns the value of the maximum flow between the source and target vertices.

Parameters

`source:` the source vertex ID

`target:` the target vertex ID

`capacity:` the capacity of the edges. It must be a list or a valid attribute name or `None`. In the latter case, every edge will have the same capacity.

Return Value

the value of the maximum flow between the given vertices

maximal_cliques(*min*=`0`, *max*=`0`, *file*=`None`)

Returns the maximal cliques of the graph as a list of tuples.

A maximal clique is a clique which can't be extended by adding any other vertex to it. A maximal clique is not necessarily one of the largest cliques in the graph.

Parameters

`min:` the minimum size of maximal cliques to be returned. If zero or negative, no lower bound will be used.

`max:` the maximum size of maximal cliques to be returned. If zero or negative, no upper bound will be used. If nonzero, the size of every maximal clique found will be compared to this value and a clique will be returned only if its size is smaller than this limit.

`file:` a file object or the name of the file to write the results to. When this argument is `None`, the maximal cliques will be returned as a list of lists.

Return Value

the maximal cliques of the graph as a list of lists, or `None` if the `file` argument was given.@see: `largest_cliques()` for the largest cliques.

maximal_independent_vertex_sets()

Returns the maximal independent vertex sets of the graph as a list of tuples.

A maximal independent vertex set is an independent vertex set which can't be extended by adding any other vertex to it. A maximal independent vertex set is not necessarily one of the largest independent vertex sets in the graph.

See Also: `largest_independent_vertex_sets()` for the largest independent vertex sets

Reference: S. Tsukiyama, M. Ide, H. Ariyoshi and I. Shirawaka: *A new algorithm for generating all the maximal independent sets*. SIAM J Computing, 6:505–517, 1977.

mincut(*source*=`None`, *target*=`None`, *capacity*=`None`)

Calculates the minimum cut between the source and target vertices or within the whole graph.

The minimum cut is the minimum set of edges that needs to be removed to separate the source and the target (if they are given) or to disconnect the graph (if the source and target are not given). The minimum is calculated using the weights (capacities) of the edges, so the cut with the minimum total capacity is calculated. For undirected graphs and no source and target, the method uses the Stoer-Wagner algorithm. For a given source and target, the method uses the push-relabel algorithm; see the references below.

Parameters

`source`: the source vertex ID. If `None`, target must also be {None} and the calculation will be done for the entire graph (i.e. all possible vertex pairs).

`target`: the target vertex ID. If `None`, source must also be {None} and the calculation will be done for the entire graph (i.e. all possible vertex pairs).

`capacity`: the capacity of the edges. It must be a list or a valid attribute name or `None`. In the latter case, every edge will have the same capacity.

Return Value

the value of the minimum cut, the IDs of vertices in the first and second partition, and the IDs of edges in the cut, packed in a 4-tuple

Attention: this function has a more convenient interface in class `Graph` which wraps the result in a `Cut` object. It is advised to use that.

Reference:

- M. Stoer, F. Wagner: A simple min-cut algorithm. Journal of the ACM 44(4):585-591, 1997.
- A. V. Goldberg, R. E. Tarjan: A new approach to the maximum-flow problem. Journal of the ACM 35(4):921-940, 1988.

mincut_value(*source*=`-1`, *target*=`-1`, *capacity*=`None`)

Returns the minimum cut between the source and target vertices or within the whole graph.

Parameters

`source:` the source vertex ID. If negative, the calculation is done for every vertex except the target and the minimum is returned.

`target:` the target vertex ID. If negative, the calculation is done for every vertex except the source and the minimum is returned.

`capacity:` the capacity of the edges. It must be a list or a valid attribute name or `None`. In the latter case, every edge will have the same capacity.

Return Value

the value of the minimum cut between the given vertices

minimum_size_separators()

Returns a list containing all separator vertex sets of minimum size.

A vertex set is a separator if its removal disconnects the graph. This method lists all the separators for which no smaller separator set exists in the given graph.

Return Value

a list where each item lists the vertex indices of a given separator of minimum size.

Reference: Arkady Kanevsky: Finding all minimum-size separating vertex sets in a graph. Networks 23:533–541, 1993.

modularity(*membership*, *weights*=`None`)

Calculates the modularity of the graph with respect to some vertex types.

The modularity of a graph w.r.t. some division measures how good the division is, or how separated are the different vertex types from each other. It is defined as *Q=1/(2m) * sum(Aij-ki*kj/(2m)delta(ci,cj),i,j)*. m is the number of edges, Aij is the element of the A adjacency matrix in row i and column j, ki is the degree of node i, kj is the degree of node j, and Ci and `cj` are the types of the two vertices (i and j). *delta(x,y)* is one iff x=y, 0 otherwise.

If edge weights are given, the definition of modularity is modified as follows: Aij becomes the weight of the corresponding edge, ki is the total weight of edges incident on vertex i, kj is the total weight of edges incident on vertex j and m is the total edge weight in the graph.

Parameters

`membership:` the membership vector, e.g. the vertex type index for each vertex.

`weights:` optional edge weights or `None` if all edges are weighed equally.

Return Value

the modularity score. Score larger than 0.3 usually indicates strong community structure.

Attention: method overridden in `Graph` to allow `VertexClustering` objects as a parameter. This method is not strictly necessary, since the `VertexClustering` class provides a variable called `modularity`.

Reference: MEJ Newman and M Girvan: Finding and evaluating community structure in networks. Phys Rev E 69 026113, 2004.

motifs_randesu(*size*=3, *cut_prob*=None, *callback*=None)

Counts the number of motifs in the graph

Motifs are small subgraphs of a given structure in a graph. It is argued that the motif profile (ie. the number of different motifs in the graph) is characteristic for different types of networks and network function is related to the motifs in the graph.

This function is able to find the different motifs of size three and four (ie. the number of different subgraphs with three and four vertices) in the network.

In a big network the total number of motifs can be very large, so it takes a lot of time to find all of them. In such cases, a sampling method can be used. This function is capable of doing sampling via the *cut_prob* argument. This argument gives the probability that a branch of the motif search tree will not be explored.

Parameters

`size`: the size of the motifs (3 or 4).

`cut_prob`: the cut probabilities for different levels of the search tree. This must be a list of length *size* or `None` to find all motifs.

`callback`: `None` or a callable that will be called for every motif found in the graph. The callable must accept three parameters: the graph itself, the list of vertices in the motif and the isomorphism class of the motif (see `Graph.isoclass()`). The search will stop when the callback returns an object with a non-zero truth value or raises an exception.

Return Value

the list of motifs if *callback* is `None`, or `None` otherwise

Reference: S. Wernicke and F. Rasche: FANMOD: a tool for fast network motif detection, Bioinformatics 22(9), 1152–1153, 2006.

See Also: Graph.motifs_randesu_no()

motifs_randesu_estimate(*size*=3, *cut_prob*=None, *sample*)

Counts the total number of motifs in the graph

Motifs are small subgraphs of a given structure in a graph. This function estimates the total number of motifs in a graph without assigning isomorphism classes to them by extrapolating from a random sample of vertices.

Parameters

`size:` the size of the motifs (3 or 4).

`cut_prob:` the cut probabilities for different levels of the search tree. This must be a list of length *size* or `None` to find all motifs.

`sample:` the size of the sample or the vertex IDs of the vertices to be used for sampling.

Reference: S. Wernicke and F. Rasche: FANMOD: a tool for fast network motif detection, Bioinformatics 22(9), 1152–1153, 2006.

See Also: Graph.motifs_randesu()

motifs_randesu_no(*size*=3, *cut_prob*=None)

Counts the total number of motifs in the graph

Motifs are small subgraphs of a given structure in a graph. This function counts the total number of motifs in a graph without assigning isomorphism classes to them.

Parameters

`size:` the size of the motifs (3 or 4).

`cut_prob:` the cut probabilities for different levels of the search tree. This must be a list of length *size* or `None` to find all motifs.

Reference: S. Wernicke and F. Rasche: FANMOD: a tool for fast network motif detection, Bioinformatics 22(9), 1152–1153, 2006.

See Also: Graph.motifs_randesu()

neighborhood(*vertices*=`None`, *order*=`1`, *mode*=`ALL`, *mindist*=`0`)

For each vertex specified by *vertices*, returns the vertices reachable from that vertex in at most *order* steps. If *mindist* is larger than zero, vertices that are reachable in less than *mindist* steps are excluded.

Parameters

`vertices:` a single vertex ID or a list of vertex IDs, or `None` meaning all the vertices in the graph.

`order:` the order of the neighborhood, i.e. the maximum number of steps to take from the seed vertex.

`mode:` specifies how to take into account the direction of the edges if a directed graph is analyzed. `"out"` means that only the outgoing edges are followed, so all vertices reachable from the source vertex in at most *order* steps are counted. `"in"` means that only the incoming edges are followed (in reverse direction of course), so all vertices from which the source vertex is reachable in at most *order* steps are counted. `"all"` treats directed edges as undirected.

`mindist:` the minimum distance required to include a vertex in the result. If this is one, the seed vertex is not included. If this is two, the direct neighbors of the seed vertex are not included either, and so on.

Return Value

a single list specifying the neighborhood if *vertices* was an integer specifying a single vertex index, or a list of lists if *vertices* was a list or `None`.

neighborhood_size(*vertices*=`None`, *order*=`1`, *mode*=`ALL`, *mindist*=`0`)

For each vertex specified by *vertices*, returns the number of vertices reachable from that vertex in at most *order* steps. If *mindist* is larger than zero, vertices that are reachable in less than *mindist* steps are excluded.

Parameters

`vertices`: a single vertex ID or a list of vertex IDs, or `None` meaning all the vertices in the graph.

`order`: the order of the neighborhood, i.e. the maximum number of steps to take from the seed vertex.

`mode`: specifies how to take into account the direction of the edges if a directed graph is analyzed. `"out"` means that only the outgoing edges are followed, so all vertices reachable from the source vertex in at most *order* steps are counted. `"in"` means that only the incoming edges are followed (in reverse direction of course), so all vertices from which the source vertex is reachable in at most *order* steps are counted. `"all"` treats directed edges as undirected.

`mindist`: the minimum distance required to include a vertex in the result. If this is one, the seed vertex is not counted. If this is two, the direct neighbors of the seed vertex are not counted either, and so on.

Return Value

a single number specifying the neighborhood size if *vertices* was an integer specifying a single vertex index, or a list of sizes if *vertices* was a list or `None`.

neighbors(*vertex*, *mode*=`ALL`)

Returns adjacent vertices to a given vertex.

Parameters

`vertex`: a vertex ID

`mode`: whether to return only successors (`OUT`), predecessors (`IN`) or both (`ALL`). Ignored for undirected graphs.

path_length_hist(*directed*=`True`)

Calculates the path length histogram of the graph

Parameters

`directed`: whether to consider directed paths

Return Value

a tuple. The first item of the tuple is a list of path lengths, the ith element of the list contains the number of paths with length $i+1$. The second item contains the number of unconnected vertex pairs as a float (since it might not fit into an integer)

Attention: this function is wrapped in a more convenient syntax in the derived class `Graph`. It is advised to use that instead of this version.

permute_vertices(*permutation*)

Permutes the vertices of the graph according to the given permutation and returns the new graph.

Vertex k of the original graph will become vertex *permutation[k]* in the new graph. No validity checks are performed on the permutation vector.

Return Value

the new graph

personalized_pagerank(*vertices*=None, *directed*=True, *damping*=0.85, *reset*=None, *reset_vertices*=None, *weights*=None, *arpack_options*=None, *implementation*="prpack", *niter*=1000, *eps*=0.001)

Calculates the personalized PageRank values of a graph.

The personalized PageRank calculation is similar to the PageRank calculation, but the random walk is reset to a non-uniform distribution over the vertices in every step with probability *1-damping* instead of a uniform distribution.

Parameters

`vertices`: the indices of the vertices being queried. `None` means all of the vertices.

`directed`: whether to consider directed paths.

`damping`: the damping factor. *1-damping* is the PageRank value for vertices with no incoming links.

`reset`: the distribution over the vertices to be used when resetting the random walk. Can be a sequence, an iterable or a vertex attribute name as long as they return a list of floats whose length is equal to the number of vertices. If `None`, a uniform distribution is assumed, which makes the method equivalent to the original PageRank algorithm.

`reset_vertices`: an alternative way to specify the distribution over the vertices to be used when resetting the random walk. Simply supply a list of vertex IDs here, or a `VertexSeq` or a `Vertex`. Resetting will take place using a uniform distribution over the specified vertices.

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name.

`arpack_options`: an `ARPACKOptions` object used to fine-tune the ARPACK eigenvector calculation. If omitted, the module-level variable called `arpack_options` is used. This argument is ignored if not the ARPACK implementation is used, see the *implementation* argument.

`implementation`: which implementation to use to solve the PageRank eigenproblem. Possible values are:

- `"prpack"`: use the PRPACK library. This is a new implementation in igraph 0.7
- `"arpack"`: use the ARPACK library. This implementation was used from version 0.5, until version 0.7.
- `"power"`: use a simple power method. This is the implementation that was used before igraph version 0.5.

`niter`: The number of iterations to use in the power

predecessors(*vertex*)

Returns the predecessors of a given vertex.

Equivalent to calling the `Graph.neighbors` method with type=`IN`.

radius(*mode*=`OUT`)

Calculates the radius of the graph.

The radius of a graph is defined as the minimum eccentricity of its vertices (see `eccentricity()`).

Parameters

`mode`: what kind of paths to consider for the calculation in case of directed graphs. `OUT` considers paths that follow edge directions, `IN` considers paths that follow the opposite edge directions, `ALL` ignores edge directions. The argument is ignored for undirected graphs.

Return Value

the radius

See Also: `Graph.eccentricity()`

random_walk(*start*, *steps*, *mode*=`"out"`, *stuck*=`"return"`)

Performs a random walk of a given length from a given node.

Parameters

`start`: the starting vertex of the walk

`steps`: the number of steps that the random walk should take

`mode`: whether to follow outbound edges only (`OUT`), inbound edges only (`IN`) or both (`ALL`). Ignored for undirected graphs.@param stuck: what to do when the random walk gets stuck. `"return"` returns a partial random walk; `"error"` throws an exception.

Return Value

a random walk that starts from the given vertex and has at most the given length (shorter if the random walk got stuck)

reciprocity(*ignore_loops*=`True`, *mode*=`"default"`)

Reciprocity defines the proportion of mutual connections in a directed graph. It is most commonly defined as the probability that the opposite counterpart of a directed edge is also included in the graph. This measure is calculated if `mode` is `"default"`.

Prior to igraph 0.6, another measure was implemented, defined as the probability of mutual connection between a vertex pair if we know that there is a (possibly non-mutual) connection between them. In other words, (unordered) vertex pairs are classified into three groups: (1) disconnected, (2) non-reciprocally connected and (3) reciprocally connected. The result is the size of group (3), divided by the sum of sizes of groups (2) and (3). This measure is calculated if `mode` is `"ratio"`.

Parameters

`ignore_loops`: whether loop edges should be ignored.

`mode`: the algorithm to use to calculate the reciprocity; see above for more details.

Return Value

the reciprocity of the graph

rewire(*n*=`1000`, *mode*=`"simple"`)

Randomly rewires the graph while preserving the degree distribution.

Please note that the rewiring is done "in-place", so the original graph will be modified. If you want to preserve the original graph, use the `copy` method before.

Parameters

`n`: the number of rewiring trials.

`mode`: the rewiring algorithm to use. It can either be `"simple"` or `"loops"`; the former does not create or destroy loop edges while the latter does.

rewire_edges(*prob*, *loops*=`False`, *multiple*=`False`)

Rewires the edges of a graph with constant probability.

Each endpoint of each edge of the graph will be rewired with a constant probability, given in the first argument.

Please note that the rewiring is done "in-place", so the original graph will be modified. If you want to preserve the original graph, use the `copy` method before.

Parameters

`prob:` rewiring probability

`loops:` whether the algorithm is allowed to create loop edges

`multiple:` whether the algorithm is allowed to create multiple edges.

shortest_paths(*source*=`None`, *target*=`None`, *weights*=`None`, *mode*=`OUT`)

Calculates shortest path lengths for given vertices in a graph.

The algorithm used for the calculations is selected automatically: a simple BFS is used for unweighted graphs, Dijkstra's algorithm is used when all the weights are positive. Otherwise, the Bellman-Ford algorithm is used if the number of requested source vertices is larger than 100 and Johnson's algorithm is used otherwise.

Parameters

`source:` a list containing the source vertex IDs which should be included in the result. If `None`, all vertices will be considered.

`target:` a list containing the target vertex IDs which should be included in the result. If `None`, all vertices will be considered.

`weights:` a list containing the edge weights. It can also be an attribute name (edge weights are retrieved from the given attribute) or `None` (all edges have equal weight).

`mode:` the type of shortest paths to be used for the calculation in directed graphs. `OUT` means only outgoing, `IN` means only incoming paths. `ALL` means to consider the directed graph as an undirected one.

Return Value

the shortest path lengths for given vertices in a matrix

similarity_dice(*vertices*=`None`, *pairs*=`None`, *mode*=`IGRAPH_ALL`, *loops*=`True`)

Dice similarity coefficient of vertices.

The Dice similarity coefficient of two vertices is twice the number of their common neighbors divided by the sum of their degrees. This coefficient is very similar to the Jaccard coefficient, but usually gives higher similarities than its counterpart.

Parameters

`vertices`: the vertices to be analysed. If `None` and *pairs* is also `None`, all vertices will be considered.

`pairs`: the vertex pairs to be analysed. If this is given, *vertices* must be `None`, and the similarity values will be calculated only for the given pairs. Vertex pairs must be specified as tuples of vertex IDs.

`mode`: which neighbors should be considered for directed graphs. Can be `ALL`, `IN` or `OUT`, ignored for undirected graphs.

`loops`: whether vertices should be considered adjacent to themselves. Setting this to `True` assumes a loop edge for all vertices even if none is present in the graph. Setting this to `False` may result in strange results: nonadjacent vertices may have larger similarities compared to the case when an edge is added between them – however, this might be exactly the result you want to get.

Return Value

the pairwise similarity coefficients for the vertices specified, in the form of a matrix if `pairs` is `None` or in the form of a list if `pairs` is not `None`.

similarity_inverse_log_weighted(*vertices*=`None`, *mode*=`IGRAPH_ALL`)

Inverse log-weighted similarity coefficient of vertices.

Each vertex is assigned a weight which is 1 / log(degree). The log-weighted similarity of two vertices is the sum of the weights of their common neighbors.

Parameters

`vertices`: the vertices to be analysed. If `None`, all vertices will be considered.

`mode`: which neighbors should be considered for directed graphs. Can be `ALL`, `IN` or `OUT`, ignored for undirected graphs. `IN` means that the weights are determined by the out-degrees, `OUT` means that the weights are determined by the in-degrees.

Return Value

the pairwise similarity coefficients for the vertices specified, in the form of a matrix (list of lists).

similarity_jaccard(*vertices*=None, *pairs*=None, *mode*=IGRAPH_ALL, *loops*=True)

Jaccard similarity coefficient of vertices.

The Jaccard similarity coefficient of two vertices is the number of their common neighbors divided by the number of vertices that are adjacent to at least one of them.

Parameters

`vertices`: the vertices to be analysed. If `None` and *pairs* is also `None`, all vertices will be considered.

`pairs`: the vertex pairs to be analysed. If this is given, *vertices* must be `None`, and the similarity values will be calculated only for the given pairs. Vertex pairs must be specified as tuples of vertex IDs.

`mode`: which neighbors should be considered for directed graphs. Can be `ALL`, `IN` or `OUT`, ignored for undirected graphs.

`loops`: whether vertices should be considered adjacent to themselves. Setting this to `True` assumes a loop edge for all vertices even if none is present in the graph. Setting this to `False` may result in strange results: nonadjacent vertices may have larger similarities compared to the case when an edge is added between them – however, this might be exactly the result you want to get.

Return Value

the pairwise similarity coefficients for the vertices specified, in the form of a matrix if `pairs` is `None` or in the form of a list if `pairs` is not `None`.

simplify(*multiple*=`True`, *loops*=`True`, *combine_edges*=`None`)

Simplifies a graph by removing self-loops and/or multiple edges.

For example, suppose you have a graph with an edge attribute named `weight`. `graph.simplify(combine_edges=max)` will take the maximum of the weights of multiple edges and assign that weight to the collapsed edge. `graph.simplify(combine_edges=sum)` will take the sum of the weights. You can also write `graph.simplify(combine_edges=dict(weight="sum"))` or `graph.simplify(combine_edges=dict(weight=sum))`, since `sum` is recognised both as a Python built-in function and as a string constant.

Parameters

`multiple`: whether to remove multiple edges.

`loops`: whether to remove loops.

`combine_edges`: specifies how to combine the attributes of multiple edges between the same pair of vertices into a single attribute. If it is `None`, only one of the edges will be kept and all the attributes will be lost. If it is a function, the attributes of multiple edges will be collected and passed on to that function which will return the new attribute value that has to be assigned to the single collapsed edge. It can also be one of the following string constants:

- `"ignore"`: all the edge attributes will be ignored.
- `"sum"`: the sum of the edge attribute values will be used for the new edge.
- `"product"`: the product of the edge attribute values will be used for the new edge.
- `"mean"`: the mean of the edge attribute values will be used for the new edge.
- `"median"`: the median of the edge attribute values will be used for the new edge.
- `"min"`: the minimum of the edge attribute values will be used for the new edge.
- `"max"`: the maximum of the edge attribute values will be used for the new edge.
- `"first"`: the attribute value of the first edge in the collapsed set will be used for the new edge.
- `"last"`: the attribute value of the last edge in the collapsed set will be used for the new edge.
- `"random"`: a randomly selected value will be used for the new edge
- `"concat"`: the attribute values will be concatenated for the new edge.

You can also use a dict mapping edge attribute names to functions or the above string constants if

st_mincut(*source*, *target*, *capacity*=`None`)

Calculates the minimum cut between the source and target vertices in a graph.

Parameters

`source`: the source vertex ID

`target`: the target vertex ID

`capacity`: the capacity of the edges. It must be a list or a valid attribute name or `None`. In the latter case, every edge will have the same capacity.

Return Value

the value of the minimum cut, the IDs of vertices in the first and second partition, and the IDs of edges in the cut, packed in a 4-tuple

Attention: this function has a more convenient interface in class `Graph` which wraps the result in a list of `Cut` objects. It is advised to use that.

strength(*vertices*, *mode*=`ALL`, *loops*=`True`, *weights*=`None`)

Returns the strength (weighted degree) of some vertices from the graph

This method accepts a single vertex ID or a list of vertex IDs as a parameter, and returns the strength (that is, the sum of the weights of all incident edges) of the given vertices (in the form of a single integer or a list, depending on the input parameter).

Parameters

`vertices`: a single vertex ID or a list of vertex IDs

`mode`: the type of degree to be returned (`OUT` for out-degrees, `IN` IN for in-degrees or `ALL` for the sum of them).

`loops`: whether self-loops should be counted.

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name. “None“ means to treat the graph as unweighted, falling back to ordinary degree calculations.

subcomponent(*v*, *mode*=`ALL`)

Determines the indices of vertices which are in the same component as a given vertex.

Parameters

`v:` the index of the vertex used as the source/destination

`mode:` if equals to `IN`, returns the vertex IDs from where the given vertex can be reached. If equals to `OUT`, returns the vertex IDs which are reachable from the given vertex. If equals to `ALL`, returns all vertices within the same component as the given vertex, ignoring edge directions. Note that this is not equal to calculating the union of the results of `IN` and `OUT`.

Return Value

the indices of vertices which are in the same component as a given vertex.

subgraph_edges(*edges*, *delete_vertices*=`True`)

Returns a subgraph spanned by the given edges.

Parameters

`edges:` a list containing the edge IDs which should be included in the result.

`delete_vertices:` if `True`, vertices not incident on any of the specified edges will be deleted from the result. If `False`, all vertices will be kept.

Return Value

the subgraph

subisomorphic_lad(*other*, *domains*=`None`, *induced*=`False`, *time_limit*=`0`, *return_mapping*=`False`)

Checks whether a subgraph of the graph is isomorphic to another graph.

The optional `domains` argument may be used to restrict vertices that may match each other. You can also specify whether you are interested in induced subgraphs only or not.

Parameters

`other`: the pattern graph we are looking for in the graph.

`domains`: a list of lists, one sublist belonging to each vertex in the template graph. Sublist i contains the indices of the vertices in the original graph that may match vertex i in the template graph. `None` means that every vertex may match every other vertex.

`induced`: whether to consider induced subgraphs only.

`time_limit`: an optimal time limit in seconds. Only the integral part of this number is taken into account. If the time limit is exceeded, the method will throw an exception.

`return_mapping`: when `True`, the function will return a tuple, where the first element is a boolean denoting whether a subisomorphism has been found or not, and the second element describes the mapping of the vertices from the template graph to the original graph. When `False`, only the boolean is returned.

Return Value

if no mapping is calculated, the result is `True` if the graph contains a subgraph that is isomorphic to the given template, `False` otherwise. If the mapping is calculated, the result is a tuple, the first element being the above mentioned boolean, and the second element being the mapping from the target to the original graph.

subisomorphic_vf2(*other*, *color1*=`None`, *color2*=`None`, *edge_color1*=`None`, *edge_color2*=`None`, *return_mapping_12*=`False`, *return_mapping_21*=`False`, *callback*=`None`, *node_compat_fn*=`None`, *edge_compat_fn*=`None`)

Checks whether a subgraph of the graph is isomorphic to another graph.

Vertex and edge colors may be used to restrict the isomorphisms, as only vertices and edges with the same color will be allowed to match each other.

Parameters

`other`:	the other graph with which we want to compare the graph.
`color1`:	optional vector storing the coloring of the vertices of the first graph. If `None`, all vertices have the same color.
`color2`:	optional vector storing the coloring of the vertices of the second graph. If `None`, all vertices have the same color.
`edge_color1`:	optional vector storing the coloring of the edges of the first graph. If `None`, all edges have the same color.
`edge_color2`:	optional vector storing the coloring of the edges of the second graph. If `None`, all edges have the same color.
`return_mapping_12`:	if `True`, calculates the mapping which maps the vertices of the first graph to the second. The mapping can contain -1 if a given node is not mapped.
`return_mapping_21`:	if `True`, calculates the mapping which maps the vertices of the second graph to the first. The mapping can contain -1 if a given node is not mapped.
`callback`:	if not `None`, the subisomorphism search will not stop at the first match; it will call this callback function instead for every subisomorphism found. The callback function must accept four arguments: the first graph, the second graph, a mapping from the nodes of the first graph to the second, and a mapping from the nodes of the second graph to the first. The function must return `True` if the search should continue or `False` otherwise.
`node_compat_fn`:	a function that receives the two graphs and two node indices (one from the first graph, one from the second graph) and returns `True` if the nodes given by the two indices are compatible (i.e. they could be matched to each other) or `False` otherwise. This can be

successors(*vertex*)

Returns the successors of a given vertex.

Equivalent to calling the `Graph.neighbors` method with type=`OUT`.

to_directed(*mutual*=`True`)

Converts an undirected graph to directed.

Parameters

`mutual`: `True` if mutual directed edges should be created for every undirected edge. If `False`, a directed edge with arbitrary direction is created.

to_prufer()

Converts a tree graph into a Prufer sequence.

Return Value

the Prufer sequence as a list

to_undirected(*mode*=`"collapse"`, *combine_edges*=`None`)

Converts a directed graph to undirected.

Parameters

`mode`: specifies what to do with multiple directed edges going between the same vertex pair. `True` or `"collapse"` means that only a single edge should be created from multiple directed edges. `False` or `"each"` means that every edge will be kept (with the arrowheads removed). `"mutual"` creates one undirected edge for each mutual directed edge pair.

`combine_edges`: specifies how to combine the attributes of multiple edges between the same pair of vertices into a single attribute. See `Graph.simplify()` for more details.

topological_sorting(*mode*=OUT)

Calculates a possible topological sorting of the graph.

Returns a partial sorting and issues a warning if the graph is not a directed acyclic graph.

Parameters

`mode`: if `OUT`, vertices are returned according to the forward topological order – all vertices come before their successors. If `IN`, all vertices come before their ancestors.

Return Value

a possible topological ordering as a list

transitivity_avglocal_undirected(*mode*="nan")

Calculates the average of the vertex transitivities of the graph.

The transitivity measures the probability that two neighbors of a vertex are connected. In case of the average local transitivity, this probability is calculated for each vertex and then the average is taken. Vertices with less than two neighbors require special treatment, they will either be left out from the calculation or they will be considered as having zero transitivity, depending on the *mode* parameter.

Note that this measure is different from the global transitivity measure (see `transitivity_undirected()`) as it simply takes the average local transitivity across the whole network.

Parameters

`mode`: defines how to treat vertices with degree less than two. If `TRANSITIVITT_ZERO` or `"zero"`, these vertices will have zero transitivity. If `TRANSITIVITY_NAN` or `"nan"`, these vertices will be excluded from the average.

See Also: `transitivity_undirected()`, `transitivity_local_undirected()`

Reference: D. J. Watts and S. Strogatz: *Collective dynamics of small-world networks.* Nature 393(6884):440-442, 1998.

transitivity_local_undirected(*vertices*=`None`, *mode*=`"nan"`, *weights*=`None`)

Calculates the local transitivity (clustering coefficient) of the given vertices in the graph.

The transitivity measures the probability that two neighbors of a vertex are connected. In case of the local transitivity, this probability is calculated separately for each vertex.

Note that this measure is different from the global transitivity measure (see `transitivity_undirected()`) as it calculates a transitivity value for each vertex individually.

The traditional local transitivity measure applies for unweighted graphs only. When the `weights` argument is given, this function calculates the weighted local transitivity proposed by Barrat et al (see references).

Parameters

`vertices`: a list containing the vertex IDs which should be included in the result. `None` means all of the vertices.

`mode`: defines how to treat vertices with degree less than two. If `TRANSITIVITT_ZERO` or `"zero"`, these vertices will have zero transitivity. If `TRANSITIVITY_NAN` or `"nan"`, these vertices will have `NaN` (not a number) as their transitivity.

`weights`: edge weights to be used. Can be a sequence or iterable or even an edge attribute name.

Return Value

the transitivities for the given vertices in a list

See Also: `transitivity_undirected()`, `transitivity_avglocal_undirected()`

Reference:

- Watts DJ and Strogatz S: *Collective dynamics of small-world networks*. Nature 393(6884):440-442, 1998.
- Barrat A, Barthelemy M, Pastor-Satorras R and Vespignani A: *The architecture of complex weighted networks*. PNAS 101, 3747 (2004). *http://arxiv.org/abs/cond-mat/0311416*.

transitivity_undirected(*mode*="nan")

Calculates the global transitivity (clustering coefficient) of the graph.

The transitivity measures the probability that two neighbors of a vertex are connected. More precisely, this is the ratio of the triangles and connected triplets in the graph. The result is a single real number. Directed graphs are considered as undirected ones.

Note that this measure is different from the local transitivity measure (see `transitivity_local_undirected()`) as it calculates a single value for the whole graph.

Parameters

`mode`: if `TRANSITIVITY_ZERO` or `"zero"`, the result will be zero if the graph does not have any triplets. If `"nan"` or `TRANSITIVITY_NAN`, the result will be `NaN` (not a number).

Return Value

the transitivity

See Also: `transitivity_local_undirected()`, `transitivity_avglocal_undirected()`

Reference: S. Wasserman and K. Faust: *Social Network Analysis: Methods and Applications*. Cambridge: Cambridge University Press, 1994.

triad_census()

Triad census, as defined by Davis and Leinhardt

Calculating the triad census means classifying every triplets of vertices in a directed graph. A triplet can be in one of 16 states, these are listed in the documentation of the C interface of igraph.

Attention: this function has a more convenient interface in class `Graph` which wraps the result in a `TriadCensus` object. It is advised to use that. The name of the triplet classes are also documented there.

unfold_tree(*sources*=`None`, *mode*=`OUT`)

Unfolds the graph using a BFS to a tree by duplicating vertices as necessary.

Parameters

`sources`: the source vertices to start the unfolding from. It should be a list of vertex indices, preferably one vertex from each connected component. You can use `Graph.topological_sorting()` to determine a suitable set. A single vertex index is also accepted.

`mode`: which edges to follow during the BFS. `OUT` follows outgoing edges, `IN` follows incoming edges, `ALL` follows both. Ignored for undirected graphs.

Return Value

the unfolded tree graph and a mapping from the new vertex indices to the old ones.

vcount()

Counts the number of vertices.

Return Value

the number of vertices in the graph.

(type=integer)

vertex_attributes()

Return Value

the attribute name list of the graph's vertices

vertex_connectivity(*source*=-1, *target*=-1, *checks*=True, *neighbors*="error")

Calculates the vertex connectivity of the graph or between some vertices.

The vertex connectivity between two given vertices is the number of vertices that have to be removed in order to disconnect the two vertices into two separate components. This is also the number of vertex disjoint directed paths between the vertices (apart from the source and target vertices of course). The vertex connectivity of the graph is the minimal vertex connectivity over all vertex pairs.

This method calculates the vertex connectivity of a given vertex pair if both the source and target vertices are given. If none of them is given (or they are both negative), the overall vertex connectivity is returned.

Parameters

`source`: the source vertex involved in the calculation.

`target`: the target vertex involved in the calculation.

`checks`: if the whole graph connectivity is calculated and this is `True`, igraph performs some basic checks before calculation. If the graph is not strongly connected, then the connectivity is obviously zero. If the minimum degree is one, then the connectivity is also one. These simple checks are much faster than checking the entire graph, therefore it is advised to set this to `True`. The parameter is ignored if the connectivity between two given vertices is computed.

`neighbors`: tells igraph what to do when the two vertices are connected. `"error"` raises an exception, `"infinity"` returns infinity, `"ignore"` ignores the edge.

Return Value

the vertex connectivity

write_dimacs(*f*, *source*, *target*, *capacity*=`None`)

Writes the graph in DIMACS format to the given file.

Parameters

`f`: the name of the file to be written or a Python file handle

`source`: the source vertex ID

`target`: the target vertex ID

`capacity`: the capacities of the edges in a list. If it is not a list, the corresponding edge attribute will be used to retrieve capacities.

write_dot(*f*)

Writes the graph in DOT format to the given file.

DOT is the format used by the GraphViz[a] software package.

Parameters

`f`: the name of the file to be written or a Python file handle

[a] http://www.graphviz.org

write_edgelist(*f*)

Writes the edge list of a graph to a file.

Directed edges are written in (from, to) order.

Parameters

`f`: the name of the file to be written or a Python file handle

write_gml(*f*, *creator*=`None`, *ids*=`None`)

Writes the graph in GML format to the given file.

Parameters

`f`: the name of the file to be written or a Python file handle

`creator`: optional creator information to be written to the file. If `None`, the current date and time is added.

`ids`: optional numeric vertex IDs to use in the file. This must be a list of integers or `None`. If `None`, the `id` attribute of the vertices are used, or if they don't exist, numeric vertex IDs will be generated automatically.

write_graphml(*f*)

Writes the graph to a GraphML file.

Parameters

`f:` the name of the file to be written or a Python file handle

write_leda(*f*, *names*=`"name"`, *weights*=`"weights"`)

Writes the graph to a file in LEDA native format.

The LEDA format supports at most one attribute per vertex and edge. You can specify which vertex and edge attribute you want to use. Note that the name of the attribute is not saved in the LEDA file.

Parameters

`f:` the name of the file to be written or a Python file handle

`names:` the name of the vertex attribute to be stored along with the vertices. It is usually used to store the vertex names (hence the name of the keyword argument), but you may also use a numeric attribute. If you don't want to store any vertex attributes, supply `None` here.

`weights:` the name of the edge attribute to be stored along with the edges. It is usually used to store the edge weights (hence the name of the keyword argument), but you may also use a string attribute. If you don't want to store any edge attributes, supply `None` here.

write_lgl(*f*, *names*=`"name"`, *weights*=`"weights"`, *isolates*=`True`)

Writes the edge list of a graph to a file in .lgl format.

Note that multiple edges and/or loops break the LGL software, but igraph does not check for this condition. Unless you know that the graph does not have multiple edges and/or loops, it is wise to call `simplify()` before saving.

Parameters

`f:` the name of the file to be written or a Python file handle

`names:` the name of the vertex attribute containing the name of the vertices. If you don't want to store vertex names, supply `None` here.

`weights:` the name of the edge attribute containing the weight of the vertices. If you don't want to store weights, supply `None` here.

`isolates:` whether to include isolated vertices in the output.

write_ncol(*f*, *names*="name", *weights*="weights")

Writes the edge list of a graph to a file in .ncol format.

Note that multiple edges and/or loops break the LGL software, but igraph does not check for this condition. Unless you know that the graph does not have multiple edges and/or loops, it is wise to call `simplify()` before saving.

Parameters

`f`: the name of the file to be written or a Python file handle

`names`: the name of the vertex attribute containing the name of the vertices. If you don't want to store vertex names, supply `None` here.

`weights`: the name of the edge attribute containing the weight of the vertices. If you don't want to store weights, supply `None` here.

write_pajek(*f*)

Writes the graph in Pajek format to the given file.

Parameters

`f`: the name of the file to be written or a Python file handle

Inherited from object

__delattr__(), __format__(), __getattribute__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

1.12.2 Properties

Name	Description
Inherited from object	
__class__	

1.12.3 Class Variables

Name	Description
__hash__	**Value:** `None`

2 Module igraph._igraph

Low-level Python interface for the igraph library. Should not be used directly.

2.1 Functions

convex_hull(*vs*, *coords*=`False`)

Calculates the convex hull of a given point set.

Parameters

`vs`: the point set as a list of lists

`coords`: if `True`, the function returns the coordinates of the corners of the convex hull polygon, otherwise returns the corner indices.

Return Value

either the hull's corner coordinates or the point indices corresponding to them, depending on the `coords` parameter.

is_degree_sequence(*out_deg*, *in_deg*=`None`)

Returns whether a list of degrees can be a degree sequence of some graph.

Note that it is not required for the graph to be simple; in other words, this function may return `True` for degree sequences that can be realized using one or more multiple or loop edges only.

In particular, this function checks whether

- all the degrees are non-negative
- for undirected graphs, the sum of degrees are even
- for directed graphs, the two degree sequences are of the same length and equal sums

Parameters

`out_deg`: the list of degrees. For directed graphs, this list must contain the out-degrees of the vertices.

`in_deg`: the list of in-degrees for directed graphs. This parameter must be `None` for undirected graphs.

Return Value

`True` if there exists some graph that can realize the given degree sequence, `False` otherwise.@see: `is_graphical_degree_sequence()` if you do not want to allow multiple or loop edges.

is_graphical_degree_sequence(*out_deg*, *in_deg*=`None`)

Returns whether a list of degrees can be a degree sequence of some simple graph.

Note that it is required for the graph to be simple; in other words, this function will return `False` for degree sequences that cannot be realized without using one or more multiple or loop edges.

Parameters

`out_deg`: the list of degrees. For directed graphs, this list must contain the out-degrees of the vertices.

`in_deg`: the list of in-degrees for directed graphs. This parameter must be `None` for undirected graphs.

Return Value

`True` if there exists some simple graph that can realize the given degree sequence, `False` otherwise.

See Also: `is_degree_sequence()` if you want to allow multiple or loop edges.

set_progress_handler(*handler*)

Sets the handler to be called when igraph is performing a long operation.

Parameters

`handler`: the progress handler function. It must accept two arguments, the first is the message informing the user about what igraph is doing right now, the second is the actual progress information (a percentage).

set_random_number_generator(*generator*)

Sets the random number generator used by igraph.

Parameters

`generator`: the generator to be used. It must be a Python object with at least three attributes: `random`, `randint` and `gauss`. Each of them must be callable and their signature and behaviour must be identical to `random.random`, `random.randint` and `random.gauss`. By default, igraph uses the `random` module for random number generation, but you can supply your alternative implementation here. If the given generator is `None`, igraph reverts to the default Mersenne twister generator implemented in the C layer, which might be slightly faster than calling back to Python for random numbers, but you cannot set its seed or save its state.

set_status_handler(*handler*)

Sets the handler to be called when igraph tries to display a status message.

This is used to communicate the progress of some calculations where no reasonable progress percentage can be given (so it is not possible to use the progress handler).

Parameters

`handler`: the status handler function. It must accept a single argument, the message that informs the user about what igraph is doing right now.

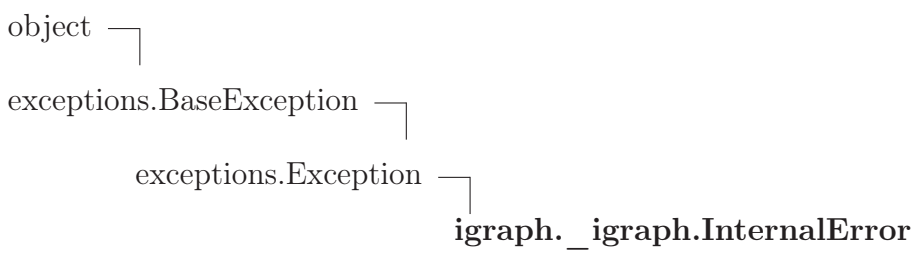
2.2 Variables

Name	Description
ADJ_DIRECTED	**Value:** 0
ADJ_LOWER	**Value:** 3
ADJ_MAX	**Value:** 1
ADJ_MIN	**Value:** 4
ADJ_PLUS	**Value:** 5
ADJ_UNDIRECTED	**Value:** 1
ADJ_UPPER	**Value:** 2
ALL	**Value:** 3
BLISS_F	**Value:** 0
BLISS_FL	**Value:** 1
BLISS_FLM	**Value:** 4
BLISS_FM	**Value:** 3
BLISS_FS	**Value:** 2
BLISS_FSM	**Value:** 5
GET_ADJACENCY_BOTH	**Value:** 2
GET_ADJACENCY_LOWER	**Value:** 1
GET_ADJACENCY_UPPER	**Value:** 0
IN	**Value:** 2
OUT	**Value:** 1
REWIRING_SIMPLE	**Value:** 0
REWIRING_SIMPLE_LOOPS	**Value:** 1
STAR_IN	**Value:** 1
STAR_MUTUAL	**Value:** 3
STAR_OUT	**Value:** 0
STAR_UNDIRECTED	**Value:** 2

continued on next page

Name	Description
STRONG	**Value:** 2
TRANSITIVITY_NAN	**Value:** 0
TRANSITIVITY_ZERO	**Value:** 1
TREE_IN	**Value:** 1
TREE_OUT	**Value:** 0
TREE_UNDIRECTED	**Value:** 2
WEAK	**Value:** 1
__build_date__	**Value:** 'Oct 8 2020'
__igraph_version__	**Value:** '0.8.3'
__package__	**Value:** None
arpack_options	**Value:** <igraph.ARPACKOptions object at 0x10c48ab90>

2.3 Class InternalError



2.3.1 Methods

Inherited from exceptions.Exception

__init__(), __new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

2.3.2 Properties

Name	Description
Inherited from exceptions.BaseException	
args, message	
Inherited from object	

continued on next page

Name	Description
__class__	

3 Package igraph.app

User interfaces of igraph

3.1 Modules

- **shell**: Command-line user interface of igraph
  (Section 4, p. 232)

3.2 Variables

Name	Description
__package__	**Value:** `None`

4 Module igraph.app.shell

Command-line user interface of igraph

The command-line interface launches a Python shell with the igraph module automatically imported into the main namespace. This is mostly a convenience module and it is used only by the `igraph` command line script which executes a suitable Python shell and automatically imports `igraph`'s classes and functions in the top-level namespace.

Supported Python shells are:

- IDLE shell (class `IDLEShell`)
- IPython shell (class `IPythonShell`)
- Classic Python shell (class `ClassicPythonShell`)

The shells are tried in the above mentioned preference order one by one, unless the `global.shells` configuration key is set which overrides the default order. IDLE shell is only tried in Windows unless explicitly stated by `global.shells`, since Linux and Mac OS X users are likely to invoke igraph from the command line.

Version: 0.8.3

4.1 Functions

main()

The main entry point for igraph when invoked from the command line shell

4.2 Variables

Name	Description
__package__	**Value:** `'igraph.app'`

4.3 Class TerminalController

A class that can be used to portably generate formatted output to a terminal.

'TerminalController' defines a set of instance variables whose values are initialized to the control sequence necessary to perform a given action. These can be simply included in normal output to the terminal:

```
>>> term = TerminalController()
>>> print 'This is '+term.GREEN+'green'+term.NORMAL
This is green
```

Alternatively, the 'render()' method can used, which replaces '${action}' with the string required to perform 'action':

```
>>> term = TerminalController()
>>> print term.render('This is ${GREEN}green${NORMAL}')
This is green
```

If the terminal doesn't support a given action, then the value of the corresponding instance variable will be set to ''. As a result, the above code will still work on terminals that do not support color, except that their output will not be colored. Also, this means that you can test whether the terminal supports a given action by simply testing the truth value of the corresponding instance variable:

```
>>> term = TerminalController()
>>> if term.CLEAR_SCREEN:
...     print 'This terminal supports clearning the screen.'
...
```

Finally, if the width and height of the terminal are known, then they will be stored in the 'COLS' and 'LINES' attributes.

Author: Edward Loper

4.3.1 Methods

__init__(*self*, *term_stream*=`sys.stdout`)

Create a 'TerminalController' and initialize its attributes with appropriate values for the current terminal. 'term_stream' is the stream that will be used for terminal output; if this stream is not a tty, then the terminal is assumed to be a dumb terminal (i.e., have no capabilities).

render(*self*, *template*)

Replace each $-substitutions in the given template string with the corresponding terminal control string (if it's defined) or '' (if it's not).

4.3.2 Class Variables

Name	Description
BOL	Move the cursor to the beginning of the line **Value:** ''
UP	Move the cursor up one line **Value:** ''
DOWN	Move the cursor down one line **Value:** ''
LEFT	Move the cursor left one char **Value:** ''
RIGHT	Move the cursor right one char **Value:** ''

continued on next page

Name	Description
CLEAR_SCREEN	Clear the screen and move to home position **Value:** ''
CLEAR_EOL	Clear to the end of the line. **Value:** ''
CLEAR_BOL	Clear to the beginning of the line. **Value:** ''
CLEAR_EOS	Clear to the end of the screen **Value:** ''
BOLD	Turn on bold mode **Value:** ''
BLINK	Turn on blink mode **Value:** ''
DIM	Turn on half-bright mode **Value:** ''
REVERSE	Turn on reverse-video mode **Value:** ''
NORMAL	Turn off all modes **Value:** ''
HIDE_CURSOR	Make the cursor invisible **Value:** ''
SHOW_CURSOR	Make the cursor visible **Value:** ''
COLS	Width of the terminal (None for unknown) **Value:** `None`
LINES	Height of the terminal (None for unknown) **Value:** `None`
WHITE	**Value:** ''
YELLOW	**Value:** ''
MAGENTA	**Value:** ''
RED	**Value:** ''
CYAN	**Value:** ''
GREEN	**Value:** ''
BLUE	**Value:** ''
BLACK	**Value:** ''
BG_CYAN	**Value:** ''
BG_GREEN	**Value:** ''
BG_BLUE	**Value:** ''
BG_BLACK	**Value:** ''
BG_WHITE	**Value:** ''
BG_YELLOW	**Value:** ''
BG_MAGENTA	**Value:** ''
BG_RED	**Value:** ''

4.4 Class ProgressBar

A 2-line progress bar, which looks like:

```
                          Header
 20% [==========----------------------------------]
```

The progress bar is colored, if the terminal supports color output; and adjusts to the width of the terminal.

4.4.1 Methods

__init__(*self*, *term*)

update(*self*, *percent*=`None`, *message*=`None`)

Updates the progress bar.

Parameters

`percent`: the percentage to be shown. If `None`, the previous value will be used.

`message`: the message to be shown above the progress bar. If `None`, the previous message will be used.

update_message(*self*, *message*)

Updates the message of the progress bar.

Parameters

`message`: the message to be shown above the progress bar

clear(*self*)

Clears the progress bar (i.e. removes it from the screen)

4.4.2 Class Variables

Name	Description
BAR	**Value:** `'%3d%% ${GREEN}[${BOLD}%s%s${NORMAL}${GREEN}]${NORMAL}'`
HEADER	**Value:** `'${BOLD}${CYAN}%s${NORMAL}\n'`

4.4.3 Instance Variables

Name	Description
cleared	true if we haven't drawn the bar yet.

4.5 Class Shell

object
└
igraph.app.shell.Shell

Known Subclasses: igraph.app.shell.ClassicPythonShell, igraph.app.shell.IDLEShell, igraph.app.shell.IP

Superclass of the embeddable shells supported by igraph

4.5.1 Methods

__init__(*self*)

x.__init__(...) initializes x; see help(type(x)) for signature

Overrides: object.__init__ extit(inherited documentation)

__call__(*self*)

supports_progress_bar(*self*)

Checks whether the shell supports progress bars.

This is done by checking for the existence of an attribute called `_progress_handler`.

supports_status_messages(*self*)

Checks whether the shell supports status messages.

This is done by checking for the existence of an attribute called `_status_handler`.

get_progress_handler(*self*)

Returns the progress handler (if exists) or None (if not).

get_status_handler(*self*)

Returns the status handler (if exists) or None (if not).

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

4.5.2 Properties

Name	Description
Inherited from object	
__class__	

4.6 Class IDLEShell

object
└
igraph.app.shell.Shell
└
igraph.app.shell.IDLEShell

IDLE embedded shell interface.

This class allows igraph to be embedded in IDLE (the Tk Python IDE).

To Do: no progress bar support yet. Shell/Restart Shell command should re-import igraph again.

4.6.1 Methods

__init__(*self*)

Constructor.

Imports IDLE's embedded shell. The implementation of this method is ripped from idlelib.PyShell.main() after removing the unnecessary parts.

Overrides: object.__init__

__call__(*self*)

Starts the shell

Overrides: igraph.app.shell.Shell.__call__

Inherited from igraph.app.shell.Shell(Section 4.5)

get_progress_handler(), get_status_handler(), supports_progress_bar(), supports_status_message

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

4.6.2 Properties

Name	Description
Inherited from object	
__class__	

4.7 Class ConsoleProgressBarMixin

Known Subclasses: igraph.app.shell.ClassicPythonShell, igraph.app.shell.IPythonShell

Mixin class for console shells that support a progress bar.

4.7.1 Methods

__init__(*self*)

x.__init__(...) initializes x; see help(type(x)) for signature

Overrides: object.__init__ extit(inherited documentation)

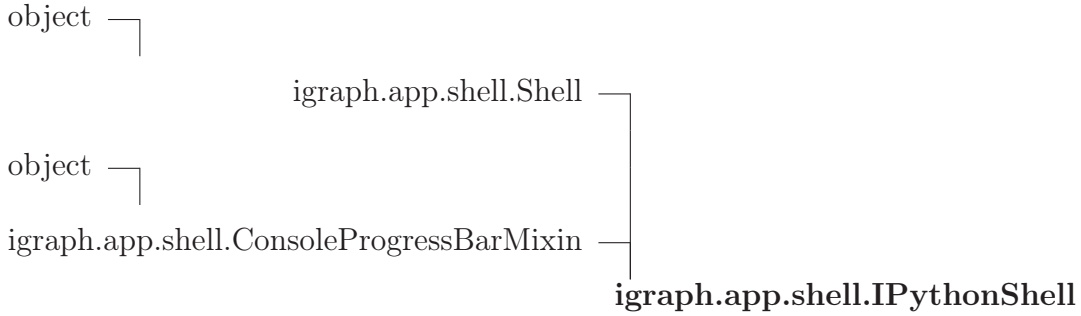
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

4.7.2 Properties

Name	Description
Inherited from object	
__class__	

4.8 Class IPythonShell



IPython embedded shell interface.

This class allows igraph to be embedded in IPython's interactive shell.

4.8.1 Methods

__init__(*self*)

Constructor.

Imports IPython's embedded shell with separator lines removed.

Overrides: object.__init__

__call__(*self*)

Starts the embedded shell.

Overrides: igraph.app.shell.Shell.__call__

Inherited from igraph.app.shell.Shell(Section 4.5)

get_progress_handler(), get_status_handler(), supports_progress_bar(), supports_status_message

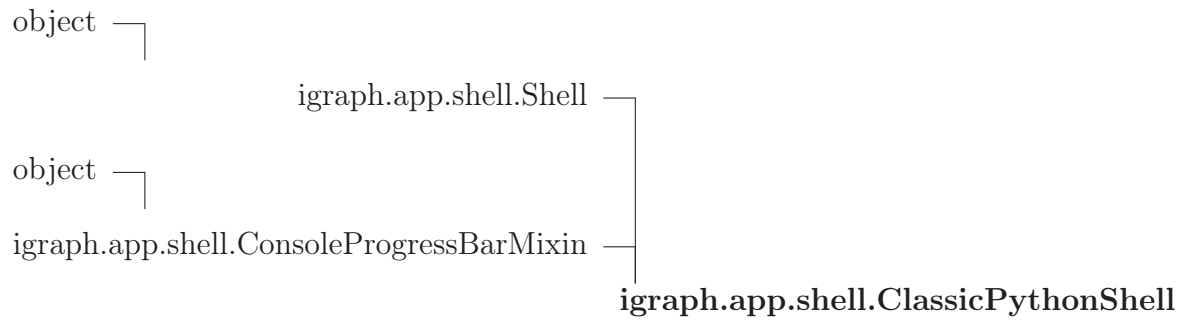
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

4.8.2 Properties

Name	Description
Inherited from object	
__class__	

4.9 Class ClassicPythonShell



Classic Python shell interface.

This class allows igraph to be embedded in Python's shell.

4.9.1 Methods

__init__(*self*)

Constructor.

Imports Python's classic shell

Overrides: object.__init__

__call__(*self*)

Starts the embedded shell.

Overrides: igraph.app.shell.Shell.__call__

Inherited from igraph.app.shell.Shell(Section 4.5)

get_progress_handler(), get_status_handler(), supports_progress_bar(), supports_status_message

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

4.9.2 Properties

Name	Description
Inherited from object	
__class__	

5 Module igraph.clustering

Classes related to graph clustering.

5.1 Functions

compare_communities(*comm1*, *comm2*, *method*='vi', *remove_none*=False)

Compares two community structures using various distance measures.

Parameters

`comm1`: the first community structure as a membership list or as a `Clustering` object.

`comm2`: the second community structure as a membership list or as a `Clustering` object.

`method`: the measure to use. `"vi"` or `"meila"` means the variation of information metric of Meila (2003), `"nmi"` or `"danon"` means the normalized mutual information as defined by Danon et al (2005), `"split-join"` means the split-join distance of van Dongen (2000), `"rand"` means the Rand index of Rand (1971), `"adjusted_rand"` means the adjusted Rand index of Hubert and Arabie (1985).

`remove_none`: whether to remove `None` entries from the membership lists. This is handy if your `Clustering` object was constructed using `VertexClustering.FromAttribute` using an attribute which was not defined for all the vertices. If `remove_none` is `False`, a `None` entry in either `comm1` or `comm2` will result in an exception. If `remove_none` is `True`, `None` values are filtered away and only the remaining lists are compared.

Return Value

the calculated measure.

Reference:

- Meila M: Comparing clusterings by the variation of information. In: Scholkopf B, Warmuth MK (eds). Learning Theory and Kernel Machines: 16th Annual Conference on Computational Learning Theory and 7th Kernel Workship, COLT/Kernel 2003, Washington, DC, USA. Lecture Notes in Computer Science, vol. 2777, Springer, 2003. ISBN: 978-3-540-40720-1.
- Danon L, Diaz-Guilera A, Duch J, Arenas A: Comparing community structure identification. J Stat Mech P09008, 2005.
- van Dongen D: Performance criteria for graph clustering and Markov cluster experiments. Technical Report INS-R0012, National Research Institute for Mathematics and Computer Science in the Netherlands, Amsterdam, May 2000.
- Rand WM: Objective criteria for the evaluation of clustering methods. J Am Stat Assoc 66(336):846-850, 1971.
- Hubert L and Arabie P: Comparing partitions. Journal of Classification 2:193-218, 1985.

split_join_distance(*comm1*, *comm2*, *remove_none*=`False`)

Calculates the split-join distance between two community structures.

The split-join distance is a distance measure defined on the space of partitions of a given set. It is the sum of the projection distance of one partition from the other and vice versa, where the projection number of A from B is if calculated as follows:

1. For each set in A, find the set in B with which it has the maximal overlap, and take note of the size of the overlap.
2. Take the sum of the maximal overlap sizes for each set in A.
3. Subtract the sum from n, the number of elements in the partition.

Note that the projection distance is asymmetric, that's why it has to be calculated in both directions and then added together. This function returns the projection distance of `comm1` from `comm2` and the projection distance of `comm2` from `comm1`, and returns them in a pair. The actual split-join distance is the sum of the two distances. The reason why it is presented this way is that one of the elements being zero then implies that one of the partitions is a subpartition of the other (and if it is close to zero, then one of the partitions is close to being a subpartition of the other).

Parameters

`comm1`: the first community structure as a membership list or as a `Clustering` object.

`comm2`: the second community structure as a membership list or as a `Clustering` object.

`remove_none`: whether to remove `None` entries from the membership lists. This is handy if your `Clustering` object was constructed using `VertexClustering.FromAttribute` using an attribute which was not defined for all the vertices. If `remove_none` is `False`, a `None` entry in either `comm1` or `comm2` will result in an exception. If `remove_none` is `True`, `None` values are filtered away and only the remaining lists are compared.

Return Value

the projection distance of `comm1` from `comm2` and vice versa in a tuple. The split-join distance is the sum of the two.

Reference: van Dongen D: Performance criteria for graph clustering and Markov cluster experiments. Technical Report INS-R0012, National Research Institute for Mathematics and Computer Science in the Netherlands, Amsterdam, May 2000.

See Also: `compare_communities()` with `method = "split-join"` if you are not interested in the individual projection distances but only the sum of them.

5.2 Variables

Name	Description
__package__	**Value:** `'igraph'`

5.3 Class Clustering

object
└
igraph.clustering.Clustering

Known Subclasses: igraph.clustering.VertexClustering

Class representing a clustering of an arbitrary ordered set.

This is now used as a base for `VertexClustering`, but it might be useful for other purposes as well.

Members of an individual cluster can be accessed by the `[]` operator:

```
>>> cl = Clustering([0,0,0,0,1,1,1,2,2,2,2])
>>> cl[0]
[0, 1, 2, 3]
```

The membership vector can be accessed by the `membership` property:

```
>>> cl.membership
[0, 0, 0, 0, 1, 1, 1, 2, 2, 2, 2]
```

The number of clusters can be retrieved by the `len` function:

```
>>> len(cl)
3
```

You can iterate over the clustering object as if it were a regular list of clusters:

```
>>> for cluster in cl:
...     print " ".join(str(idx) for idx in cluster)
...
0 1 2 3
4 5 6
7 8 9 10
```

If you need all the clusters at once as lists, you can simply convert the clustering object to a list:

```
>>> cluster_list = list(cl)
>>> print cluster_list
[[0, 1, 2, 3], [4, 5, 6], [7, 8, 9, 10]]
```

5.3.1 Methods

__init__(*self*, *membership*, *params*=`None`)

Constructor.

Parameters

`membership:` the membership list – that is, the cluster index in which each element of the set belongs to.

`params:` additional parameters to be stored in this object's dictionary.

Overrides: object.__init__

__getitem__(*self*, *idx*)

Returns the members of the specified cluster.

Parameters

`idx:` the index of the cluster

Return Value

the members of the specified cluster as a list

Raises

`IndexError` if the index is out of bounds

__iter__(*self*)

Iterates over the clusters in this clustering.

This method will return a generator that generates the clusters one by one.

__len__(*self*)

Returns the number of clusters.

Return Value

the number of clusters

__str__(*self*)

str(x)

Overrides: object.__str__ extit(inherited documentation)

as_cover(*self*)

Returns a `Cover` that contains the same clusters as this clustering.

compare_to(*self*, *other*, **args*, ***kwds*)

Compares this clustering to another one using some similarity or distance metric.

This is a convenience method that simply calls `compare_communities` with the two clusterings as arguments. Any extra positional or keyword argument is also forwarded to `compare_communities`.

size(*self*, *idx*)

Returns the size of a given cluster.

Parameters

`idx`: the cluster in which we are interested.

sizes(*self*, **args*)

Returns the size of given clusters.

The indices are given as positional arguments. If there are no positional arguments, the function will return the sizes of all clusters.

size_histogram(*self*, *bin_width*=`1`)

Returns the histogram of cluster sizes.

Parameters

`bin_width`: the bin width of the histogram

Return Value

a `Histogram` object

summary(*self*, *verbosity*=`0`, *width*=`None`)

Returns the summary of the clustering.

The summary includes the number of items and clusters, and also the list of members for each of the clusters if the verbosity is nonzero.

Parameters

`verbosity`: determines whether the cluster members should be printed. Zero verbosity prints the number of items and clusters only.

Return Value

the summary of the clustering as a string.

Inherited from object

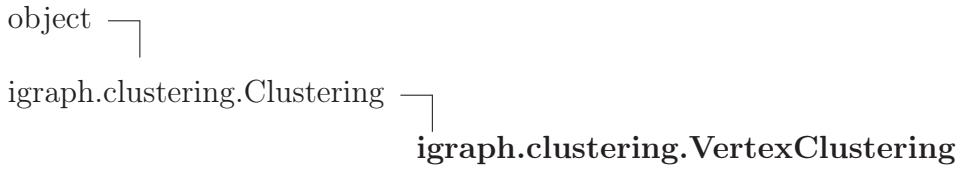
__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(),

__reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

5.3.2 Properties

Name	Description
membership	Returns the membership vector.
n	Returns the number of elements covered by this clustering.
Inherited from object __class__	

5.4 Class VertexClustering



Known Subclasses: igraph.cut.Cut

The clustering of the vertex set of a graph.

This class extends `Clustering` by linking it to a specific `Graph` object and by optionally storing the modularity score of the clustering. It also provides some handy methods like getting the subgraph corresponding to a cluster and such.

Note: since this class is linked to a `Graph`, destroying the graph by the `del` operator does not free the memory occupied by the graph if there exists a `VertexClustering` that references the `Graph`.

5.4.1 Methods

__init__(*self*, *graph*, *membership*=`None`, *modularity*=`None`, *params*=`None`, *modularity_params*=`None`)

Creates a clustering object for a given graph.

Parameters

`graph`:	the graph that will be associated to the clustering
`membership`:	the membership list. The length of the list must be equal to the number of vertices in the graph. If `None`, every vertex is assumed to belong to the same cluster.
`modularity`:	the modularity score of the clustering. If `None`, it will be calculated when needed.
`params`:	additional parameters to be stored in this object.
`modularity_params`:	arguments that should be passed to `Graph.modularity` when the modularity is (re)calculated. If the original graph was weighted, you should pass a dictionary containing a `weight` key with the appropriate value here.

Overrides: object.__init__

FromAttribute(*cls*, *graph*, *attribute*, *intervals*=`None`, *params*=`None`)

Creates a vertex clustering based on the value of a vertex attribute.

Vertices having the same attribute will correspond to the same cluster.

Parameters

`graph`: the graph on which we are working

`attribute`: name of the attribute on which the clustering is based.

`intervals`: for numeric attributes, you can either pass a single number or a list of numbers here. A single number means that the vertices will be put in bins of that width and vertices ending up in the same bin will be in the same cluster. A list of numbers specify the bin positions explicitly; e.g., `[10, 20, 30]` means that there will be four categories: vertices with the attribute value less than 10, between 10 and 20, between 20 and 30 and over 30. Intervals are closed from the left and open from the right.

`params`: additional parameters to be stored in this object.

Return Value

a new VertexClustering object

as_cover(*self*)

Returns a `VertexCover` that contains the same clusters as this clustering.

Overrides: igraph.clustering.Clustering.as_cover

cluster_graph(*self*, *combine_vertices*=`None`, *combine_edges*=`None`)

Returns a graph where each cluster is contracted into a single vertex.

In the resulting graph, vertex i represents cluster i in this clustering. Vertex i and j will be connected if there was at least one connected vertex pair *(a, b)* in the original graph such that vertex a was in cluster i and vertex b was in cluster j.

Parameters

`combine_vertices`: specifies how to derive the attributes of the vertices in the new graph from the attributes of the old ones. See `Graph.contract_vertices()` for more details.

`combine_edges`: specifies how to derive the attributes of the edges in the new graph from the attributes of the old ones. See `Graph.simplify()` for more details. If you specify `False` here, edges will not be combined, and the number of edges between the vertices representing the original clusters will be equal to the number of edges between the members of those clusters in the original graph.

Return Value

the new graph.

crossing(*self*)

Returns a boolean vector where element i is `True` iff edge i lies between clusters, `False` otherwise.

recalculate_modularity(*self*)

Recalculates the stored modularity value.

This method must be called before querying the modularity score of the clustering through the class member `modularity` or `q` if the graph has been modified (edges have been added or removed) since the creation of the `VertexClustering` object.

Return Value

the new modularity score

subgraph(*self*, *idx*)

Get the subgraph belonging to a given cluster.

Parameters
`idx`: the cluster index

Return Value
a copy of the subgraph

Precondition: the vertex set of the graph hasn't been modified since the moment the clustering was constructed.

subgraphs(*self*)

Gets all the subgraphs belonging to each of the clusters.

Return Value
a list containing copies of the subgraphs

Precondition: the vertex set of the graph hasn't been modified since the moment the clustering was constructed.

giant(*self*)

Returns the largest cluster of the clustered graph.

The largest cluster is a cluster for which no larger cluster exists in the clustering. It may also be known as the *giant community* if the clustering represents the result of a community detection function.

Return Value
a copy of the largest cluster.

Note: there can be multiple largest clusters, this method will return the copy of an arbitrary one if there are multiple largest clusters.

Precondition: the vertex set of the graph hasn't been modified since the moment the clustering was constructed.

__plot__(*self*, *context*, *bbox*, *palette*, **args*, ***kwds*)

Plots the clustering to the given Cairo context in the given bounding box.

This is done by calling `Graph.__plot__()` with the same arguments, but coloring the graph vertices according to the current clustering (unless overridden by the `vertex_color` argument explicitly).

This method understands all the positional and keyword arguments that are understood by `Graph.__plot__()`, only the differences will be highlighted here:

- `mark_groups`: whether to highlight some of the vertex groups by colored polygons. Besides the values accepted by `Graph.__plot__` (i.e., a dict mapping colors to vertex indices, a list containing lists of vertex indices, or `False`), the following are also accepted:
  - `True`: all the groups will be highlighted, the colors matching the corresponding color indices from the current palette (see the `palette` keyword argument of `Graph.__plot__`.
  - A dict mapping cluster indices or tuples of vertex indices to color names. The given clusters or vertex groups will be highlighted by the given colors.
  - A list of cluster indices. This is equivalent to passing a dict mapping numeric color indices from the current palette to cluster indices; therefore, the cluster referred to by element i of the list will be highlighted by color i from the palette.

  The value of the `plotting.mark_groups` configuration key is also taken into account here; if that configuration key is `True` and `mark_groups` is not given explicitly, it will automatically be set to `True`.

  In place of lists of vertex indices, you may also use `VertexSeq` instances.

  In place of color names, you may also use color indices into the current palette. `None` as a color name will mean that the corresponding group is ignored.
- `palette`: the palette used to resolve numeric color indices to RGBA values. By default, this is an instance of `ClusterColoringPalette`.

See Also: `Graph.__plot__()` for more supported keyword arguments.

Inherited from igraph.clustering.Clustering(Section 5.3)

__getitem__(), __iter__(), __len__(), __str__(), compare_to(), size(), size_histogram(), sizes(), summary()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(),

__reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

5.4.2 Properties

Name	Description
modularity	Returns the modularity score
q	Returns the modularity score
graph	Returns the graph belonging to this object
Inherited from igraph.clustering.Clustering (Section 5.3) membership, n	
Inherited from object __class__	

5.5 Class Dendrogram

object
└
igraph.clustering.Dendrogram

Known Subclasses: igraph.clustering.VertexDendrogram

The hierarchical clustering (dendrogram) of some dataset.

A hierarchical clustering means that we know not only the way the elements are separated into groups, but also the exact history of how individual elements were joined into larger subgroups.

This class internally represents the hierarchy by a matrix with n rows and 2 columns – or more precisely, a list of lists of size 2. This is exactly the same as the original format used by `igraph`'s C core. The ith row of the matrix contains the indices of the two clusters being joined in time step i. The joint group will be represented by the ID $n+i$, with i starting from one. The ID of the joint group will be referenced in the upcoming steps instead of any of its individual members. So, IDs less than or equal to n (where n is the number of rows in the matrix) mean the original members of the dataset (with ID from 0 to n), while IDs up from $n+1$ mean joint groups. As an example, take a look at the dendrogram and the internal representation of a given clustering of five nodes:

```
0 -+
   |
1 -+-+
     |
2 ---+-+        <====>   [[0, 1], [3, 4], [2, 5], [6, 7]]
       |
3 -+   |
   |   |
```

```
4 -+---+---
```

5.5.1 Methods

__init__(*self*, *merges*)

Creates a hierarchical clustering.

Parameters

`merges`: the merge history either in matrix or tuple format

Overrides: object.__init__

__str__(*self*)

str(x)

Overrides: object.__str__ extit(inherited documentation)

format(*self*, *format*=`'newick'`)

Formats the dendrogram in a foreign format.

Currently only the Newick format is supported.

Example:

```
>>> d = Dendrogram([(2, 3), (0, 1), (4, 5)])
>>> d.format()
'((2,3)4,(0,1)5)6;'
>>> d.names = list("ABCDEFG")
>>> d.format()
'((C,D)E,(A,B)F)G;'
```

summary(*self*, *verbosity*=0, *max_leaf_count*=40)

Returns the summary of the dendrogram.

The summary includes the number of leafs and branches, and also an ASCII art representation of the dendrogram unless it is too large.

Parameters

`verbosity`: determines whether the ASCII representation of the dendrogram should be printed. Zero verbosity prints only the number of leafs and branches.

`max_leaf_count`: the maximal number of leafs to print in the ASCII representation. If the dendrogram has more leafs than this limit, the ASCII representation will not be printed even if the verbosity is larger than or equal to 1.

Return Value

the summary of the dendrogram as a string.

__plot__(*self*, *context*, *bbox*, *palette*, **args*, ***kwds*)

Draws the dendrogram on the given Cairo context

Supported keyword arguments are:

- `orientation`: the orientation of the dendrogram. Must be one of the following values: `left-right`, `bottom-top`, `right-left` or `top-bottom`. Individual elements are always placed at the former edge and merges are performed towards the latter edge. Possible aliases: `horizontal` = `left-right`, `vertical` = `bottom-top`, `lr` = `left-right`, `rl` = `right-left`, `tb` = `top-bottom`, `bt` = `bottom-top`. The default is `left-right`.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

5.5.2 Properties

Name	Description
merges	Returns the performed merges in matrix format
names	Returns the names of the nodes in the dendrogram
Inherited from object	
__class__	

continued on next page

Name	Description

5.6 Class VertexDendrogram

object
└
igraph.clustering.Dendrogram
└
igraph.clustering.VertexDendrogram

The dendrogram resulting from the hierarchical clustering of the vertex set of a graph.

5.6.1 Methods

__init__(*self*, *graph*, *merges*, *optimal_count*=`None`, *params*=`None`, *modularity_params*=`None`)

Creates a dendrogram object for a given graph.

Parameters

`graph`:	the graph that will be associated to the clustering
`merges`:	the merges performed given in matrix form.
`optimal_count`:	the optimal number of clusters where the dendrogram should be cut. This is a hint usually provided by the clustering algorithm that produces the dendrogram. `None` means that such a hint is not available; the optimal count will then be selected based on the modularity in such a case.
`params`:	additional parameters to be stored in this object.
`modularity_params`:	arguments that should be passed to `Graph.modularity` when the modularity is (re)calculated. If the original graph was weighted, you should pass a dictionary containing a `weight` key with the appropriate value here.

Overrides: object.__init__

as_clustering(*self*, *n*=`None`)

Cuts the dendrogram at the given level and returns a corresponding `VertexClustering` object.

Parameters

`n`: the desired number of clusters. Merges are replayed from the beginning until the membership vector has exactly *n* distinct elements or until there are no more recorded merges, whichever happens first. If `None`, the optimal count hint given by the clustering algorithm will be used If the optimal count was not given either, it will be calculated by selecting the level where the modularity is maximal.

Return Value

a new `VertexClustering` object.

__plot__(*self*, *context*, *bbox*, *palette*, **args*, ***kwds*)

Draws the vertex dendrogram on the given Cairo context

See `Dendrogram.__plot__` for the list of supported keyword arguments.

Overrides: igraph.clustering.Dendrogram.__plot__

Inherited from igraph.clustering.Dendrogram(Section 5.5)

__str__(), format(), summary()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

5.6.2 Properties

Name	Description
optimal_count	Returns the optimal number of clusters for this dendrogram. If an optimal count hint was given at construction time, this property simply returns the hint. If such a count was not given, this method calculates the optimal number of clusters by maximizing the modularity along all the possible cuts in the dendrogram.
Inherited from igraph.clustering.Dendrogram (Section 5.5) merges, names	
Inherited from object	

continued on next page

Name	Description
__class__	

5.7 Class Cover

object
└ **igraph.clustering.Cover**

Known Subclasses: igraph.clustering.VertexCover

Class representing a cover of an arbitrary ordered set.

Covers are similar to clusterings, but each element of the set may belong to more than one cluster in a cover, and elements not belonging to any cluster are also allowed.

`Cover` instances provide a similar API as `Clustering` instances; for instance, iterating over a `Cover` will iterate over the clusters just like with a regular `Clustering` instance. However, they are not derived from each other or from a common superclass, and there might be functions that exist only in one of them or the other.

Clusters of an individual cover can be accessed by the `[]` operator:

```
>>> cl = Cover([[0,1,2,3], [2,3,4], [0,1,6]])
>>> cl[0]
[0, 1, 2, 3]
```

The membership vector can be accessed by the `membership` property. Note that contrary to `Clustering` instances, the membership vector will contain lists that contain the cluster indices each item belongs to:

```
>>> cl.membership
[[0, 2], [0, 2], [0, 1], [0, 1], [1], [], [2]]
```

The number of clusters can be retrieved by the `len` function:

```
>>> len(cl)
3
```

You can iterate over the cover as if it were a regular list of clusters:

```
>>> for cluster in cl:
...     print " ".join(str(idx) for idx in cluster)
...
0 1 2 3
2 3 4
0 1 6
```

If you need all the clusters at once as lists, you can simply convert the cover to a list:

```
>>> cluster_list = list(cl)
>>> print cluster_list
[[0, 1, 2, 3], [2, 3, 4], [0, 1, 6]]
```

`Clustering` objects can readily be converted to `Cover` objects using the constructor:

```
>>> clustering = Clustering([0, 0, 0, 0, 1, 1, 1, 2, 2, 2])
>>> cover = Cover(clustering)
>>> list(clustering) == list(cover)
True
```

5.7.1 Methods

__init__(*self*, *clusters*, *n*=0)

Constructs a cover with the given clusters.

Parameters

`clusters`: the clusters in this cover, as a list or iterable. Each cluster is specified by a list or tuple that contains the IDs of the items in this cluster. IDs start from zero.

`n`: the total number of elements in the set that is covered by this cover. If it is less than the number of unique elements found in all the clusters, we will simply use the number of unique elements, so it is safe to leave this at zero. You only have to specify this parameter if there are some elements that are covered by none of the clusters.

Overrides: object.__init__

__getitem__(*self*, *index*)

Returns the cluster with the given index.

__iter__(*self*)

Iterates over the clusters in this cover.

__len__(*self*)

Returns the number of clusters in this cover.

__str__(*self*)

Returns a string representation of the cover.

Overrides: object.__str__

size(*self*, *idx*)

Returns the size of a given cluster.

Parameters

`idx`: the cluster in which we are interested.

sizes(*self*, **args*)

Returns the size of given clusters.

The indices are given as positional arguments. If there are no positional arguments, the function will return the sizes of all clusters.

size_histogram(*self*, *bin_width*=`1`)

Returns the histogram of cluster sizes.

Parameters

`bin_width`: the bin width of the histogram

Return Value

a `Histogram` object

summary(*self*, *verbosity*=`0`, *width*=`None`)

Returns the summary of the cover.

The summary includes the number of items and clusters, and also the list of members for each of the clusters if the verbosity is nonzero.

Parameters

`verbosity`: determines whether the cluster members should be printed. Zero verbosity prints the number of items and clusters only.

Return Value

the summary of the cover as a string.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

5.7.2 Properties

Name	Description
membership	Returns the membership vector of this cover. The membership vector of a cover covering n elements is a list of length n, where element i contains the cluster indices of the ith item.
n	Returns the number of elements in the set covered by this cover.
Inherited from object __class__	

5.8 Class VertexCover

Known Subclasses: igraph.clustering.CohesiveBlocks

The cover of the vertex set of a graph.

This class extends `Cover` by linking it to a specific `Graph` object. It also provides some handy methods like getting the subgraph corresponding to a cluster and such.

Note: since this class is linked to a `Graph`, destroying the graph by the `del` operator does not free the memory occupied by the graph if there exists a `VertexCover` that references the `Graph`.

5.8.1 Methods

__init__(*self*, *graph*, *clusters*=`None`)

Creates a cover object for a given graph.

Parameters

`graph`: the graph that will be associated to the cover

`clusters`: the list of clusters. If `None`, it is assumed that there is only a single cluster that covers the whole graph.

Overrides: object.__init__

crossing(*self*)

Returns a boolean vector where element i is `True` iff edge i lies between clusters, `False` otherwise.

subgraph(*self*, *idx*)

Get the subgraph belonging to a given cluster.

Parameters

`idx`: the cluster index

Return Value

a copy of the subgraph

Precondition: the vertex set of the graph hasn't been modified since the moment the cover was constructed.

subgraphs(*self*)

Gets all the subgraphs belonging to each of the clusters.

Return Value

a list containing copies of the subgraphs

Precondition: the vertex set of the graph hasn't been modified since the moment the cover was constructed.

__plot__(*self*, *context*, *bbox*, *palette*, **args*, ***kwds*)

Plots the cover to the given Cairo context in the given bounding box.

This is done by calling `Graph.__plot__()` with the same arguments, but drawing nice colored blobs around the vertex groups.

This method understands all the positional and keyword arguments that are understood by `Graph.__plot__()`, only the differences will be highlighted here:

- `mark_groups`: whether to highlight the vertex clusters by colored polygons. Besides the values accepted by `Graph.__plot__` (i.e., a dict mapping colors to vertex indices, a list containing lists of vertex indices, or `False`), the following are also accepted:
  - `True`: all the clusters will be highlighted, the colors matching the corresponding color indices from the current palette (see the `palette` keyword argument of `Graph.__plot__`.
  - A dict mapping cluster indices or tuples of vertex indices to color names. The given clusters or vertex groups will be highlighted by the given colors.
  - A list of cluster indices. This is equivalent to passing a dict mapping numeric color indices from the current palette to cluster indices; therefore, the cluster referred to by element i of the list will be highlighted by color i from the palette.

  The value of the `plotting.mark_groups` configuration key is also taken into account here; if that configuration key is `True` and `mark_groups` is not given explicitly, it will automatically be set to `True`.

  In place of lists of vertex indices, you may also use `VertexSeq` instances.

  In place of color names, you may also use color indices into the current palette. `None` as a color name will mean that the corresponding group is ignored.
- `palette`: the palette used to resolve numeric color indices to RGBA values. By default, this is an instance of `ClusterColoringPalette`.

See Also: `Graph.__plot__()` for more supported keyword arguments.

Inherited from igraph.clustering.Cover(Section 5.7)

__getitem__(), __iter__(), __len__(), __str__(), size(), size_histogram(), sizes(), summary()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(),

__subclasshook__()

5.8.2 Properties

Name	Description
graph	Returns the graph belonging to this object
Inherited from igraph.clustering.Cover (Section 5.7) membership, n	
Inherited from object __class__	

5.9 Class CohesiveBlocks

object
└
igraph.clustering.Cover
└
igraph.clustering.VertexCover
└
igraph.clustering.CohesiveBlocks

The cohesive block structure of a graph.

Instances of this type are created by `Graph.cohesive_blocks()`. See the documentation of `Graph.cohesive_blocks()` for an explanation of what cohesive blocks are.

This class provides a few more methods that make handling of cohesive block structures easier.

5.9.1 Methods

__init__(*self*, *graph*, *blocks*=`None`, *cohesion*=`None`, *parent*=`None`)

Constructs a new cohesive block structure for the given graph.

If any of *blocks*, *cohesion* or *parent* is `None`, all the arguments will be ignored and `Graph.cohesive_blocks()` will be called to calculate the cohesive blocks. Otherwise, these three variables should describe the *result* of a cohesive block structure calculation. Chances are that you never have to construct `CohesiveBlocks` instances directly, just use `Graph.cohesive_blocks()`.

Parameters

`graph:` the graph itself

`blocks:` a list containing the blocks; each block is described as a list containing vertex IDs.

`cohesion:` the cohesion of each block. The length of this list must be equal to the length of *blocks*.

`parent:` the parent block of each block. Negative values or `None` mean that there is no parent block for that block. There should be only one parent block, which covers the entire graph.

Overrides: object.__init__

See Also: Graph.cohesive_blocks()

cohesion(*self*, *idx*)

Returns the cohesion of the group with the given index.

cohesions(*self*)

Returns the list of cohesion values for each group.

hierarchy(*self*)

Returns a new graph that describes the hierarchical relationships between the groups.

The new graph will be a directed tree; an edge will point from vertex i to vertex j if group i is a superset of group j. In other words, the edges point downwards.

max_cohesion(*self*, *idx*)

Finds the maximum cohesion score among all the groups that contain the given vertex.

max_cohesions(*self*)

For each vertex in the graph, returns the maximum cohesion score among all the groups that contain the vertex.

parent(*self*, *idx*)

Returns the parent group index of the group with the given index or `None` if the given group is the root.

parents(*self*)

Returns the list of parent group indices for each group or `None` if the given group is the root.

__plot__(*self*, *context*, *bbox*, *palette*, **args*, ***kwds*)

Plots the cohesive block structure to the given Cairo context in the given bounding box.

Since a `CohesiveBlocks` instance is also a `VertexCover`, keyword arguments accepted by `VertexCover.__plot__()` are also accepted here. The only difference is that the vertices are colored according to their maximal cohesions by default, and groups are marked by colored blobs except the last group which encapsulates the whole graph.

See the documentation of `VertexCover.__plot__()` for more details.

Overrides: igraph.clustering.VertexCover.__plot__

Inherited from igraph.clustering.VertexCover(Section 5.8)

crossing(), subgraph(), subgraphs()

Inherited from igraph.clustering.Cover(Section 5.7)

__getitem__(), __iter__(), __len__(), __str__(), size(), size_histogram(), sizes(), summary()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

5.9.2 Properties

Name	Description
Inherited from igraph.clustering.VertexCover (Section 5.8)	

continued on next page

Name	Description
graph	
Inherited from igraph.clustering.Cover (Section 5.7) membership, n	
Inherited from object __class__	

6 Module igraph.configuration

Configuration framework for igraph.

igraph has some parameters which usually affect the behaviour of many functions. This module provides the framework for altering and querying igraph parameters as well as saving them to and retrieving them from disk.

6.1 Functions

get_platform_image_viewer()

Returns the path of an image viewer on the given platform

get_user_config_file()

Returns the path where the user-level configuration file is stored

init()

Default mechanism to initiate igraph configuration

This method loads the user-specific configuration file from the user's home directory, or if it does not exist, creates a default configuration.

The method is safe to be called multiple times, it will not parse the configuration file twice.

Return Value

the `Configuration` object loaded or created.

6.2 Variables

Name	Description
__package__	**Value:** 'igraph'

6.3 Class Configuration

object
└
igraph.configuration.Configuration

Class representing igraph configuration details.

(section) General ideas

The configuration of igraph is stored in the form of name-value pairs. This object provides an interface to the configuration data using the syntax known from dict:

```
>>> c=Configuration()
>>> c["general.verbose"] = True
>>> print c["general.verbose"]
True
```

Configuration keys are organized into sections, and the name to be used for a given key is always in the form `section.keyname`, like `general.verbose` in the example above. In that case, `general` is the name of the configuration section, and `verbose` is the name of the key. If the name of the section is omitted, it defaults to `general`, so `general.verbose` can be referred to as `verbose`:

```
>>> c=Configuration()
>>> c["verbose"] = True
>>> print c["general.verbose"]
True
```

User-level configuration is stored in `~/.igraphrc` per default on Linux and Mac OS X systems, or in `C:\Documents and Settings\username\.igraphrc` on Windows systems. However, this configuration is read only when `igraph` is launched through its shell interface defined in `igraph.app.shell`. This behaviour might change before version 1.0.

(section) Known configuration keys

The known configuration keys are presented below, sorted by section. When referring to them in program code, don't forget to add the section name, expect in the case of section `general`.

(section) General settings

These settings are all stored in section `general`.

- **shells**: the list of preferred Python shells to be used with the command-line `igraph`

script. The shells in the list are tried one by one until any of them is found on the system. `igraph` functions are then imported into the main namespace of the shell and the shell is launched. Known shells and their respective class names to be used can be found in `igraph.app.shell`. Example: `IPythonShell, ClassicPythonShell`. This is the default, by the way.

- **verbose**: whether `igraph` should talk more than really necessary. For instance, if set to `True`, some functions display progress bars.

(section) Application settings

These settings specify the external applications that are possibly used by `igraph`. They are all stored in section `apps`.

- **image_viewer**: image viewer application. If set to an empty string, it will be determined automatically from the platform `igraph` runs on. On Mac OS X, it defaults to the Preview application. On Linux, it chooses a viewer from several well-known Linux viewers like `gthumb`, `kuickview` and so on (see the source code for the full list). On Windows, it defaults to the system's built-in image viewer.

(section) Plotting settings

These settings specify the default values used by plotting functions. They are all stored in section `plotting`.

- **layout**: default graph layout algorithm to be used.
- **mark_groups**: whether to mark the clusters by polygons when plotting a clustering object.
- **palette**: default palette to be used for converting integer numbers to colors. See `colors.Palette` for more information. Valid palette names are stored in `colors.palettes`.
- **wrap_labels**: whether to try to wrap the labels of the vertices automatically if they don't fit within the vertex. Default: `False`.

(section) Shell settings

These settings specify options for external environments in which igraph is embedded (e.g., IPython and its Qt console). These settings are stored in section `shell`.

- **ipython.inlining.Plot**: whether to show instances of the `Plot` class inline in IPython's console if the console supports it. Default: `True`

6.3.1 Methods

__init__(*self*, *filename*=`None`)

Creates a new configuration instance.

Parameters

`filename`: file or file pointer to be read. Can be omitted.

Overrides: object.__init__

__contains__(*self*, *item*)

Checks whether the given configuration item is set.

Parameters

`item`: the configuration key to check.

Return Value

`True` if the key has an associated value, `False` otherwise.

__getitem__(*self*, *item*)

Returns the given configuration item.

Parameters

`item`: the configuration key to retrieve.

Return Value

the configuration value

__setitem__(*self*, *item*, *value*)

Sets the given configuration item.

Parameters

`item`: the configuration key to set

`value`: the new value of the configuration key

__delitem__(*self*, *item*)

Deletes the given item from the configuration.

If the item has a default value, the default value is written back instead of the current value. Without a default value, the item is really deleted.

has_key(*self*, *item*)

Checks if the configuration has a given key.

Parameters

`item`: the key being sought

load(*self*, *stream*=`None`)

Loads the configuration from the given file.

Parameters

`stream`: name of a file or a file object. The configuration will be loaded from here. Can be omitted, in this case, the user-level configuration is loaded.

save(*self*, *stream*=`None`)

Saves the configuration.

Parameters

`stream`: name of a file or a file object. The configuration will be saved there. Can be omitted, in this case, the user-level configuration file will be overwritten.

instance(*cls*)

Returns the single instance of the configuration object.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

6.3.2 Properties

Name	Description
filename	Returns the filename associated to the object. It is usually the name of the configuration file that was used when creating the object. `Configuration.load` always overwrites it with the filename given to it. If `None`, the configuration was either created from scratch or it was updated from a stream without name information.
Inherited from object __class__	

7 Module igraph.cut

Classes representing cuts and flows on graphs.

7.1 Variables

Name	Description
__package__	**Value:** `'igraph'`

7.2 Class Cut

Known Subclasses: igraph.cut.Flow

A cut of a given graph.

This is a simple class used to represent cuts returned by `Graph.mincut()`, `Graph.all_st_cuts()` and other functions that calculate cuts.

A cut is a special vertex clustering with only two clusters. Besides the usual `VertexClustering` methods, it also has the following attributes:

- `value` - the value (capacity) of the cut. It is equal to the number of edges if there are no capacities on the edges.
- `partition` - vertex IDs in the parts created after removing edges in the cut
- `cut` - edge IDs in the cut

- `es` - an edge selector restricted to the edges in the cut.

You can use indexing on this object to obtain lists of vertex IDs for both sides of the partition.

This class is usually not instantiated directly, everything is taken care of by the functions that return cuts.

Examples:

```
>>> from igraph import Graph
>>> g = Graph.Ring(20)
>>> mc = g.mincut()
>>> print mc.value
2.0
>>> print min(map(len, mc))
1
>>> mc.es["color"] = "red"
```

7.2.1 Methods

__init__(*self*, *graph*, *value*=`None`, *cut*=`None`, *partition*=`None`, *partition2*=`None`)

Initializes the cut.

This should not be called directly, everything is taken care of by the functions that return cuts.

Parameters

`graph:`	the graph that will be associated to the clustering
`membership:`	the membership list. The length of the list must be equal to the number of vertices in the graph. If `None`, every vertex is assumed to belong to the same cluster.
`modularity:`	the modularity score of the clustering. If `None`, it will be calculated when needed.
`params:`	additional parameters to be stored in this object.
`modularity_params:`	arguments that should be passed to `Graph.modularity` when the modularity is (re)calculated. If the original graph was weighted, you should pass a dictionary containing a `weight` key with the appropriate value here.

Overrides: object.__init__

__repr__(*self*)

repr(x)

Overrides: object.__repr__ extit(inherited documentation)

__str__(*self*)

str(x)

Overrides: object.__str__ extit(inherited documentation)

Inherited from igraph.clustering.VertexClustering(Section 5.4)

FromAttribute(), __plot__(), as_cover(), cluster_graph(), crossing(), giant(), recalculate_modularity(), subgraph(), subgraphs()

Inherited from igraph.clustering.Clustering(Section 5.3)

__getitem__(), __iter__(), __len__(), compare_to(), size(), size_histogram(), sizes(), summary()

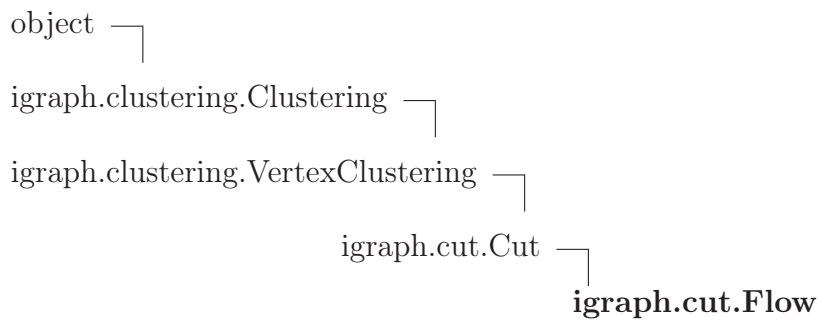
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __subclasshook__()

7.2.2 Properties

Name	Description
es	Returns an edge selector restricted to the cut
partition	Returns the vertex IDs partitioned according to the cut
cut	Returns the edge IDs in the cut
value	Returns the sum of edge capacities in the cut
Inherited from igraph.clustering.VertexClustering (Section 5.4) graph, modularity, q	
Inherited from igraph.clustering.Clustering (Section 5.3) membership, n	
Inherited from object __class__	

7.3 Class Flow



A flow of a given graph.

This is a simple class used to represent flows returned by `Graph.maxflow`. It has the following attributes:

- `graph` - the graph on which this flow is defined
- `value` - the value (capacity) of the flow
- `flow` - the flow values on each edge. For directed graphs, this is simply a list where element i corresponds to the flow on edge i. For undirected graphs, the direction of

the flow is not constrained (since the edges are undirected), hence positive flow always means a flow from the smaller vertex ID to the larger, while negative flow means a flow from the larger vertex ID to the smaller.

- `cut` - edge IDs in the minimal cut corresponding to the flow.
- `partition` - vertex IDs in the parts created after removing edges in the cut
- `es` - an edge selector restricted to the edges in the cut.

This class is usually not instantiated directly, everything is taken care of by `Graph.maxflow`.

Examples:

```
>>> from igraph import Graph
>>> g = Graph.Ring(20)
>>> mf = g.maxflow(0, 10)
>>> print mf.value
2.0
>>> mf.es["color"] = "red"
```

7.3.1 Methods

__init__(*self*, *graph*, *value*, *flow*, *cut*, *partition*)

Initializes the flow.

This should not be called directly, everything is taken care of by `Graph.maxflow`.

Parameters

`graph:`	the graph that will be associated to the clustering
`membership:`	the membership list. The length of the list must be equal to the number of vertices in the graph. If `None`, every vertex is assumed to belong to the same cluster.
`modularity:`	the modularity score of the clustering. If `None`, it will be calculated when needed.
`params:`	additional parameters to be stored in this object.
`modularity_params:`	arguments that should be passed to `Graph.modularity` when the modularity is (re)calculated. If the original graph was weighted, you should pass a dictionary containing a `weight` key with the appropriate value here.

Overrides: object.__init__

__repr__(*self*)

repr(x)

Overrides: object.__repr__ extit(inherited documentation)

__str__(*self*)

str(x)

Overrides: object.__str__ extit(inherited documentation)

Inherited from igraph.clustering.VertexClustering(Section 5.4)

FromAttribute(), __plot__(), as_cover(), cluster_graph(), crossing(), giant(), recalculate_modularity(), subgraph(), subgraphs()

Inherited from igraph.clustering.Clustering(Section 5.3)

__getitem__(), __iter__(), __len__(), compare_to(), size(), size_histogram(), sizes(), summary()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __subclasshook__()

7.3.2 Properties

Name	Description
flow	Returns the flow values for each edge. For directed graphs, this is simply a list where element *i* corresponds to the flow on edge *i*. For undirected graphs, the direction of the flow is not constrained (since the edges are undirected), hence positive flow always means a flow from the smaller vertex ID to the larger, while negative flow means a flow from the larger vertex ID to the smaller.
Inherited from igraph.cut.Cut (Section 7.2) cut, es, partition, value	
Inherited from igraph.clustering.VertexClustering (Section 5.4) graph, modularity, q	
Inherited from igraph.clustering.Clustering (Section 5.3) membership, n	
Inherited from object __class__	

8 Module igraph.datatypes

Additional auxiliary data types

8.1 Variables

Name	Description
__package__	**Value:** `'igraph'`

8.2 Class Matrix

object
└ **igraph.datatypes.Matrix**

Simple matrix data type.

Of course there are much more advanced matrix data types for Python (for instance, the `ndarray` data type of Numeric Python) and this implementation does not want to compete with them. The only role of this data type is to provide a convenient interface for the matrices returned by the `Graph` object (for instance, allow indexing with tuples in the case of adjacency matrices and so on).

8.2.1 Methods

__init__(*self*, *data*=`None`)

Initializes a matrix.

Parameters

`data`: the elements of the matrix as a list of lists, or `None` to create a 0x0 matrix.

Overrides: object.__init__

Fill(*cls*, *value*, **args*)

Creates a matrix filled with the given value

Parameters

`value`: the value to be used

`shape`: the shape of the matrix. Can be a single integer, two integers or a tuple. If a single integer is given here, the matrix is assumed to be square-shaped.

Zero(*cls*, **args*)

Creates a matrix filled with zeros.

Parameters

`shape`: the shape of the matrix. Can be a single integer, two integers or a tuple. If a single integer is given here, the matrix is assumed to be square-shaped.

Identity(*cls*, **args*)

Creates an identity matrix.

Parameters

`shape`: the shape of the matrix. Can be a single integer, two integers or a tuple. If a single integer is given here, the matrix is assumed to be square-shaped.

__add__(*self*, *other*)

Adds the given value to the matrix.

Parameters

`other`: either a scalar or a matrix. Scalars will be added to each element of the matrix. Matrices will be added together elementwise.

Return Value

the result matrix

__eq__(*self*, *other*)

Checks whether a given matrix is equal to another one

__getitem__(*self*, *i*)

Returns a single item, a row or a column of the matrix

Parameters

`i`: if a single integer, returns the *i*th row as a list. If a slice, returns the corresponding rows as another `Matrix` object. If a 2-tuple, the first element of the tuple is used to select a row and the second is used to select a column.

__hash__(*self*)

Returns a hash value for a matrix.

Overrides: object.__hash__

__iadd__(*self*, *other*)

In-place addition of a matrix or scalar.

__isub__(*self*, *other*)

In-place subtraction of a matrix or scalar.

__ne__(*self*, *other*)

Checks whether a given matrix is not equal to another one

__setitem__(*self*, *i*, *value*)

Sets a single item, a row or a column of the matrix

Parameters

`i`: if a single integer, sets the *i*th row as a list. If a slice, sets the corresponding rows from another `Matrix` object. If a 2-tuple, the first element of the tuple is used to select a row and the second is used to select a column.

`value`: the new value

__sub__(*self*, *other*)

Subtracts the given value from the matrix.

Parameters

`other`: either a scalar or a matrix. Scalars will be subtracted from each element of the matrix. Matrices will be subtracted together elementwise.

Return Value

the result matrix

__repr__(*self*)

repr(x)

Overrides: object.__repr__ extit(inherited documentation)

__str__(*self*)

str(x)

Overrides: object.__str__ extit(inherited documentation)

__iter__(*self*)

Support for iteration.

This is actually implemented as a generator, so there is no need for a separate iterator class. The generator returns *copies* of the rows in the matrix as lists to avoid messing around with the internals. Feel free to do anything with the copies, the changes won't be reflected in the original matrix.

__plot__(*self*, *context*, *bbox*, *palette*, ***kwds*)

Plots the matrix to the given Cairo context in the given box

Besides the usual self-explanatory plotting parameters (`context`, `bbox`, `palette`), it accepts the following keyword arguments:

- `style`: the style of the plot. `boolean` is useful for plotting matrices with boolean (`True`/`False` or 0/1) values: `False` will be shown with a white box and `True` with a black box. `palette` uses the given palette to represent numbers by colors, the minimum will be assigned to palette color index 0 and the maximum will be assigned to the length of the palette. `None` draws transparent cell backgrounds only. The default style is `boolean` (but it may change in the future). `None` values in the matrix are treated specially in both cases: nothing is drawn in the cell corresponding to `None`.
- `square`: whether the cells of the matrix should be square or not. Default is `True`.
- `grid_width`: line width of the grid shown on the matrix. If zero or negative, the grid is turned off. The grid is also turned off if the size of a cell is less than three times the given line width. Default is `1`. Fractional widths are also allowed.
- `border_width`: line width of the border drawn around the matrix. If zero or negative, the border is turned off. Default is `1`.
- `row_names`: the names of the rows
- `col_names`: the names of the columns.
- `values`: values to be displayed in the cells. If `None` or `False`, no values are displayed. If `True`, the values come from the matrix being plotted. If it is another matrix, the values of that matrix are shown in the cells. In this case, the shape of the value matrix must match the shape of the matrix being plotted.
- `value_format`: a format string or a callable that specifies how the values should be plotted. If it is a callable, it must be a function that expects a single value and returns a string. Example: `"%#.2f"` for floating-point numbers with always exactly two digits after the decimal point. See the Python documentation of the `%` operator for details on the format string. If the format string is not given, it defaults to the `str` function.

If only the row names or the column names are given and the matrix is square-shaped, the same names are used for both column and row names.

min(*self*, *dim*=`None`)

Returns the minimum of the matrix along the given dimension

Parameters

`dim`: the dimension. 0 means determining the column minimums, 1 means determining the row minimums. If `None`, the global minimum is returned.

max(*self*, *dim*=`None`)

Returns the maximum of the matrix along the given dimension

Parameters

`dim`: the dimension. 0 means determining the column maximums, 1 means determining the row maximums. If `None`, the global maximum is returned.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __subclasshook__()

8.2.2 Properties

Name	Description
data	Returns the data stored in the matrix as a list of lists
shape	Returns the shape of the matrix as a tuple
Inherited from object __class__	

8.3 Class DyadCensus

object
  tuple
    igraph.datatypes.DyadCensus

Dyad census of a graph.

This is a pretty simple class - basically it is a tuple, but it allows the user to refer to its individual items by the names `mutual` (or `mut`), `asymmetric` (or `asy` or `asym` or `asymm`) and `null`.

Examples:

```
>>> from igraph import Graph
```

```
>>> g=Graph.Erdos_Renyi(100, 0.2, directed=True)
>>> dc=g.dyad_census()
>>> print dc.mutual                #doctest:+SKIP
179
>>> print dc["asym"]               #doctest:+SKIP
1609
>>> print tuple(dc), list(dc)      #doctest:+SKIP
(179, 1609, 3162) [179, 1609, 3162]
>>> print sorted(dc.as_dict().items())    #doctest:+ELLIPSIS
[('asymmetric', ...), ('mutual', ...), ('null', ...)]
```

8.3.1 Methods

__getitem__(*self*, *idx*)

x[y]

Overrides: tuple.__getitem__ extit(inherited documentation)

__getattr__(*self*, *attr*)

__repr__(*self*)

repr(x)

Overrides: object.__repr__ extit(inherited documentation)

__str__(*self*)

str(x)

Overrides: object.__str__ extit(inherited documentation)

as_dict(*self*)

Converts the dyad census to a dict using the known dyad names.

Inherited from tuple

__add__(), __contains__(), __eq__(), __ge__(), __getattribute__(), __getnewargs__(), __getslice__(), __gt__(), __hash__(), __iter__(), __le__(), __len__(), __lt__(), __mul__(), __ne__(), __new__(), __rmul__(), count(), index()

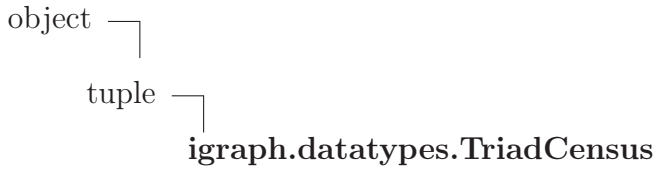
Inherited from object

__delattr__(), __format__(), __init__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __subclasshook__()

8.3.2 Properties

Name	Description
Inherited from object	
__class__	

8.4 Class TriadCensus



Triad census of a graph.

This is a pretty simple class - basically it is a tuple, but it allows the user to refer to its individual items by the following triad names:

- `003` – the empty graph
- `012` – a graph with a single directed edge (`A -> B, C`)
- `102` – a graph with a single mutual edge (`A <-> B, C`)
- `021D` – the binary out-tree (`A <- B -> C`)
- `021U` – the binary in-tree (`A -> B <- C`)
- `021C` – the directed line (`A -> B -> C`)
- `111D` – `A <-> B <- C`
- `111U` – `A <-> B -> C`
- `030T` – `A -> B <- C, A -> C`
- `030C` – `A <- B <- C, A -> C`
- `201` – `A <-> B <-> C`
- `120D` – `A <- B -> C, A <-> C`
- `120U` – `A -> B <- C, A <-> C`
- `120C` – `A -> B -> C, A <-> C`
- `210C` – `A -> B <-> C, A <-> C`
- `300` – the complete graph (`A <-> B <-> C, A <-> C`)

Attribute and item accessors are provided. Due to the syntax of Python, attribute names are not allowed to start with a number, therefore the triad names must be prepended with a lowercase `t` when accessing them as attributes. This is not necessary with the item accessor syntax.

Examples:

```
>>> from igraph import Graph
```

```
>>> g=Graph.Erdos_Renyi(100, 0.2, directed=True)
>>> tc=g.triad_census()
>>> print tc.t003                          #doctest:+SKIP
39864
>>> print tc["030C"]                       #doctest:+SKIP
1206
```

8.4.1 Methods

__getitem__(*self*, *idx*)

x[y]

Overrides: tuple.__getitem__ extit(inherited documentation)

__getattr__(*self*, *attr*)

__repr__(*self*)

repr(x)

Overrides: object.__repr__ extit(inherited documentation)

__str__(*self*)

str(x)

Overrides: object.__str__ extit(inherited documentation)

Inherited from tuple

__add__(), __contains__(), __eq__(), __ge__(), __getattribute__(), __getnewargs__(), __getslice__(), __gt__(), __hash__(), __iter__(), __le__(), __len__(), __lt__(), __mul__(), __ne__(), __new__(), __rmul__(), count(), index()

Inherited from object

__delattr__(), __format__(), __init__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __subclasshook__()

8.4.2 Properties

Name	Description
Inherited from object	
__class__	

8.5 Class UniqueIdGenerator

object
└ **igraph.datatypes.UniqueIdGenerator**

A dictionary-like class that can be used to assign unique IDs to names (say, vertex names).

Usage:

```
>>> gen = UniqueIdGenerator()
>>> gen["A"]
0
>>> gen["B"]
1
>>> gen["C"]
2
>>> gen["A"]       # Retrieving already existing ID
0
>>> gen.add("D")   # Synonym of gen["D"]
3
>>> len(gen)       # Number of already used IDs
4
>>> "C" in gen
True
>>> "E" in gen
False
```

8.5.1 Methods

__init__(*self*, *id_generator*=None, *initial*=None)

Creates a new unique ID generator. ‘id_generator‘ specifies how do we assign new IDs to elements that do not have an ID yet. If it is ‘None‘, elements will be assigned integer identifiers starting from 0. If it is an integer, elements will be assigned identifiers starting from the given integer. If it is an iterator or generator, its ‘next‘ method will be called every time a new ID is needed.

Overrides: object.__init__

__contains__(*self*, *item*)

Checks whether ‘item‘ already has an ID or not.

__getitem__(*self*, *item*)

Retrieves the ID corresponding to ‘item‘. Generates a new ID for ‘item‘ if it is the first time we request an ID for it.

__setitem__(*self*, *item*, *value*)

Overrides the ID for ‘item‘.

__len__(*self*)

"Returns the number of items

reverse_dict(*self*)

Returns the reverse mapping, i.e., the one that maps from generated IDs to their corresponding objects

values(*self*)

Returns the values stored so far. If the generator generates items according to the standard sorting order, the values returned will be exactly in the order they were added. This holds for integer IDs for instance (but for many other ID generators as well).

add(*self*, *item*)

Retrieves the ID corresponding to ‘item‘. Generates a new ID for ‘item‘ if it is the first time we request an ID for it.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

8.5.2 Properties

Name	Description
Inherited from object	
__class__	

9 Package igraph.drawing

Drawing and plotting routines for IGraph.

Plotting is dependent on the `pycairo` or `cairocffi` libraries that provide Python bindings to the popular Cairo library[1]. This means that if you don't have pycairo[2] or cairocffi[3] installed, you won't be able to use the plotting capabilities. However, you can still use `Graph.write_svg` to save the graph to an SVG file and view it from Mozilla Firefox[4] (free) or edit it in Inkscape[5] (free), Skencil[6] (formerly known as Sketch, also free) or Adobe Illustrator.

Whenever the documentation refers to the `pycairo` library, you can safely replace it with `cairocffi` as the two are API-compatible.

License: GPL

9.1 Modules

- **baseclasses**: Abstract base classes for the drawing routines.
  (Section 10, p. 312)
- **colors**: Color handling functions.
  (Section 11, p. 315)
- **coord**: Coordinate systems and related plotting routines
  (Section 12, p. 328)
- **edge**: Drawers for various edge styles in graph plots.
  (Section 13, p. 331)
- **graph**: Drawing routines to draw graphs.
  (Section 14, p. 339)
- **metamagic**: Auxiliary classes for the default graph drawer in igraph.
  (Section 15, p. 346)
- **shapes**: Shape drawing classes for igraph
  (Section 16, p. 349)
- **text**: Drawers for labels on plots.
  (Section 17, p. 352)
- **utils**: Utility classes for drawing routines.
  (Section 18, p. 357)
- **vertex**: Drawing routines to draw the vertices of graphs.
  (Section 19, p. 368)

[1] http://www.cairographics.org
[2] http://www.cairographics.org/pycairo
[3] http://cairocffi.readthedocs.io
[4] http://www.mozilla.org/firefox
[5] http://www.inkscape.org
[6] http://www.skencil.org

9.2 Functions

plot(*obj*, *target*=`None`, *bbox*=`(0, 0, 600, 600)`, **args*, ***kwds*)

Plots the given object to the given target.

Positional and keyword arguments not explicitly mentioned here will be passed down to the `__plot__` method of the object being plotted. Since you are most likely interested in the keyword arguments available for graph plots, see `Graph.__plot__` as well.

Parameters

`obj`: the object to be plotted

`target`: the target where the object should be plotted. It can be one of the following types:

- `None` – an appropriate surface will be created and the object will be plotted there.
- `cairo.Surface` – the given Cairo surface will be used. This can refer to a PNG image, an arbitrary window, an SVG file, anything that Cairo can handle.
- `string` – a file with the given name will be created and an appropriate Cairo surface will be attached to it. The supported image formats are: PNG, PDF, SVG and PostScript.

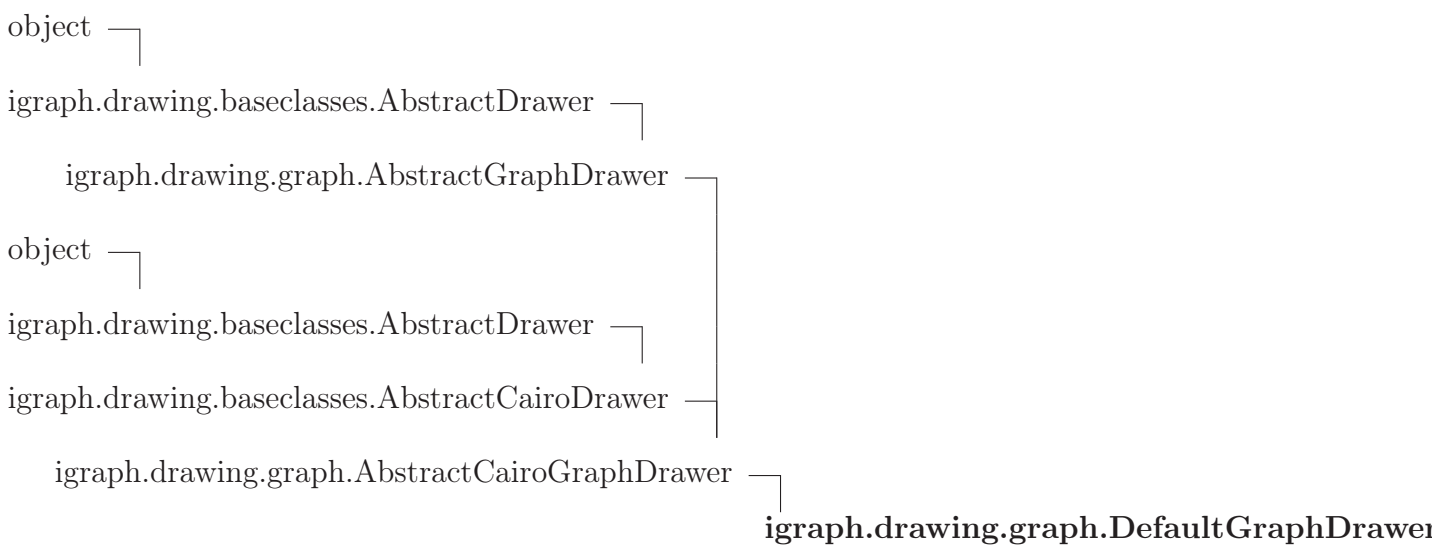
`bbox`: the bounding box of the plot. It must be a tuple with either two or four integers, or a `BoundingBox` object. If this is a tuple with two integers, it is interpreted as the width and height of the plot (in pixels for PNG images and on-screen plots, or in points for PDF, SVG and PostScript plots, where 72 pt = 1 inch = 2.54 cm). If this is a tuple with four integers, the first two denotes the X and Y coordinates of a corner and the latter two denoting the X and Y coordinates of the opposite corner.

`opacity`: the opacity of the object being plotted. It can be used to overlap several plots of the same graph if you use the same layout for them – for instance, you might plot a graph with opacity 0.5 and then plot its spanning tree over it with opacity 0.1. To achieve this, you'll need to modify the `Plot` object returned with `Plot.add`.

`palette`: the palette primarily used on the plot if the added objects do not specify a private palette. Must be either an `igraph.drawing.colors.Palette` object or a string referring to a valid key of `igraph.drawing.colors.palettes` (see module `igraph.drawing.colors`) or `None`. In the latter case, the default palette given by the configuration key `plotting.palette` is used.

`margin`: the top, right, bottom, left margins as a 4-tuple. If it has less than 4 elements or is a single float, the elements will be re-used until the length is at least 4. The default

9.3 Class DefaultGraphDrawer



Class implementing the default visualisation of a graph.

The default visualisation of a graph draws the nodes on a 2D plane according to a given `Layout`, then draws a straight or curved edge between nodes connected by edges. This is the visualisation used when one invokes the `plot()` function on a `Graph` object.

See `Graph.__plot__()` for the keyword arguments understood by this drawer.

9.3.1 Methods

__init__(*self*, *context*, *bbox*, *vertex_drawer_factory*=<class 'igraph.drawing.vertex.DefaultVertexDrawer'>, *edge_drawer_factory*=<class 'igraph.drawing.edge.ArrowEdgeDrawer'>, *label_drawer_factory*=<class 'igraph.drawing.text.TextDrawer'>)

Constructs the graph drawer and associates it to the given Cairo context and the given `BoundingBox`.

Parameters

`context:`	the context on which we will draw
`bbox:`	the bounding box within which we will draw. Can be anything accepted by the constructor of `BoundingBox` (i.e., a 2-tuple, a 4-tuple or a `BoundingBox` object).
`vertex_drawer_factory:`	a factory method that returns an `AbstractCairoVertexDrawer` instance bound to a given Cairo context. The factory method must take three parameters: the Cairo context, the bounding box of the drawing area and the palette to be used for drawing colored vertices. The default vertex drawer is `DefaultVertexDrawer`.
`edge_drawer_factory:`	a factory method that returns an `AbstractEdgeDrawer` instance bound to a given Cairo context. The factory method must take two parameters: the Cairo context and the palette to be used for drawing colored edges. You can use any of the actual `AbstractEdgeDrawer` implementations here to control the style of edges drawn by igraph. The default edge drawer is `ArrowEdgeDrawer`.
`label_drawer_factory:`	a factory method that returns a `TextDrawer` instance bound to a given Cairo context. The method must take one parameter: the Cairo context. The default label drawer is `TextDrawer`.

Overrides: object.__init__

draw(*self*, *graph*, *palette*, **args*, ***kwds*)

Abstract method, must be implemented in derived classes.

Overrides: igraph.drawing.baseclasses.AbstractDrawer.draw extit(inherited documentation)

Inherited from igraph.drawing.graph.AbstractGraphDrawer

ensure_layout()

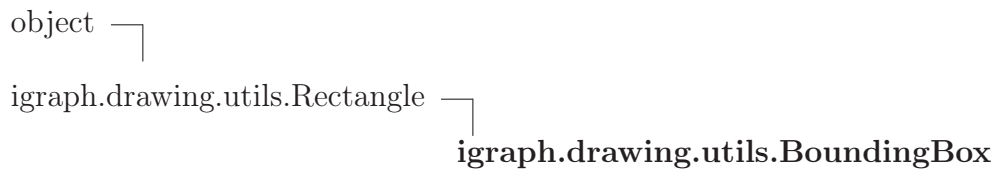
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

9.3.2 Properties

Name	Description
Inherited from igraph.drawing.baseclasses.AbstractCairoDrawer (Section 10.3)	
bbox	
Inherited from object	
__class__	

9.4 Class BoundingBox



Class representing a bounding box (a rectangular area) that encloses some objects.

9.4.1 Methods

__ior__(*self*, *other*)

Replaces this bounding box with the union of itself and another.

Example:

```
>>> box1 = BoundingBox(10, 20, 50, 60)
>>> box2 = BoundingBox(70, 40, 100, 90)
>>> box1 |= box2
>>> print(box1)
BoundingBox(10.0, 20.0, 100.0, 90.0)
```

Overrides: igraph.drawing.utils.Rectangle.__ior__

__or__(*self*, *other*)

Takes the union of this bounding box with another.

The result is a bounding box which encloses both bounding boxes.

Example:

```
>>> box1 = BoundingBox(10, 20, 50, 60)
>>> box2 = BoundingBox(70, 40, 100, 90)
>>> box1 | box2
BoundingBox(10.0, 20.0, 100.0, 90.0)
```

Overrides: igraph.drawing.utils.Rectangle.__or__

Inherited from igraph.drawing.utils.Rectangle(Section 18.1)

__and__(), __bool__(), __eq__(), __hash__(), __init__(), __ne__(), __nonzero__(), __repr__(), contract(), expand(), intersection(), isdisjoint(), isempty(), translate(), union()

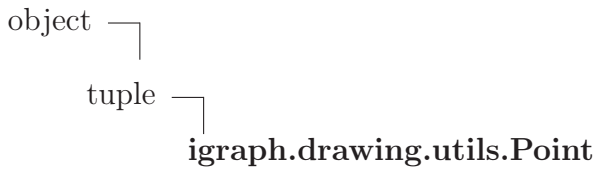
Inherited from object

__delattr__(), __format__(), __getattribute__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

9.4.2 Properties

Name	Description
Inherited from igraph.drawing.utils.Rectangle (Section 18.1)	
bottom, coords, height, left, midx, midy, right, shape, top, width	
Inherited from object	
__class__	

9.5 Class Point



Class representing a point on the 2D plane.

9.5.1 Methods

__new__(*cls*, *x*, *y*)

Creates a new point with the given coordinates

Return Value

a new object with type S, a subtype of T

Overrides: object.__new__

__repr__(*self*)

Returns a nicely formatted representation of the point

Overrides: object.__repr__

__getnewargs__(*self*)

Return self as a plain tuple. Used by copy and pickle.

Overrides: tuple.__getnewargs__

__add__(*self*, *other*)

Adds the coordinates of a point to another one

Overrides: tuple.__add__

__sub__(*self*, *other*)

Subtracts the coordinates of a point to another one

__mul__(*self*, *scalar*)

Multiplies the coordinates by a scalar

Overrides: tuple.__mul__

__rmul__(*self*, *scalar*)

Multiplies the coordinates by a scalar

Overrides: tuple.__rmul__

__div__(*self*, *scalar*)

Divides the coordinates by a scalar

as_polar(*self*)

Returns the polar coordinate representation of the point.

Return Value

the radius and the angle in a tuple.

distance(*self*, *other*)

Returns the distance of the point from another one.

Example:

```
>>> p1 = Point(5, 7)
>>> p2 = Point(8, 3)
>>> p1.distance(p2)
5.0
```

interpolate(*self*, *other*, *ratio*=`0.5`)

Linearly interpolates between the coordinates of this point and another one.

Parameters

`other`: the other point

`ratio`: the interpolation ratio between 0 and 1. Zero will return this point, 1 will return the other point.

length(*self*)

Returns the length of the vector pointing from the origin to this point.

normalized(*self*)

Normalizes the coordinates of the point s.t. its length will be 1 after normalization. Returns the normalized point.

sq_length(*self*)

Returns the squared length of the vector pointing from the origin to this point.

towards(*self*, *other*, *distance*=0)

Returns the point that is at a given distance from this point towards another one.

FromPolar(*cls*, *radius*, *angle*)

Constructs a point from polar coordinates.

'radius' is the distance of the point from the origin; 'angle' is the angle between the X axis and the vector pointing to the point from the origin.

Inherited from tuple

__contains__(), __eq__(), __ge__(), __getattribute__(), __getitem__(), __getslice__(), __gt__(), __hash__(), __iter__(), __le__(), __len__(), __lt__(), __ne__(), count(), index()

Inherited from object

__delattr__(), __format__(), __init__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

9.5.2 Properties

Name	Description
x	Alias for field number 0
y	Alias for field number 1
Inherited from object __class__	

9.6 Class Rectangle

object
└ **igraph.drawing.utils.Rectangle**

Known Subclasses: igraph.drawing.utils.BoundingBox

Class representing a rectangle.

9.6.1 Methods

__init__(*self*, **args*)

Creates a rectangle.

The corners of the rectangle can be specified by either a tuple (four items, two for each corner, respectively), four separate numbers (X and Y coordinates for each corner) or two separate numbers (width and height, the upper left corner is assumed to be at (0,0))

Overrides: object.__init__

contract(*self*, *margins*)

Contracts the rectangle by the given margins.

Return Value

a new `Rectangle` object.

expand(*self*, *margins*)

Expands the rectangle by the given margins.

Return Value

a new `Rectangle` object.

isdisjoint(*self*, *other*)

Returns "True" if the two rectangles have no intersection.

Example:

```
>>> r1 = Rectangle(10, 10, 30, 30)
>>> r2 = Rectangle(20, 20, 50, 50)
>>> r3 = Rectangle(70, 70, 90, 90)
>>> r1.isdisjoint(r2)
False
>>> r2.isdisjoint(r1)
False
>>> r1.isdisjoint(r3)
True
>>> r3.isdisjoint(r1)
True
```

isempty(*self*)

Returns "True" if the rectangle is empty (i.e. it has zero width and height).

Example:

```
>>> r1 = Rectangle(10, 10, 30, 30)
>>> r2 = Rectangle(70, 70, 90, 90)
>>> r1.isempty()
False
>>> r2.isempty()
False
>>> r1.intersection(r2).isempty()
True
```

intersection(*self*, *other*)

Returns the intersection of this rectangle with another.

Example:

```
>>> r1 = Rectangle(10, 10, 30, 30)
>>> r2 = Rectangle(20, 20, 50, 50)
>>> r3 = Rectangle(70, 70, 90, 90)
>>> r1.intersection(r2)
Rectangle(20.0, 20.0, 30.0, 30.0)
>>> r2 & r1
Rectangle(20.0, 20.0, 30.0, 30.0)
>>> r2.intersection(r1) == r1.intersection(r2)
True
>>> r1.intersection(r3)
Rectangle(0.0, 0.0, 0.0, 0.0)
```

__and__(*self*, *other*)

Returns the intersection of this rectangle with another.

Example:

```
>>> r1 = Rectangle(10, 10, 30, 30)
>>> r2 = Rectangle(20, 20, 50, 50)
>>> r3 = Rectangle(70, 70, 90, 90)
>>> r1.intersection(r2)
Rectangle(20.0, 20.0, 30.0, 30.0)
>>> r2 & r1
Rectangle(20.0, 20.0, 30.0, 30.0)
>>> r2.intersection(r1) == r1.intersection(r2)
True
>>> r1.intersection(r3)
Rectangle(0.0, 0.0, 0.0, 0.0)
```

translate(*self*, *dx*, *dy*)

Translates the rectangle in-place.

Example:

```
>>> r = Rectangle(10, 20, 50, 70)
>>> r.translate(30, -10)
>>> r
Rectangle(40.0, 10.0, 80.0, 60.0)
```

Parameters

`dx`: the X coordinate of the translation vector

`dy`: the Y coordinate of the translation vector

union(*self*, *other*)

Returns the union of this rectangle with another.

The resulting rectangle is the smallest rectangle that contains both rectangles.

Example:

```
>>> r1 = Rectangle(10, 10, 30, 30)
>>> r2 = Rectangle(20, 20, 50, 50)
>>> r3 = Rectangle(70, 70, 90, 90)
>>> r1.union(r2)
Rectangle(10.0, 10.0, 50.0, 50.0)
>>> r2 | r1
Rectangle(10.0, 10.0, 50.0, 50.0)
>>> r2.union(r1) == r1.union(r2)
True
>>> r1.union(r3)
Rectangle(10.0, 10.0, 90.0, 90.0)
```

__or__(*self*, *other*)

Returns the union of this rectangle with another.

The resulting rectangle is the smallest rectangle that contains both rectangles.

Example:

```
>>> r1 = Rectangle(10, 10, 30, 30)
>>> r2 = Rectangle(20, 20, 50, 50)
>>> r3 = Rectangle(70, 70, 90, 90)
>>> r1.union(r2)
Rectangle(10.0, 10.0, 50.0, 50.0)
>>> r2 | r1
Rectangle(10.0, 10.0, 50.0, 50.0)
>>> r2.union(r1) == r1.union(r2)
True
>>> r1.union(r3)
Rectangle(10.0, 10.0, 90.0, 90.0)
```

__ior__(*self*, *other*)

Expands this rectangle to include itself and another completely while still being as small as possible.

Example:

```
>>> r1 = Rectangle(10, 10, 30, 30)
>>> r2 = Rectangle(20, 20, 50, 50)
>>> r3 = Rectangle(70, 70, 90, 90)
>>> r1 |= r2
>>> r1
Rectangle(10.0, 10.0, 50.0, 50.0)
>>> r1 |= r3
>>> r1
Rectangle(10.0, 10.0, 90.0, 90.0)
```

__repr__(*self*)

repr(x)

Overrides: object.__repr__ extit(inherited documentation)

__eq__(*self*, *other*)

__ne__(*self*, *other*)

__bool__(*self*)

__nonzero__(*self*)

__hash__(*self*)

hash(x)

Overrides: object.__hash__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

9.6.2 Properties

Name	Description
coords	The coordinates of the corners. The coordinates are returned as a 4-tuple in the following order: left edge, top edge, right edge, bottom edge.
width	The width of the rectangle
height	The height of the rectangle
left	The X coordinate of the left side of the box
right	The X coordinate of the right side of the box
top	The Y coordinate of the top edge of the box
bottom	The Y coordinate of the bottom edge of the box
midx	The X coordinate of the center of the box
midy	The Y coordinate of the center of the box
shape	The shape of the rectangle (width, height)
Inherited from object __class__	

9.7 Class Plot

object
└
igraph.drawing.Plot

Class representing an arbitrary plot

Every plot has an associated surface object where the plotting is done. The surface is an instance of `cairo.Surface`, a member of the `pycairo` library. The surface itself provides a unified API to various plotting targets like SVG files, X11 windows, PostScript files, PNG files and so on. `igraph` usually does not know on which surface it is plotting right now, since `pycairo` takes care of the actual drawing. Everything that's supported by `pycairo` should be supported by this class as well.

Current Cairo surfaces that I'm aware of are:

- `cairo.GlitzSurface` – OpenGL accelerated surface for the X11 Window System.
- `cairo.ImageSurface` – memory buffer surface. Can be written to a `PNG` image file.
- `cairo.PDFSurface` – PDF document surface.
- `cairo.PSSurface` – PostScript document surface.
- `cairo.SVGSurface` – SVG (Scalable Vector Graphics) document surface.
- `cairo.Win32Surface` – Microsoft Windows screen rendering.
- `cairo.XlibSurface` – X11 Window System screen rendering.

If you create a `Plot` object with a string given as the target surface, the string will be treated as a filename, and its extension will decide which surface class will be used. Please note that

not all surfaces might be available, depending on your `pycairo` installation.

A `Plot` has an assigned default palette (see `igraph.drawing.colors.Palette`) which is used for plotting objects.

A `Plot` object also has a list of objects to be plotted with their respective bounding boxes, palettes and opacities. Palettes assigned to an object override the default palette of the plot. Objects can be added by the `Plot.add` method and removed by the `Plot.remove` method.

9.7.1 Methods

__init__(*self*, *target*=`None`, *bbox*=`None`, *palette*=`None`, *background*=`None`)

Creates a new plot.

Parameters

`target`: the target surface to write to. It can be one of the following types:

- `None` – an appropriate surface will be created and the object will be plotted there.
- `cairo.Surface` – the given Cairo surface will be used.
- `string` – a file with the given name will be created and an appropriate Cairo surface will be attached to it.

`bbox`: the bounding box of the surface. It is interpreted differently with different surfaces: PDF and PS surfaces will treat it as points (1 point = 1/72 inch). Image surfaces will treat it as pixels. SVG surfaces will treat it as an abstract unit, but it will mostly be interpreted as pixels when viewing the SVG file in Firefox.

`palette`: the palette primarily used on the plot if the added objects do not specify a private palette. Must be either an `igraph.drawing.colors.Palette` object or a string referring to a valid key of `igraph.drawing.colors.palettes` (see module `igraph.drawing.colors`) or `None`. In the latter case, the default palette given by the configuration key `plotting.palette` is used.

`background`: the background color. If `None`, the background will be transparent. You can use any color specification here that is understood by `igraph.drawing.colors.color_name_to_rgba`.

Overrides: object.__init__

add(*self*, *obj*, *bbox*=`None`, *palette*=`None`, *opacity*=`1.0`, **args*, ***kwds*)

Adds an object to the plot.

Arguments not specified here are stored and passed to the object's plotting function when necessary. Since you are most likely interested in the arguments acceptable by graphs, see `Graph.__plot__` for more details.

Parameters

`obj`: the object to be added

`bbox`: the bounding box of the object. If `None`, the object will fill the entire area of the plot.

`palette`: the color palette used for drawing the object. If the object tries to get a color assigned to a positive integer, it will use this palette. If `None`, defaults to the global palette of the plot.

`opacity`: the opacity of the object being plotted, in the range 0.0-1.0

See Also: Graph.__plot__

remove(*self*, *obj*, *bbox*=`None`, *idx*=`1`)

Removes an object from the plot.

If the object has been added multiple times and no bounding box was specified, it removes the instance which occurs *idx*th in the list of identical instances of the object.

Parameters

`obj`: the object to be removed

`bbox`: optional bounding box specification for the object. If given, only objects with exactly this bounding box will be considered.

`idx`: if multiple objects match the specification given by *obj* and *bbox*, only the *idx*th occurrence will be removed.

Return Value

`True` if the object has been removed successfully, `False` if the object was not on the plot at all or *idx* was larger than the count of occurrences

mark_dirty(*self*)

Marks the plot as dirty (should be redrawn)

redraw(*self*, *context*=`None`)

Redraws the plot

save(*self*, *fname*=`None`)

Saves the plot.

Parameters

`fname`: the filename to save to. It is ignored if the surface of the plot is not an `ImageSurface`.

show(*self*)

Saves the plot to a temporary file and shows it.

_repr_svg_(*self*)

Returns an SVG representation of this plot as a string.

This method is used by IPython to display this plot inline.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

9.7.2 Properties

Name	Description
background	Returns the background color of the plot. `None` means a transparent background.
bounding_box	Returns the bounding box of the Cairo surface as a `BoundingBox` object
height	Returns the height of the Cairo surface on which the plot is drawn
surface	Returns the Cairo surface on which the plot is drawn
width	Returns the width of the Cairo surface on which the plot is drawn
Inherited from object __class__	

10 Module igraph.drawing.baseclasses

Abstract base classes for the drawing routines.

10.1 Variables

Name	Description
__package__	**Value:** `'igraph.drawing'`

10.2 Class AbstractDrawer

object
└ **igraph.drawing.baseclasses.AbstractDrawer**

Known Subclasses: igraph.drawing.graph.AbstractGraphDrawer, igraph.drawing.baseclasses.AbstractCa
igraph.drawing.vertex.AbstractVertexDrawer, igraph.drawing.baseclasses.AbstractXMLRPCDrawer

Abstract class that serves as a base class for anything that draws an igraph object.

10.2.1 Methods

draw(*self*, **args*, ***kwds*)

Abstract method, must be implemented in derived classes.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

10.2.2 Properties

Name	Description
Inherited from object	
__class__	

10.3 Class AbstractCairoDrawer

object
└
igraph.drawing.baseclasses.AbstractDrawer
└ **igraph.drawing.baseclasses.AbstractCairoDrawer**

Known Subclasses: igraph.drawing.graph.AbstractCairoGraphDrawer, igraph.drawing.vertex.AbstractC

igraph.drawing.text.TextDrawer, igraph.drawing.shapes.PolygonDrawer, igraph.drawing.coord.CoordinateS

Abstract class that serves as a base class for anything that draws on a Cairo context within a given bounding box.

A subclass of `AbstractCairoDrawer` is guaranteed to have an attribute named `context` that represents the Cairo context to draw on, and an attribute named `bbox` for the `BoundingBox` of the drawing area.

10.3.1 Methods

__init__(*self*, *context*, *bbox*)

Constructs the drawer and associates it to the given Cairo context and the given `BoundingBox`.

Parameters

`context`: the context on which we will draw

`bbox`: the bounding box within which we will draw. Can be anything accepted by the constructor of `BoundingBox` (i.e., a 2-tuple, a 4-tuple or a `BoundingBox` object).

Overrides: object.__init__

draw(*self*, **args*, ***kwds*)

Abstract method, must be implemented in derived classes.

Overrides: igraph.drawing.baseclasses.AbstractDrawer.draw

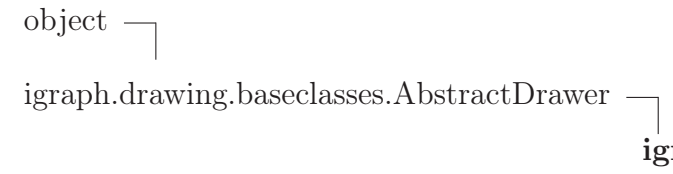
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

10.3.2 Properties

Name	Description
bbox	The bounding box of the drawing area where this drawer will draw.
Inherited from object __class__	

10.4 Class AbstractXMLRPCDrawer



Known Subclasses: igraph.drawing.graph.CytoscapeGraphDrawer, igraph.drawing.graph.UbiGraphDraw

Abstract drawer that uses a remote service via XML-RPC to draw something on a remote display.

10.4.1 Methods

__init__(*self*, *url*, *service*=`None`)

Constructs an abstract drawer using the XML-RPC service at the given URL.

Parameters

`url`: the URL where the XML-RPC calls for the service should be addressed to.

`service`: the name of the service at the XML-RPC address. If `None`, requests will be directed to the server proxy object constructed by `xmlrpclib.ServerProxy`; if not `None`, the given attribute will be looked up in the server proxy object.

Overrides: object.__init__

Inherited from igraph.drawing.baseclasses.AbstractDrawer(Section 10.2)

draw()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

10.4.2 Properties

Name	Description
Inherited from object	
__class__	

11 Module igraph.drawing.colors

Color handling functions.

11.1 Functions

color_name_to_rgb(*color*, *palette*=`None`)

Converts a color given in one of the supported color formats to R-G-B values.

This is done by calling `color_name_to_rgba` and then throwing away the alpha value.

See Also: color_name_to_rgba for more details about what formats are understood by this function.

color_name_to_rgba(*color*, *palette*=None)

Converts a color given in one of the supported color formats to R-G-B-A values.

Examples:

```
>>> color_name_to_rgba("red")
(1.0, 0.0, 0.0, 1.0)
>>> color_name_to_rgba("#ff8000") == (1.0, 128/255.0, 0.0, 1.0)
True
>>> color_name_to_rgba("#ff800080") == (1.0, 128/255.0, 0.0, 128/255.0)
True
>>> color_name_to_rgba("#08f") == (0.0, 136/255.0, 1.0, 1.0)
True
>>> color_name_to_rgba("rgb(100%, 50%, 0%)")
(1.0, 0.5, 0.0, 1.0)
>>> color_name_to_rgba("rgba(100%, 50%, 0%, 25%)")
(1.0, 0.5, 0.0, 0.25)
>>> color_name_to_rgba("hsla(120, 100%, 50%, 0.5)")
(0.0, 1.0, 0.0, 0.5)
>>> color_name_to_rgba("hsl(60, 100%, 50%)")
(1.0, 1.0, 0.0, 1.0)
>>> color_name_to_rgba("hsv(60, 100%, 100%)")
(1.0, 1.0, 0.0, 1.0)
```

Parameters

`color`: the color to be converted in one of the following formats:

- **CSS3 color specification**: `#rrggbb`, `#rgb`, `#rrggbbaa`, `#rgba`, `rgb(red, green, blue)`, `rgba(red, green, blue, alpha)`, `hsl(hue, saturation, lightness)`, `hsla(hue, saturation, lightness, alpha)`, `hsv(hue, saturation, value)` and `hsva(hue, saturation, value, alpha)` where the components are given as hexadecimal numbers in the first four cases and as decimals or percentages (0%-100%) in the remaining cases. Red, green and blue components are between 0 and 255; hue is between 0 and 360; saturation, lightness and value is between 0 and 100; alpha is between 0 and 1.
- **Valid HTML color names**, i.e. those that are present in the HTML 4.0 specification
- **Valid X11 color names**, see *http://en.wikipedia.org/wiki/X11_color_names*
- **Red-green-blue components** given separately in either a comma-, slash- or whitespace-separated string or a list or a tuple, in the range of 0-255. An alpha value of 255 (maximal opacity) will be assumed.
- **Red-green-blue-alpha components** given separately in either a comma-, slash- or whitespace-separated string or a list or a tuple, in the

hsla_to_rgba(*h*, *s*, *l*, *alpha*=`1.0`)

Converts a color given by its HSLA coordinates (hue, saturation, lightness, alpha) to RGBA coordinates.

Each of the HSLA coordinates must be in the range [0, 1].

hsl_to_rgb(*h*, *s*, *l*)

Converts a color given by its HSL coordinates (hue, saturation, lightness) to RGB coordinates.

Each of the HSL coordinates must be in the range [0, 1].

hsva_to_rgba(*h*, *s*, *v*, *alpha*=`1.0`)

Converts a color given by its HSVA coordinates (hue, saturation, value, alpha) to RGB coordinates.

Each of the HSVA coordinates must be in the range [0, 1].

hsv_to_rgb(*h*, *s*, *v*)

Converts a color given by its HSV coordinates (hue, saturation, value) to RGB coordinates.

Each of the HSV coordinates must be in the range [0, 1].

rgba_to_hsla(*r*, *g*, *b*, *alpha*=`1.0`)

Converts a color given by its RGBA coordinates to HSLA coordinates (hue, saturation, lightness, alpha).

Each of the RGBA coordinates must be in the range [0, 1].

rgba_to_hsva(*r*, *g*, *b*, *alpha*=`1.0`)

Converts a color given by its RGBA coordinates to HSVA coordinates (hue, saturation, value, alpha).

Each of the RGBA coordinates must be in the range [0, 1].

rgb_to_hsl(*r*, *g*, *b*)

Converts a color given by its RGB coordinates to HSL coordinates (hue, saturation, lightness).

Each of the RGB coordinates must be in the range [0, 1].

rgb_to_hsv(*r*, *g*, *b*)

Converts a color given by its RGB coordinates to HSV coordinates (hue, saturation, value).

Each of the RGB coordinates must be in the range [0, 1].

11.2 Variables

Name	Description
known_colors	**Value:** `{'alice blue': (0.941176470588, 0.972549019608, 1.0, 1.0)...`
palettes	**Value:** `{'gray': <GradientPalette with 256 colors>, 'heat': <Adva...`

11.3 Class Palette

object
└
igraph.drawing.colors.Palette

Known Subclasses: igraph.drawing.colors.AdvancedGradientPalette, igraph.drawing.colors.Precalculate igraph.drawing.colors.GradientPalette, igraph.drawing.colors.RainbowPalette

Base class of color palettes.

Color palettes are mappings that assign integers from the range 0..*n-1* to colors (4-tuples). *n* is called the size or length of the palette. `igraph` comes with a number of predefined palettes, so this class is useful for you only if you want to define your own palette. This can be done by subclassing this class and implementing the `Palette._get` method as necessary.

Palettes can also be used as lists or dicts, for the `__getitem__` method is overridden properly to call `Palette.get`.

11.3.1 Methods

__init__(*self*, *n*)

x.__init__(...) initializes x; see help(type(x)) for signature

Overrides: object.__init__ extit(inherited documentation)

clear_cache(*self*)

Clears the result cache.

The return values of `Palette.get` are cached. Use this method to clear the cache.

get(*self*, *v*)

Returns the given color from the palette.

Values are cached: if the specific value given has already been looked up, its value will be returned from the cache instead of calculating it again. Use `Palette.clear_cache` to clear the cache if necessary.

Parameters

`v`: the color to be retrieved. If it is an integer, it is passed to `Palette._get` to be translated to an RGBA quadruplet. Otherwise it is passed to `color_name_to_rgb()` to determine the RGBA values.

Return Value

the color as an RGBA quadruplet

Note: you shouldn't override this method in subclasses, override `_get` instead. If you override this method, lookups in the `known_colors` dict won't work, so you won't be able to refer to colors by names or RGBA quadruplets, only by integer indices. The caching functionality will disappear as well. However, feel free to override this method if this is exactly the behaviour you want.

get_many(*self*, *colors*)

Returns multiple colors from the palette.

Values are cached: if the specific value given has already been looked upon, its value will be returned from the cache instead of calculating it again. Use `Palette.clear_cache` to clear the cache if necessary.

Parameters

`colors`: the list of colors to be retrieved. The palette class tries to make an educated guess here: if it is not possible to interpret the value you passed here as a list of colors, the class will simply try to interpret it as a single color by forwarding the value to `Palette.get`.

Return Value

the colors as a list of RGBA quadruplets. The result will be a list even if you passed a single color index or color name.

__getitem__(*self*, *v*)

Returns the given color from the palette.

Values are cached: if the specific value given has already been looked up, its value will be returned from the cache instead of calculating it again. Use `Palette.clear_cache` to clear the cache if necessary.

Parameters

`v`: the color to be retrieved. If it is an integer, it is passed to `Palette._get` to be translated to an RGBA quadruplet. Otherwise it is passed to `color_name_to_rgb()` to determine the RGBA values.

Return Value

the color as an RGBA quadruplet

Note: you shouldn't override this method in subclasses, override `_get` instead. If you override this method, lookups in the `known_colors` dict won't work, so you won't be able to refer to colors by names or RGBA quadruplets, only by integer indices. The caching functionality will disappear as well. However, feel free to override this method if this is exactly the behaviour you want.

__len__(*self*)

Returns the number of colors in this palette

__plot__(*self*, *context*, *bbox*, *palette*, **args*, ***kwds*)

Plots the colors of the palette on the given Cairo context

Supported keyword arguments are:

- `border_width`: line width of the border shown around the palette. If zero or negative, the border is turned off. Default is `1`.
- `grid_width`: line width of the grid that separates palette cells. If zero or negative, the grid is turned off. The grid is also turned off if the size of a cell is less than three times the given line width. Default is `0`. Fractional widths are also allowed.
- `orientation`: the orientation of the palette. Must be one of the following values: `left-right`, `bottom-top`, `right-left` or `top-bottom`. Possible aliases: `horizontal` = `left-right`, `vertical` = `bottom-top`, `lr` = `left-right`, `rl` = `right-left`, `tb` = `top-bottom`, `bt` = `bottom-top`. The default is `left-right`.

__repr__(*self*)

repr(x)

Overrides: object.__repr__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

11.3.2 Properties

Name	Description
length	Returns the number of colors in this palette
Inherited from object	
__class__	

11.4 Class GradientPalette

Base class for gradient palettes

Gradient palettes contain a gradient between two given colors.

Example:

```
>>> pal = GradientPalette("red", "blue", 5)
>>> pal.get(0)
(1.0, 0.0, 0.0, 1.0)
>>> pal.get(2)
(0.5, 0.0, 0.5, 1.0)
>>> pal.get(4)
(0.0, 0.0, 1.0, 1.0)
```

11.4.1 Methods

__init__(*self*, *color1*, *color2*, *n*=256)

Creates a gradient palette.

Parameters

`color1`: the color where the gradient starts.

`color2`: the color where the gradient ends.

`n`: the number of colors in the palette.

Overrides: object.__init__

Inherited from igraph.drawing.colors.Palette(Section 11.3)

__getitem__(), __len__(), __plot__(), __repr__(), clear_cache(), get(), get_many()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

11.4.2 Properties

Name	Description
Inherited from igraph.drawing.colors.Palette (Section 11.3)	
length	
Inherited from object	
__class__	

11.5 Class AdvancedGradientPalette

object

igraph.drawing.colors.Palette

igraph.drawing.colors.AdvancedGradientPalette

Advanced gradient that consists of more than two base colors.

Example:

```
>>> pal = AdvancedGradientPalette(["red", "black", "blue"], n=9)
>>> pal.get(2)
(0.5, 0.0, 0.0, 1.0)
>>> pal.get(7)
```

```
(0.0, 0.0, 0.75, 1.0)
```

11.5.1 Methods

__init__(*self*, *colors*, *indices*=`None`, *n*=`256`)

Creates an advanced gradient palette

Parameters

`colors`: the colors in the gradient.

`indices`: the color indices belonging to the given colors. If `None`, the colors are distributed equidistantly

`n`: the total number of colors in the palette

Overrides: object.__init__

Inherited from igraph.drawing.colors.Palette(Section 11.3)

__getitem__(), __len__(), __plot__(), __repr__(), clear_cache(), get(), get_many()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

11.5.2 Properties

Name	Description
Inherited from igraph.drawing.colors.Palette (Section 11.3)	
length	
Inherited from object	
__class__	

11.6 Class RainbowPalette

object

igraph.drawing.colors.Palette

igraph.drawing.colors.RainbowPalette

A palette that varies the hue of the colors along a scale.

Colors in a rainbow palette all have the same saturation, value and alpha components, while the hue is varied between two given extremes linearly. This palette has the advantage that it wraps around nicely if the hue is varied between zero and one (which is the default).

Example:

```
>>> pal = RainbowPalette(n=120)
>>> pal.get(0)
(1.0, 0.0, 0.0, 1.0)
>>> pal.get(20)
(1.0, 1.0, 0.0, 1.0)
>>> pal.get(40)
(0.0, 1.0, 0.0, 1.0)
>>> pal = RainbowPalette(n=120, s=1, v=0.5, alpha=0.75)
>>> pal.get(60)
(0.0, 0.5, 0.5, 0.75)
>>> pal.get(80)
(0.0, 0.0, 0.5, 0.75)
>>> pal.get(100)
(0.5, 0.0, 0.5, 0.75)
>>> pal = RainbowPalette(n=120)
>>> pal2 = RainbowPalette(n=120, start=0.5, end=0.5)
>>> pal.get(60) == pal2.get(0)
True
>>> pal.get(90) == pal2.get(30)
True
```

This palette was modeled after the `rainbow` command of R.

11.6.1 Methods

__init__(*self*, *n*=256, *s*=1, *v*=1, *start*=0, *end*=1, *alpha*=1)

Creates a rainbow palette.

Parameters

`n`: the number of colors in the palette.

`s`: the saturation of the colors in the palette.

`v`: the value component of the colors in the palette.

`start`: the hue at which the rainbow begins (between 0 and 1).

`end`: the hue at which the rainbow ends (between 0 and 1).

`alpha`: the alpha component of the colors in the palette.

Overrides: object.__init__

Inherited from igraph.drawing.colors.Palette(Section 11.3)

__getitem__(), __len__(), __plot__(), __repr__(), clear_cache(), get(), get_many()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

11.6.2 Properties

Name	Description
Inherited from igraph.drawing.colors.Palette (Section 11.3)	
length	
Inherited from object	
__class__	

11.7 Class PrecalculatedPalette

object

igraph.drawing.colors.Palette

igraph.drawing.colors.PrecalculatedPalette

Known Subclasses: igraph.drawing.colors.ClusterColoringPalette

A palette that returns colors from a pre-calculated list of colors

11.7.1 Methods

__init__(*self*, *l*)

Creates the palette backed by the given list. The list must contain RGBA quadruplets or color names, which will be resolved first by `color_name_to_rgba()`. Anything that is understood by `color_name_to_rgba()` is OK here.

Overrides: object.__init__

Inherited from igraph.drawing.colors.Palette(Section 11.3)

__getitem__(), __len__(), __plot__(), __repr__(), clear_cache(), get(), get_many()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

11.7.2 Properties

Name	Description
Inherited from igraph.drawing.colors.Palette (Section 11.3)	
length	
Inherited from object	
__class__	

11.8 Class ClusterColoringPalette

object
igraph.drawing.colors.Palette
igraph.drawing.colors.PrecalculatedPalette
igraph.drawing.colors.ClusterColoringPalette

A palette suitable for coloring vertices when plotting a clustering.

This palette tries to make sure that the colors are easily distinguishable. This is achieved by using a set of base colors and their lighter and darker variants, depending on the number of elements in the palette.

When the desired size of the palette is less than or equal to the number of base colors (denoted by *n*), only the bsae colors will be used. When the size of the palette is larger than *n* but less than *2*n*, the base colors and their lighter variants will be used. Between *2*n* and *3*n*, the base colors and their lighter and darker variants will be used. Above *3*n*, more darker and lighter variants will be generated, but this makes the individual colors less and less distinguishable.

11.8.1 Methods

__init__(*self*, *n*)

Creates the palette backed by the given list. The list must contain RGBA quadruplets or color names, which will be resolved first by `color_name_to_rgba()`. Anything that is understood by `color_name_to_rgba()` is OK here.

Overrides: object.__init__ extit(inherited documentation)

Inherited from igraph.drawing.colors.Palette(Section 11.3)

__getitem__(), __len__(), __plot__(), __repr__(), clear_cache(), get(), get_many()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

11.8.2 Properties

Name	Description
Inherited from igraph.drawing.colors.Palette (Section 11.3)	
length	
Inherited from object	
__class__	

12 Module igraph.drawing.coord

Coordinate systems and related plotting routines

License: GPL

12.1 Variables

Name	Description
__package__	**Value:** `'igraph.drawing'`

12.2 Class CoordinateSystem

object
└
igraph.drawing.baseclasses.AbstractDrawer
└
igraph.drawing.baseclasses.AbstractCairoDrawer
└
igraph.drawing.coord.CoordinateSystem

Known Subclasses: igraph.drawing.coord.DescartesCoordinateSystem

Class implementing a coordinate system object.

Coordinate system objects are used when drawing plots which 2D or 3D coordinate system axes. This is an abstract class which must be extended in order to use it. In general, you'll only need the documentation of this class if you intend to implement an own coordinate system not present in igraph yet.

12.2.1 Methods

__init__(*self*, *context*, *bbox*)

Initializes the coordinate system.

Parameters

`context:` the context on which the coordinate system will be drawn.

`bbox:` the bounding box that will contain the coordinate system.

Overrides: object.__init__

draw(*self*)

Draws the coordinate system.

This method must be overridden in derived classes.

Overrides: igraph.drawing.baseclasses.AbstractDrawer.draw

local_to_context(*self*, *x*, *y*)

Converts local coordinates to the context coordinate system (given by the bounding box).

This method must be overridden in derived classes.

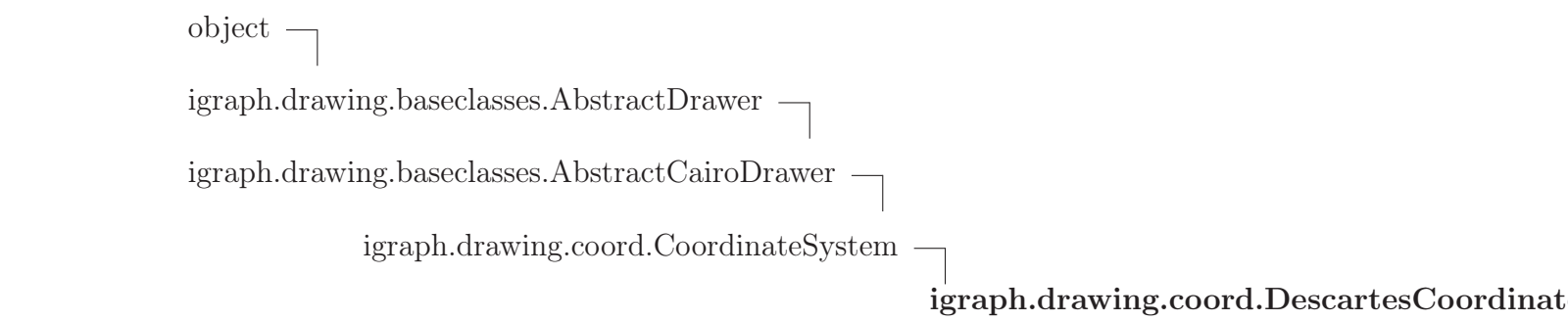
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

12.2.2 Properties

Name	Description
Inherited from igraph.drawing.baseclasses.AbstractCairoDrawer (Section 10.3)	
bbox	
Inherited from object	
__class__	

12.3 Class DescartesCoordinateSystem



Class implementing a 2D Descartes coordinate system object.

12.3.1 Methods

__init__(*self*, *context*, *bbox*, *bounds*)

Initializes the coordinate system.

Parameters

`context:` the context on which the coordinate system will be drawn.

`bbox:` the bounding box that will contain the coordinate system.

`bounds:` minimum and maximum X and Y values in a 4-tuple.

Overrides: object.__init__

draw(*self*)

Draws the coordinate system.

Overrides: igraph.drawing.baseclasses.AbstractDrawer.draw

local_to_context(*self*, *x*, *y*)

Converts local coordinates to the context coordinate system (given by the bounding box).

Overrides: igraph.drawing.coord.CoordinateSystem.local_to_context

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

12.3.2 Properties

Name	Description
bbox	Returns the bounding box of the coordinate system
bounds	Returns the lower and upper bounds of the X and Y values
Inherited from object __class__	

13 Module igraph.drawing.edge

Drawers for various edge styles in graph plots.

License: GPL

13.1 Class AbstractEdgeDrawer

object
└ **igraph.drawing.edge.AbstractEdgeDrawer**

Known Subclasses: igraph.drawing.edge.ArrowEdgeDrawer, igraph.drawing.edge.AlphaVaryingEdgeDra
igraph.drawing.edge.TaperedEdgeDrawer

Abstract edge drawer object from which all concrete edge drawer implementations are derived.

13.1.1 Methods

__init__(*self*, *context*, *palette*)

Constructs the edge drawer.

Parameters

`context`: a Cairo context on which the edges will be drawn.

`palette`: the palette that can be used to map integer color indices to colors when drawing edges

Overrides: object.__init__

draw_directed_edge(*self*, *edge*, *src_vertex*, *dest_vertex*)

Draws a directed edge.

Parameters

`edge`: the edge to be drawn. Visual properties of the edge are defined by the attributes of this object.

`src_vertex`: the source vertex. Visual properties are given again as attributes.

`dest_vertex`: the target vertex. Visual properties are given again as attributes.

draw_loop_edge(*self*, *edge*, *vertex*)

Draws a loop edge.

The default implementation draws a small circle.

Parameters

`edge:` the edge to be drawn. Visual properties of the edge are defined by the attributes of this object.

`vertex:` the vertex to which the edge is attached. Visual properties are given again as attributes.

draw_undirected_edge(*self*, *edge*, *src_vertex*, *dest_vertex*)

Draws an undirected edge.

The default implementation of this method draws undirected edges as straight lines. Loop edges are drawn as small circles.

Parameters

`edge:` the edge to be drawn. Visual properties of the edge are defined by the attributes of this object.

`src_vertex:` the source vertex. Visual properties are given again as attributes.

`dest_vertex:` the target vertex. Visual properties are given again as attributes.

get_label_position(*self*, *edge*, *src_vertex*, *dest_vertex*)

Returns the position where the label of an edge should be drawn. The default implementation returns the midpoint of the edge and an alignment that tries to avoid overlapping the label with the edge.

Parameters

`edge`: the edge to be drawn. Visual properties of the edge are defined by the attributes of this object.

`src_vertex`: the source vertex. Visual properties are given again as attributes.

`dest_vertex`: the target vertex. Visual properties are given again as attributes.

Return Value

a tuple containing two more tuples: the desired position of the label and the desired alignment of the label, where the position is given as `(x, y)` and the alignment is given as `(horizontal, vertical)`. Members of the alignment tuple are taken from constants in the `TextAlignment` class.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

13.1.2 Properties

Name	Description
Inherited from object	
__class__	

13.2 Class ArrowEdgeDrawer

object

igraph.drawing.edge.AbstractEdgeDrawer

igraph.drawing.edge.ArrowEdgeDrawer

Edge drawer implementation that draws undirected edges as straight lines and directed edges as arrows.

13.2.1 Methods

draw_directed_edge(*self*, *edge*, *src_vertex*, *dest_vertex*)

Draws a directed edge.

Parameters

`edge`: the edge to be drawn. Visual properties of the edge are defined by the attributes of this object.

`src_vertex`: the source vertex. Visual properties are given again as attributes.

`dest_vertex`: the target vertex. Visual properties are given again as attributes.

Overrides: igraph.drawing.edge.AbstractEdgeDrawer.draw_directed_edge extit(inherited documentation)

Inherited from igraph.drawing.edge.AbstractEdgeDrawer(Section 13.1)

__init__(), draw_loop_edge(), draw_undirected_edge(), get_label_position()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

13.2.2 Properties

Name	Description
Inherited from object	
__class__	

13.3 Class TaperedEdgeDrawer

object

igraph.drawing.edge.AbstractEdgeDrawer

igraph.drawing.edge.TaperedEdgeDrawer

Edge drawer implementation that draws undirected edges as straight lines and directed edges as tapered lines that are wider at the source and narrow at the destination.

13.3.1 Methods

draw_directed_edge(*self*, *edge*, *src_vertex*, *dest_vertex*)

Draws a directed edge.

Parameters

`edge`: the edge to be drawn. Visual properties of the edge are defined by the attributes of this object.

`src_vertex`: the source vertex. Visual properties are given again as attributes.

`dest_vertex`: the target vertex. Visual properties are given again as attributes.

Overrides: igraph.drawing.edge.AbstractEdgeDrawer.draw_directed_edge extit(inherited documentation)

Inherited from igraph.drawing.edge.AbstractEdgeDrawer(Section 13.1)

__init__(), draw_loop_edge(), draw_undirected_edge(), get_label_position()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

13.3.2 Properties

Name	Description
Inherited from object	
__class__	

13.4 Class AlphaVaryingEdgeDrawer

object

igraph.drawing.edge.AbstractEdgeDrawer

igraph.drawing.edge.AlphaVaryingEdgeDrawer

Known Subclasses: igraph.drawing.edge.DarkToLightEdgeDrawer, igraph.drawing.edge.LightToDarkEd

Edge drawer implementation that draws undirected edges as straight lines and directed edges by varying the alpha value of the specified edge color between the source and the destination.

13.4.1 Methods

__init__(*self*, *context*, *alpha_at_src*, *alpha_at_dest*)

Constructs the edge drawer.

Parameters

`context`: a Cairo context on which the edges will be drawn.

`palette`: the palette that can be used to map integer color indices to colors when drawing edges

Overrides: object.__init__ extit(inherited documentation)

draw_directed_edge(*self*, *edge*, *src_vertex*, *dest_vertex*)

Draws a directed edge.

Parameters

`edge`: the edge to be drawn. Visual properties of the edge are defined by the attributes of this object.

`src_vertex`: the source vertex. Visual properties are given again as attributes.

`dest_vertex`: the target vertex. Visual properties are given again as attributes.

Overrides: igraph.drawing.edge.AbstractEdgeDrawer.draw_directed_edge extit(inherited documentation)

Inherited from igraph.drawing.edge.AbstractEdgeDrawer(Section 13.1)

draw_loop_edge(), draw_undirected_edge(), get_label_position()

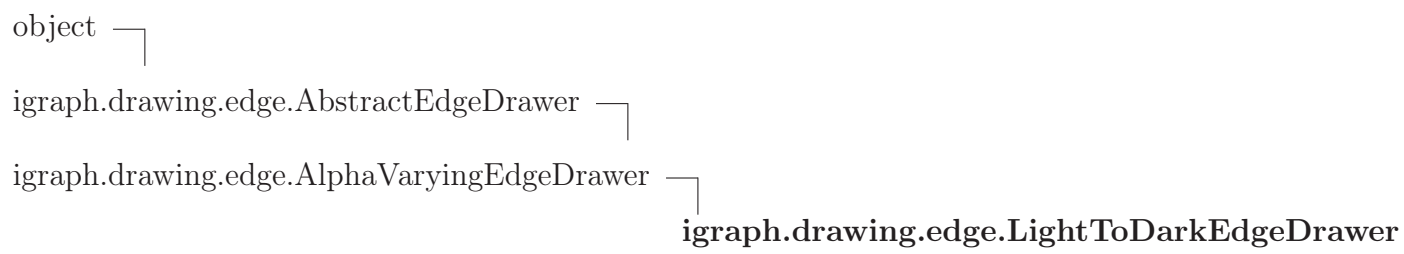
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

13.4.2 Properties

Name	Description
Inherited from object	
__class__	

13.5 Class LightToDarkEdgeDrawer



Edge drawer implementation that draws undirected edges as straight lines and directed edges by using an alpha value of zero (total transparency) at the source and an alpha value of one (full opacity) at the destination. The alpha value is interpolated in-between.

13.5.1 Methods

__init__(*self*, *context*)

Constructs the edge drawer.

Parameters

`context`: a Cairo context on which the edges will be drawn.

`palette`: the palette that can be used to map integer color indices to colors when drawing edges

Overrides: object.__init__ extit(inherited documentation)

Inherited from igraph.drawing.edge.AlphaVaryingEdgeDrawer(Section 13.4)

draw_directed_edge()

Inherited from igraph.drawing.edge.AbstractEdgeDrawer(Section 13.1)

draw_loop_edge(), draw_undirected_edge(), get_label_position()

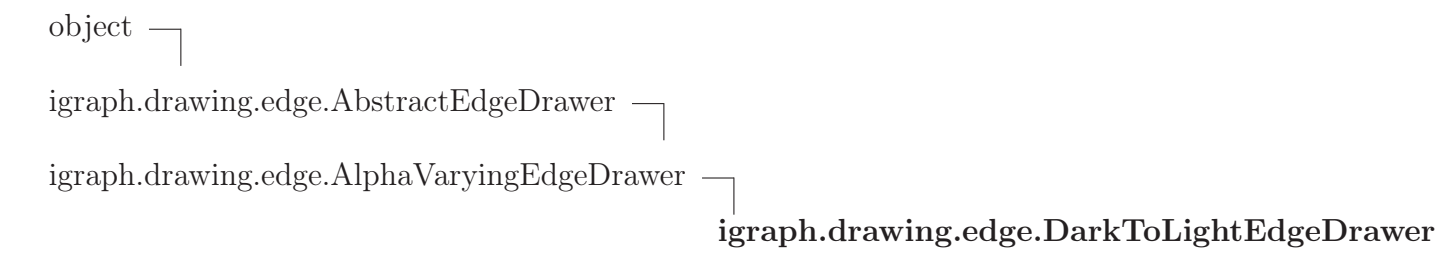
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

13.5.2 Properties

Name	Description
Inherited from object	
__class__	

13.6 Class DarkToLightEdgeDrawer



Edge drawer implementation that draws undirected edges as straight lines and directed edges by using an alpha value of one (full opacity) at the source and an alpha value of zero (total transparency) at the destination. The alpha value is interpolated in-between.

13.6.1 Methods

__init__(*self*, *context*)

Constructs the edge drawer.

Parameters

`context`: a Cairo context on which the edges will be drawn.

`palette`: the palette that can be used to map integer color indices to colors when drawing edges

Overrides: object.__init__ extit(inherited documentation)

Inherited from igraph.drawing.edge.AlphaVaryingEdgeDrawer(Section 13.4)

draw_directed_edge()

Inherited from igraph.drawing.edge.AbstractEdgeDrawer(Section 13.1)

draw_loop_edge(), draw_undirected_edge(), get_label_position()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

13.6.2 Properties

Name	Description
Inherited from object	
__class__	

14 Module igraph.drawing.graph

Drawing routines to draw graphs.

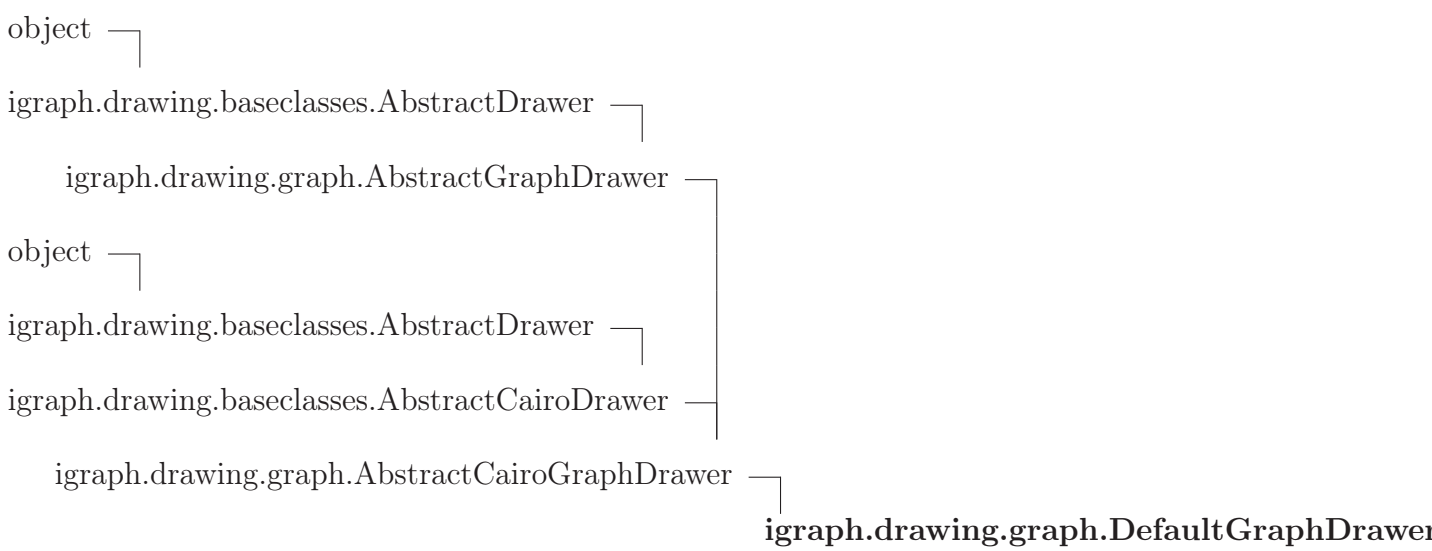
This module contains routines to draw graphs on:

- Cairo surfaces (`DefaultGraphDrawer`)
- UbiGraph displays (`UbiGraphDrawer`, see *http://ubietylab.net/ubigraph*)

It also contains routines to send an igraph graph directly to (Cytoscape[7]) using the (CytoscapeRPC plugin[8]), see `CytoscapeGraphDrawer`. `CytoscapeGraphDrawer` can also fetch the current network from Cytoscape and convert it to igraph format.

License: GPL

14.1 Class DefaultGraphDrawer



Class implementing the default visualisation of a graph.

The default visualisation of a graph draws the nodes on a 2D plane according to a given `Layout`, then draws a straight or curved edge between nodes connected by edges. This is the visualisation used when one invokes the `plot()` function on a `Graph` object.

See `Graph.__plot__()` for the keyword arguments understood by this drawer.

[7] http://www.cytoscape.org
[8] http://gforge.nbic.nl/projects/cytoscaperpc/

14.1.1 Methods

__init__(*self*, *context*, *bbox*, *vertex_drawer_factory*=<class 'igraph.drawing.vertex.DefaultVertexDrawer'>, *edge_drawer_factory*=<class 'igraph.drawing.edge.ArrowEdgeDrawer'>, *label_drawer_factory*=<class 'igraph.drawing.text.TextDrawer'>)

Constructs the graph drawer and associates it to the given Cairo context and the given `BoundingBox`.

Parameters

`context:`	the context on which we will draw
`bbox:`	the bounding box within which we will draw. Can be anything accepted by the constructor of `BoundingBox` (i.e., a 2-tuple, a 4-tuple or a `BoundingBox` object).
`vertex_drawer_factory:`	a factory method that returns an `AbstractCairoVertexDrawer` instance bound to a given Cairo context. The factory method must take three parameters: the Cairo context, the bounding box of the drawing area and the palette to be used for drawing colored vertices. The default vertex drawer is `DefaultVertexDrawer`.
`edge_drawer_factory:`	a factory method that returns an `AbstractEdgeDrawer` instance bound to a given Cairo context. The factory method must take two parameters: the Cairo context and the palette to be used for drawing colored edges. You can use any of the actual `AbstractEdgeDrawer` implementations here to control the style of edges drawn by igraph. The default edge drawer is `ArrowEdgeDrawer`.
`label_drawer_factory:`	a factory method that returns a `TextDrawer` instance bound to a given Cairo context. The method must take one parameter: the Cairo context. The default label drawer is `TextDrawer`.

Overrides: object.__init__

draw(*self*, *graph*, *palette*, **args*, ***kwds*)

Abstract method, must be implemented in derived classes.

Overrides: igraph.drawing.baseclasses.AbstractDrawer.draw extit(inherited documentation)

Inherited from igraph.drawing.graph.AbstractGraphDrawer

ensure_layout()

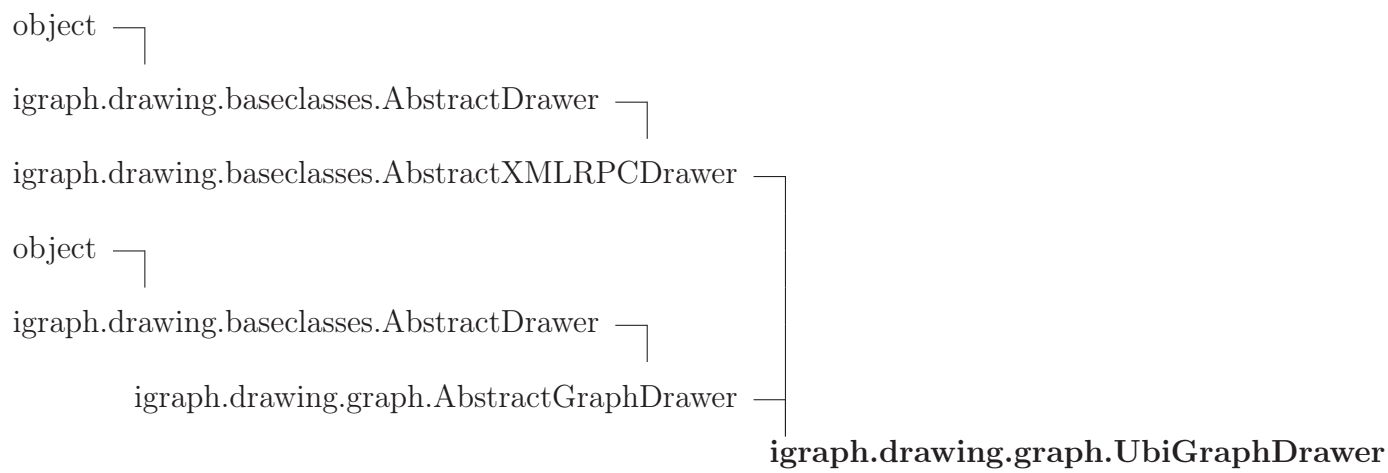
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

14.1.2 Properties

Name	Description
Inherited from igraph.drawing.baseclasses.AbstractCairoDrawer (Section 10.3)	
bbox	
Inherited from object	
__class__	

14.2 Class UbiGraphDrawer



Graph drawer that draws a given graph on an UbiGraph display using the XML-RPC API of UbiGraph.

The following vertex attributes are supported: `color`, `label`, `shape`, `size`. See the Ubigraph documentation for supported shape names. Sizes are relative to the default Ubigraph size.

The following edge attributes are supported: `color`, `label`, `width`. Edge widths are relative

to the default Ubigraph width.

All color specifications supported by igraph (e.g., color names, palette indices, RGB triplets, RGBA quadruplets, HTML format) are understood by the Ubigraph graph drawer.

The drawer also has two attributes, `vertex_defaults` and `edge_defaults`. These are dictionaries that can be used to set default values for the vertex/edge attributes in Ubigraph.

14.2.1 Methods

__init__(*self*, *url*=`'http://localhost:20738/RPC2'`)

Constructs an UbiGraph drawer using the display at the given URL.

Parameters

`url`: the URL where the XML-RPC calls for the service should be addressed to.

`service`: the name of the service at the XML-RPC address. If `None`, requests will be directed to the server proxy object constructed by `xmlrpclib.ServerProxy`; if not `None`, the given attribute will be looked up in the server proxy object.

Overrides: object.__init__

draw(*self*, *graph*, **args*, ***kwds*)

Draws the given graph on an UbiGraph display.

Parameters

`clear`: whether to clear the current UbiGraph display before plotting. Default: `True`.

Overrides: igraph.drawing.baseclasses.AbstractDrawer.draw

Inherited from igraph.drawing.graph.AbstractGraphDrawer

ensure_layout()

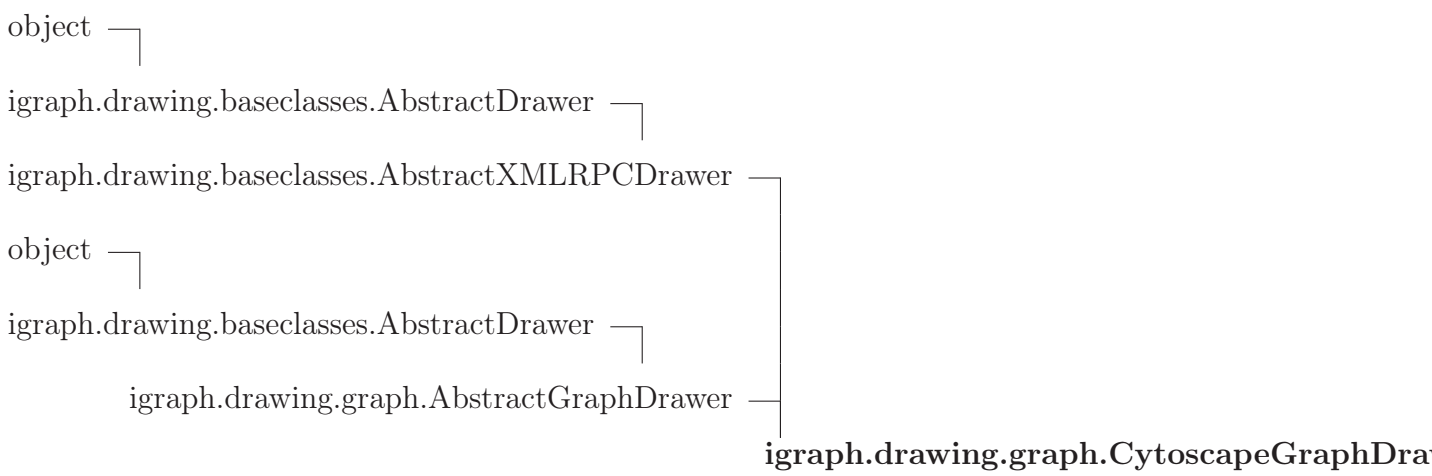
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

14.2.2 Properties

Name	Description
Inherited from object	
__class__	

14.3 Class CytoscapeGraphDrawer



Graph drawer that sends/receives graphs to/from Cytoscape using CytoscapeRPC.

This graph drawer cooperates with Cytoscape[9] using CytoscapeRPC[10]. You need to install the CytoscapeRPC plugin first and start the XML-RPC server on a given port (port 9000 by default) from the appropriate Plugins submenu in Cytoscape.

Graph, vertex and edge attributes are transferred to Cytoscape whenever possible (i.e. when a suitable mapping exists between a Python type and a Cytoscape type). If there is no suitable Cytoscape type for a Python type, the drawer will use a string attribute on the Cytoscape side and invoke `str()` on the Python attributes.

If an attribute to be created on the Cytoscape side already exists with a different type, an underscore will be appended to the attribute name to resolve the type conflict.

You can use the `network_id` attribute of this class to figure out the network ID of the last graph drawn with this drawer.

[9] http://www.cytoscape.org
[10] http://wiki.nbic.nl/index.php/CytoscapeRPC

14.3.1 Methods

__init__(*self*, *url*='http://localhost:9000/Cytoscape')

Constructs a Cytoscape graph drawer using the XML-RPC interface of Cytoscape at the given URL.

Parameters

`url`: the URL where the XML-RPC calls for the service should be addressed to.

`service`: the name of the service at the XML-RPC address. If `None`, requests will be directed to the server proxy object constructed by `xmlrpclib.ServerProxy`; if not `None`, the given attribute will be looked up in the server proxy object.

Overrides: object.__init__

draw(*self*, *graph*, *name*='Network from igraph', *create_view*=True, **args*, ***kwds*)

Sends the given graph to Cytoscape as a new network.

Parameters

`name`: the name of the network in Cytoscape.

`create_view`: whether to create a view for the network in Cytoscape.The default is `True`.

`node_ids`: specifies the identifiers of the nodes to be used in Cytoscape. This must either be the name of a vertex attribute or a list specifying the identifiers, one for each node in the graph. The default is `None`, which simply uses the vertex index for each vertex.

Overrides: igraph.drawing.baseclasses.AbstractDrawer.draw

fetch(*self*, *name*=`None`, *directed*=`False`, *keep_canonical_names*=`False`)

Fetches the network with the given name from Cytoscape.

When fetching networks from Cytoscape, the `canonicalName` attributes of vertices and edges are not converted by default. Use the `keep_canonical_names` parameter to retrieve these attributes as well.

Parameters

`name:`	the name of the network in Cytoscape.
`directed:`	whether the network is directed.
`keep_canonical_names:`	whether to keep the `canonicalName` vertex/edge attributes that are added automatically by Cytoscape

Return Value

an appropriately constructed igraph `Graph`.

infer_cytoscape_type(*values*)

Returns a Cytoscape type that can be used to represent all the values in ‘values‘ and an appropriately converted copy of ‘values‘ that is suitable for an XML-RPC call. Note that the string type in Cytoscape is used as a catch-all type; if no other type fits, attribute values will be converted to string and then posted to Cytoscape.

“None“ entries are allowed in ‘values‘, they will be ignored on the Cytoscape side.

Inherited from igraph.drawing.graph.AbstractGraphDrawer

ensure_layout()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

14.3.2 Properties

Name	Description
Inherited from object	
__class__	

15 Module igraph.drawing.metamagic

Auxiliary classes for the default graph drawer in igraph.

This module contains heavy metaclass magic. If you don't understand the logic behind these classes, probably you don't need them either.

igraph's default graph drawer uses various data sources to determine the visual appearance of vertices and edges. These data sources are the following (in order of precedence):

- The keyword arguments passed to the `igraph.plot()` function (or to `igraph.Graph.__plot__()` as a matter of fact, since `igraph.plot()` just passes these attributes on). For instance, a keyword argument named `vertex_label` can be used to set the labels of vertices.
- The attributes of the vertices/edges being drawn. For instance, a vertex that has a `label` attribute will use that label when drawn by the default graph drawer.
- The global configuration of igraph. For instance, if the global `igraph.config.Configuration` instance has a key called `plotting.vertex_color`, that will be used as a default color for the vertices.
- If all else fails, there is a built-in default; for instance, the default vertex color is `"red"`. This is hard-wired in the source code.

The logic above can be useful in other graph drawers as well, not only in the default one, therefore it is refactored into the classes found in this module. Different graph drawers may inspect different vertex or edge attributes, hence the classes that collect the attributes from the various data sources are generated in run-time using a metaclass called `AttributeCollectorMeta`. You don't have to use `AttributeCollectorMeta` directly, just implement a subclass of `AttributeCollectorBase` and it will ensure that the appropriate metaclass is used. With `AttributeCollectorBase`, you can use a simple declarative syntax to specify which attributes you are interested in. For example:

```
class VisualEdgeBuilder(AttributeCollectorBase):
    arrow_size = 1.0
    arrow_width = 1.0
    color = ("black", palette.get)
    width = 1.0

for edge in VisualEdgeBuilder(graph.es):
    print edge.color
```

The above class is a visual edge builder – a class that gives the visual attributes of the edges of a graph that is specified at construction time. It specifies that the attributes we are interested in are `arrow_size`, `arrow_width`, `color` and `width`; the default values are also given. For `color`, we also specify that a method called {palette.get} should be called on every attribute value to translate color names to RGB values. For the other three attributes, `float` will implicitly be called on all attribute values, this is inferred from the type of the default value itself.

See Also: AttributeCollectorMeta, AttributeCollectorBase

15.1 Class AttributeSpecification

object
└
igraph.drawing.metamagic.AttributeSpecification

Class that describes how the value of a given attribute should be retrieved.

The class contains the following members:

- `name`: the name of the attribute. This is also used when we are trying to get its value from a vertex/edge attribute of a graph.
- `alt_name`: alternative name of the attribute. This is used when we are trying to get its value from a Python dict or an `igraph.Configuration` object. If omitted at construction time, it will be equal to `name`.
- `default`: the default value of the attribute when none of the sources we try can provide a meaningful value.
- `transform`: optional transformation to be performed on the attribute value. If `None` or omitted, it defaults to the type of the default value.
- `func`: when given, this function will be called with an index in order to derive the value of the attribute.

15.1.1 Methods

__init__(*self*, *name*, *default*=`None`, *alt_name*=`None`, *transform*=`None`, *func*=`None`)

x.__init__(...) initializes x; see help(type(x)) for signature

Overrides: object.__init__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

15.1.2 Properties

Name	Description
accessor	
alt_name	
default	
func	
name	

continued on next page

Name	Description
transform	
Inherited from object	
__class__	

15.2 Class AttributeCollectorBase

object
└
igraph.drawing.metamagic.AttributeCollectorBase

Base class for attribute collector subclasses. Classes that inherit this class may use a declarative syntax to specify which vertex or edge attributes they intend to collect. See `AttributeCollectorMeta` for the details.

15.2.1 Methods

__init__(*self*, *seq*, *kwds*=`None`)

Constructs a new attribute collector that uses the given vertex/edge sequence and the given dict as data sources.

Parameters

`seq`: an `igraph.VertexSeq` or `igraph.EdgeSeq` class that will be used as a data source for attributes.

`kwds`: a Python dict that will be used to override the attributes collected from *seq* if necessary.

Overrides: object.__init__

__getitem__(*self*, *index*)

Returns the collected attributes of the vertex/edge with the given index.

__len__(*self*)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

15.2.2 Properties

Name	Description
Inherited from object	
__class__	

16 Module igraph.drawing.shapes

Shape drawing classes for igraph

Vertex shapes in igraph are usually referred to by short names like `"rect"` or `"circle"`. This module contains the classes that implement the actual drawing routines for these shapes, and a resolver class that determines the appropriate shape drawer class given the short name.

Classes that are derived from `ShapeDrawer` in this module are automatically registered by `ShapeDrawerDirectory`. If you implement a custom shape drawer, you must register it in `ShapeDrawerDirectory` manually if you wish to refer to it by a name in the `shape` attribute of vertices.

16.1 Class ShapeDrawerDirectory

object
└ **igraph.drawing.shapes.ShapeDrawerDirectory**

Static class that resolves shape names to their corresponding shape drawer classes.

Classes that are derived from `ShapeDrawer` in this module are automatically registered by `ShapeDrawerDirectory` when the module is loaded for the first time.

16.1.1 Methods

register(*cls*, *drawer_class*)

Registers the given shape drawer class under the given names.

Parameters

`drawer_class`: the shape drawer class to be registered

register_namespace(*cls*, *namespace*)

Registers all `ShapeDrawer` classes in the given namespace

Parameters

`namespace`: a Python dict mapping names to Python objects.

resolve(*cls*, *shape*)

Given a shape name, returns the corresponding shape drawer class

Parameters

`shape`: the name of the shape

Return Value

the corresponding shape drawer class

Raises

`ValueError` if the shape is unknown

resolve_default(*cls*, *shape*, *default*=`<class 'igraph.drawing.shapes.NullDrawer'>`)

Given a shape name, returns the corresponding shape drawer class or the given default shape drawer if the shape name is unknown.

Parameters

`shape`: the name of the shape

`default`: the default shape drawer to return when the shape is unknown

Return Value

the shape drawer class corresponding to the given name or the default shape drawer class if the name is unknown

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

16.1.2 Properties

Name	Description
Inherited from object	
__class__	

16.1.3 Class Variables

Name	Description
known_shapes	**Value:** `{'': <class 'igraph.drawing.shapes.NullDrawer'>, 'arrow':...`

17 Module igraph.drawing.text

Drawers for labels on plots.

@undocumented: test **License:** GPL

17.1 Class TextAlignment

object
┐
igraph.drawing.text.TextAlignment

Text alignment constants.

17.1.1 Methods

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

17.1.2 Properties

Name	Description
Inherited from object	
__class__	

17.1.3 Class Variables

Name	Description
BOTTOM	**Value:** `'bottom'`
CENTER	**Value:** `'center'`
LEFT	**Value:** `'left'`
RIGHT	**Value:** `'right'`
TOP	**Value:** `'top'`

17.2 Class TextDrawer

object
┐
igraph.drawing.baseclasses.AbstractDrawer
┐
igraph.drawing.baseclasses.AbstractCairoDrawer
┐
igraph.drawing.text.TextDrawer

Class that draws text on a Cairo context.

This class supports multi-line text unlike the original Cairo text drawing methods.

17.2.1 Methods

__init__(*self*, *context*, *text*=`''`, *halign*=`'center'`, *valign*=`'center'`)

Constructs a new instance that will draw the given `text` on the given Cairo `context`. **Parameters**

`context`: the context on which we will draw

`bbox`: the bounding box within which we will draw. Can be anything accepted by the constructor of `BoundingBox` (i.e., a 2-tuple, a 4-tuple or a `BoundingBox` object).

Overrides: object.__init__

draw(*self*, *wrap*=`False`)

Draws the text in the current bounding box of the drawer.

Since the class itself is an instance of `AbstractCairoDrawer`, it has an attribute named `bbox` which will be used as a bounding box. **Parameters**

`wrap`: whether to allow re-wrapping of the text if it does not fit within the bounding box horizontally. *(type=boolean)*

Overrides: igraph.drawing.baseclasses.AbstractDrawer.draw

get_text_layout(*self*, *x*=`None`, *y*=`None`, *width*=`None`, *wrap*=`False`)

Calculates the layout of the current text. `x` and `y` denote the coordinates where the drawing should start. If they are both `None`, the current position of the context will be used.

Vertical alignment settings are not taken into account in this method as the text is not drawn within a box. **Parameters**

`x`: The X coordinate of the reference point where the layout should start. *(type=float or `None`)*

`y`: The Y coordinate of the reference point where the layout should start. *(type=float or `None`)*

`width`: The width of the box in which the text will be fitted. It matters only when the text is right-aligned or centered. The text will overflow the box if any of the lines is longer than the box width and `wrap` is `False`. *(type=float or `None`)*

`wrap`: whether to allow re-wrapping of the text if it does not fit within the given width. *(type=boolean)*

Return Value

a list consisting of `(x, y, line)` tuples where `x` and `y` refer to reference points on the Cairo canvas and `line` refers to the corresponding text that should be plotted there.

draw_at(*self*, *x*=`None`, *y*=`None`, *width*=`None`, *wrap*=`False`)

Draws the text by setting up an appropriate path on the Cairo context and filling it. `x` and `y` denote the coordinates where the drawing should start. If they are both `None`, the current position of the context will be used.

Vertical alignment settings are not taken into account in this method as the text is not drawn within a box. **Parameters**

`x`: The X coordinate of the reference point where the drawing should start. *(type=float or `None`)*

`y`: The Y coordinate of the reference point where the drawing should start. *(type=float or `None`)*

`width`: The width of the box in which the text will be fitted. It matters only when the text is right-aligned or centered. The text will overflow the box if any of the lines is longer than the box width. *(type=float or `None`)*

`wrap`: whether to allow re-wrapping of the text if it does not fit within the given width. *(type=boolean)*

text_extents(*self*)

Returns the X-bearing, Y-bearing, width, height, X-advance and Y-advance of the text.

For multi-line text, the X-bearing and Y-bearing correspond to the first line, while the X-advance is extracted from the last line. and the Y-advance is the sum of all the Y-advances. The width and height correspond to the entire bounding box of the text.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

17.2.2 Properties

Name	Description
text	Returns the text to be drawn.
Inherited from igraph.drawing.baseclasses.AbstractCairoDrawer (Section 10.3)	
bbox	
Inherited from object	

continued on next page

Name	Description
__class__	

17.2.3 Class Variables

Name	Description
BOTTOM	**Value:** `'bottom'`
CENTER	**Value:** `'center'`
LEFT	**Value:** `'left'`
RIGHT	**Value:** `'right'`
TOP	**Value:** `'top'`

18 Module igraph.drawing.utils

Utility classes for drawing routines.

License: GPL

18.1 Class Rectangle

object
└ **igraph.drawing.utils.Rectangle**

Known Subclasses: igraph.drawing.utils.BoundingBox

Class representing a rectangle.

18.1.1 Methods

__init__(*self*, **args*)

Creates a rectangle.

The corners of the rectangle can be specified by either a tuple (four items, two for each corner, respectively), four separate numbers (X and Y coordinates for each corner) or two separate numbers (width and height, the upper left corner is assumed to be at (0,0))

Overrides: object.__init__

contract(*self*, *margins*)

Contracts the rectangle by the given margins.

Return Value
a new `Rectangle` object.

expand(*self*, *margins*)

Expands the rectangle by the given margins.

Return Value
a new `Rectangle` object.

isdisjoint(*self*, *other*)

Returns "True" if the two rectangles have no intersection.

Example:

```
>>> r1 = Rectangle(10, 10, 30, 30)
>>> r2 = Rectangle(20, 20, 50, 50)
>>> r3 = Rectangle(70, 70, 90, 90)
>>> r1.isdisjoint(r2)
False
>>> r2.isdisjoint(r1)
False
>>> r1.isdisjoint(r3)
True
>>> r3.isdisjoint(r1)
True
```

isempty(*self*)

Returns "True" if the rectangle is empty (i.e. it has zero width and height).

Example:

```
>>> r1 = Rectangle(10, 10, 30, 30)
>>> r2 = Rectangle(70, 70, 90, 90)
>>> r1.isempty()
False
>>> r2.isempty()
False
>>> r1.intersection(r2).isempty()
True
```

intersection(*self*, *other*)

Returns the intersection of this rectangle with another.

Example:

```
>>> r1 = Rectangle(10, 10, 30, 30)
>>> r2 = Rectangle(20, 20, 50, 50)
>>> r3 = Rectangle(70, 70, 90, 90)
>>> r1.intersection(r2)
Rectangle(20.0, 20.0, 30.0, 30.0)
>>> r2 & r1
Rectangle(20.0, 20.0, 30.0, 30.0)
>>> r2.intersection(r1) == r1.intersection(r2)
True
>>> r1.intersection(r3)
Rectangle(0.0, 0.0, 0.0, 0.0)
```

__and__(*self*, *other*)

Returns the intersection of this rectangle with another.

Example:

```
>>> r1 = Rectangle(10, 10, 30, 30)
>>> r2 = Rectangle(20, 20, 50, 50)
>>> r3 = Rectangle(70, 70, 90, 90)
>>> r1.intersection(r2)
Rectangle(20.0, 20.0, 30.0, 30.0)
>>> r2 & r1
Rectangle(20.0, 20.0, 30.0, 30.0)
>>> r2.intersection(r1) == r1.intersection(r2)
True
>>> r1.intersection(r3)
Rectangle(0.0, 0.0, 0.0, 0.0)
```

translate(*self*, *dx*, *dy*)

Translates the rectangle in-place.

Example:

```
>>> r = Rectangle(10, 20, 50, 70)
>>> r.translate(30, -10)
>>> r
Rectangle(40.0, 10.0, 80.0, 60.0)
```

Parameters

`dx`: the X coordinate of the translation vector

`dy`: the Y coordinate of the translation vector

union(*self*, *other*)

Returns the union of this rectangle with another.

The resulting rectangle is the smallest rectangle that contains both rectangles.

Example:

```
>>> r1 = Rectangle(10, 10, 30, 30)
>>> r2 = Rectangle(20, 20, 50, 50)
>>> r3 = Rectangle(70, 70, 90, 90)
>>> r1.union(r2)
Rectangle(10.0, 10.0, 50.0, 50.0)
>>> r2 | r1
Rectangle(10.0, 10.0, 50.0, 50.0)
>>> r2.union(r1) == r1.union(r2)
True
>>> r1.union(r3)
Rectangle(10.0, 10.0, 90.0, 90.0)
```

__or__(*self*, *other*)

Returns the union of this rectangle with another.

The resulting rectangle is the smallest rectangle that contains both rectangles.

Example:

```
>>> r1 = Rectangle(10, 10, 30, 30)
>>> r2 = Rectangle(20, 20, 50, 50)
>>> r3 = Rectangle(70, 70, 90, 90)
>>> r1.union(r2)
Rectangle(10.0, 10.0, 50.0, 50.0)
>>> r2 | r1
Rectangle(10.0, 10.0, 50.0, 50.0)
>>> r2.union(r1) == r1.union(r2)
True
>>> r1.union(r3)
Rectangle(10.0, 10.0, 90.0, 90.0)
```

__ior__(*self*, *other*)

Expands this rectangle to include itself and another completely while still being as small as possible.

Example:

```
>>> r1 = Rectangle(10, 10, 30, 30)
>>> r2 = Rectangle(20, 20, 50, 50)
>>> r3 = Rectangle(70, 70, 90, 90)
>>> r1 |= r2
>>> r1
Rectangle(10.0, 10.0, 50.0, 50.0)
>>> r1 |= r3
>>> r1
Rectangle(10.0, 10.0, 90.0, 90.0)
```

__repr__(*self*)

repr(x)

Overrides: object.__repr__ extit(inherited documentation)

__eq__(*self*, *other*)

__ne__(*self*, *other*)

__bool__(*self*)

__nonzero__(*self*)

__hash__(*self*)

hash(x)

Overrides: object.__hash__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

18.1.2 Properties

Name	Description
coords	The coordinates of the corners. The coordinates are returned as a 4-tuple in the following order: left edge, top edge, right edge, bottom edge.
width	The width of the rectangle
height	The height of the rectangle
left	The X coordinate of the left side of the box
right	The X coordinate of the right side of the box
top	The Y coordinate of the top edge of the box
bottom	The Y coordinate of the bottom edge of the box
midx	The X coordinate of the center of the box
midy	The Y coordinate of the center of the box
shape	The shape of the rectangle (width, height)
Inherited from object __class__	

18.2 Class BoundingBox

object

igraph.drawing.utils.Rectangle

igraph.drawing.utils.BoundingBox

Class representing a bounding box (a rectangular area) that encloses some objects.

18.2.1 Methods

__ior__(*self*, *other*)

Replaces this bounding box with the union of itself and another.

Example:

```
>>> box1 = BoundingBox(10, 20, 50, 60)
>>> box2 = BoundingBox(70, 40, 100, 90)
>>> box1 |= box2
>>> print(box1)
BoundingBox(10.0, 20.0, 100.0, 90.0)
```

Overrides: igraph.drawing.utils.Rectangle.__ior__

__or__(*self*, *other*)

Takes the union of this bounding box with another.

The result is a bounding box which encloses both bounding boxes.

Example:

```
>>> box1 = BoundingBox(10, 20, 50, 60)
>>> box2 = BoundingBox(70, 40, 100, 90)
>>> box1 | box2
BoundingBox(10.0, 20.0, 100.0, 90.0)
```

Overrides: igraph.drawing.utils.Rectangle.__or__

Inherited from igraph.drawing.utils.Rectangle(Section 18.1)

__and__(), __bool__(), __eq__(), __hash__(), __init__(), __ne__(), __nonzero__(), __repr__(), contract(), expand(), intersection(), isdisjoint(), isempty(), translate(), union()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

18.2.2 Properties

Name	Description
Inherited from igraph.drawing.utils.Rectangle (Section 18.1)	
bottom, coords, height, left, midx, midy, right, shape, top, width	
Inherited from object	
__class__	

18.3 Class FakeModule

object
└ **igraph.drawing.utils.FakeModule**

Fake module that raises an exception for everything

18.3.1 Methods

__getattr__(*self*, _)

__call__(*self*, _)

__setattr__(*self*, *key*, *value*)

x.__setattr__('name', value) <==> x.name = value

Overrides: object.__setattr__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __sizeof__(), __str__(), __subclasshook__()

18.3.2 Properties

Name	Description
Inherited from object	
__class__	

18.4 Class Point

object
└ tuple
└ **igraph.drawing.utils.Point**

Class representing a point on the 2D plane.

18.4.1 Methods

__new__(*cls*, *x*, *y*)

Creates a new point with the given coordinates

Return Value

a new object with type S, a subtype of T

Overrides: object.__new__

__repr__(*self*)

Returns a nicely formatted representation of the point

Overrides: object.__repr__

__getnewargs__(*self*)

Return self as a plain tuple. Used by copy and pickle.

Overrides: tuple.__getnewargs__

__add__(*self*, *other*)

Adds the coordinates of a point to another one

Overrides: tuple.__add__

__sub__(*self*, *other*)

Subtracts the coordinates of a point to another one

__mul__(*self*, *scalar*)

Multiplies the coordinates by a scalar

Overrides: tuple.__mul__

__rmul__(*self*, *scalar*)

Multiplies the coordinates by a scalar

Overrides: tuple.__rmul__

__div__(*self*, *scalar*)

Divides the coordinates by a scalar

as_polar(*self*)

Returns the polar coordinate representation of the point.

Return Value

the radius and the angle in a tuple.

distance(*self*, *other*)

Returns the distance of the point from another one.

Example:

```
>>> p1 = Point(5, 7)
>>> p2 = Point(8, 3)
>>> p1.distance(p2)
5.0
```

interpolate(*self*, *other*, *ratio*=0.5)

Linearly interpolates between the coordinates of this point and another one.

Parameters

`other`: the other point

`ratio`: the interpolation ratio between 0 and 1. Zero will return this point, 1 will return the other point.

length(*self*)

Returns the length of the vector pointing from the origin to this point.

normalized(*self*)

Normalizes the coordinates of the point s.t. its length will be 1 after normalization. Returns the normalized point.

sq_length(*self*)

Returns the squared length of the vector pointing from the origin to this point.

towards(*self*, *other*, *distance*=0)

Returns the point that is at a given distance from this point towards another one.

FromPolar(*cls*, *radius*, *angle*)

Constructs a point from polar coordinates.

'radius' is the distance of the point from the origin; 'angle' is the angle between the X axis and the vector pointing to the point from the origin.

Inherited from tuple

__contains__(), __eq__(), __ge__(), __getattribute__(), __getitem__(), __getslice__(), __gt__(), __hash__(), __iter__(), __le__(), __len__(), __lt__(), __ne__(), count(), index()

Inherited from object

__delattr__(), __format__(), __init__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

18.4.2 Properties

Name	Description
x	Alias for field number 0
y	Alias for field number 1
Inherited from object __class__	

19 Module igraph.drawing.vertex

Drawing routines to draw the vertices of graphs.

This module provides implementations of vertex drawers, i.e. drawers that the default graph drawer will use to draw vertices.

License: GPL

19.1 Class AbstractVertexDrawer

object
└
igraph.drawing.baseclasses.AbstractDrawer
└
igraph.drawing.vertex.AbstractVertexDrawer

Known Subclasses: igraph.drawing.vertex.AbstractCairoVertexDrawer

Abstract vertex drawer object from which all concrete vertex drawer implementations are derived.

19.1.1 Methods

__init__(*self*, *palette*, *layout*)

Constructs the vertex drawer and associates it to the given palette.

Parameters

`palette`: the palette that can be used to map integer color indices to colors when drawing vertices

`layout`: the layout of the vertices in the graph being drawn

Overrides: object.__init__

draw(*self*, *visual_vertex*, *vertex*, *coords*)

Draws the given vertex.

Parameters

`visual_vertex`: object specifying the visual properties of the vertex. Its structure is defined by the VisualVertexBuilder of the `DefaultGraphDrawer`; see its source code.

`vertex`: the raw igraph vertex being drawn

`coords`: the X and Y coordinates of the vertex as specified by the layout algorithm, scaled into the bounding box.

Overrides: igraph.drawing.baseclasses.AbstractDrawer.draw

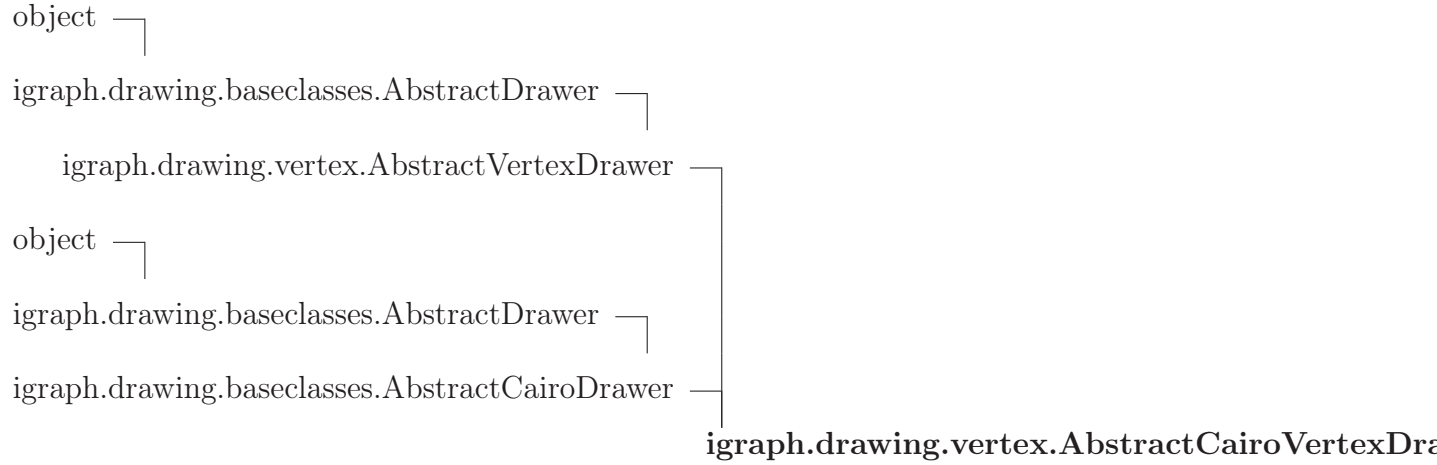
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

19.1.2 Properties

Name	Description
Inherited from object	
__class__	

19.2 Class AbstractCairoVertexDrawer



Known Subclasses: igraph.drawing.vertex.DefaultVertexDrawer

Abstract base class for vertex drawers that draw on a Cairo canvas.

19.2.1 Methods

__init__(*self*, *context*, *bbox*, *palette*, *layout*)

Constructs the vertex drawer and associates it to the given Cairo context and the given `BoundingBox`.

Parameters

`context`: the context on which we will draw

`bbox`: the bounding box within which we will draw. Can be anything accepted by the constructor of `BoundingBox` (i.e., a 2-tuple, a 4-tuple or a `BoundingBox` object).

`palette`: the palette that can be used to map integer color indices to colors when drawing vertices

`layout`: the layout of the vertices in the graph being drawn

Overrides: object.__init__

Inherited from igraph.drawing.vertex.AbstractVertexDrawer(Section 19.1)

draw()

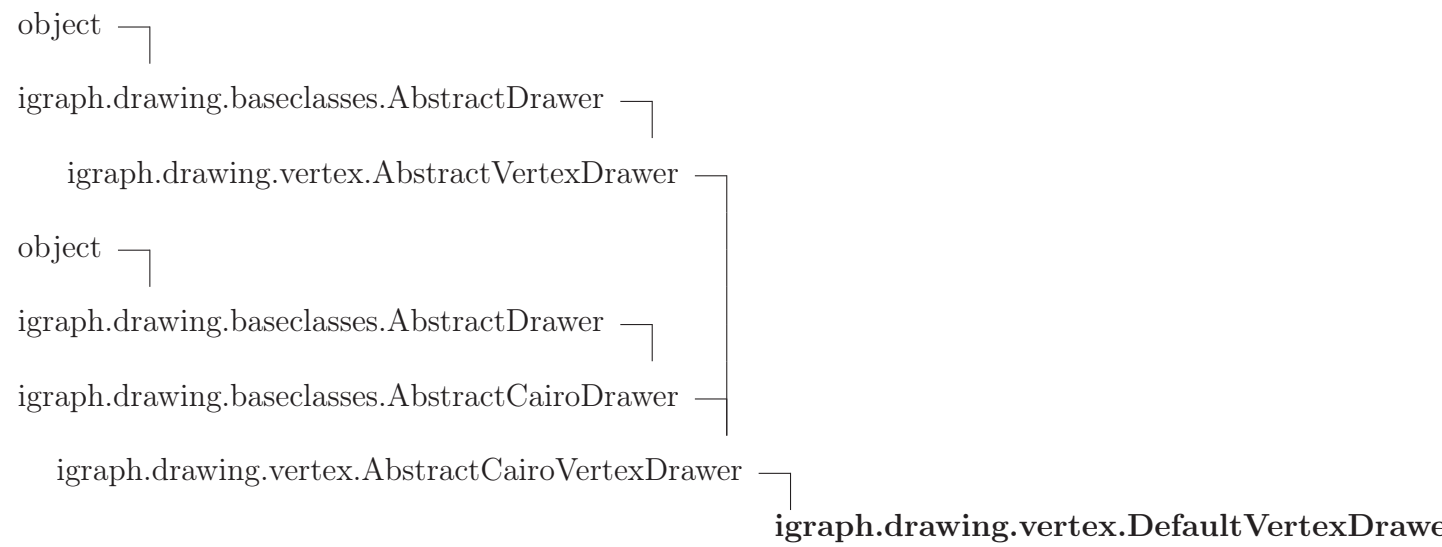
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

19.2.2 Properties

Name	Description
Inherited from igraph.drawing.baseclasses.AbstractCairoDrawer (Section 10.3)	
bbox	
Inherited from object	
__class__	

19.3 Class DefaultVertexDrawer



The default vertex drawer implementation of igraph.

19.3.1 Methods

__init__(*self*, *context*, *bbox*, *palette*, *layout*)

Constructs the vertex drawer and associates it to the given Cairo context and the given `BoundingBox`.

Parameters

`context:` the context on which we will draw

`bbox:` the bounding box within which we will draw. Can be anything accepted by the constructor of `BoundingBox` (i.e., a 2-tuple, a 4-tuple or a `BoundingBox` object).

`palette:` the palette that can be used to map integer color indices to colors when drawing vertices

`layout:` the layout of the vertices in the graph being drawn

Overrides: object.__init__ extit(inherited documentation)

draw(*self*, *visual_vertex*, *vertex*, *coords*)

Draws the given vertex.

Parameters

`visual_vertex`: object specifying the visual properties of the vertex. Its structure is defined by the VisualVertexBuilder of the `DefaultGraphDrawer`; see its source code.

`vertex`: the raw igraph vertex being drawn

`coords`: the X and Y coordinates of the vertex as specified by the layout algorithm, scaled into the bounding box.

Overrides: igraph.drawing.baseclasses.AbstractDrawer.draw extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

19.3.2 Properties

Name	Description
Inherited from igraph.drawing.baseclasses.AbstractCairoDrawer (Section 10.3)	
bbox	
Inherited from object	
__class__	

20 Module igraph.layout

Layout-related code in the IGraph library.

This package contains the implementation of the `Layout` object.

20.1 Variables

Name	Description
__package__	**Value:** `'igraph'`

20.2 Class Layout

object
└ **igraph.layout.Layout**

Represents the layout of a graph.

A layout is practically a list of coordinates in an n-dimensional space. This class is generic in the sense that it can store coordinates in any n-dimensional space.

Layout objects are not associated directly with a graph. This is deliberate: there were times when I worked with almost identical copies of the same graph, the only difference was that they had different colors assigned to the vertices. It was particularly convenient for me to use the same layout for all of them, especially when I made figures for a paper. However, `igraph` will of course refuse to draw a graph with a layout that has less coordinates than the node count of the graph.

Layouts behave exactly like lists when they are accessed using the item index operator (`[...]`). They can even be iterated through. Items returned by the index operator are only copies of the coordinates, but the stored coordinates can be modified by directly assigning to an index.

```
>>> layout = Layout([(0, 1), (0, 2)])
>>> coords = layout[1]
>>> print coords
[0, 2]
>>> coords = (0, 3)
>>> print layout[1]
[0, 2]
>>> layout[1] = coords
>>> print layout[1]
[0, 3]
```

20.2.1 Methods

__init__(*self*, *coords*=None, *dim*=None)

Constructor.

Parameters

`coords`: the coordinates to be stored in the layout.

`dim`: the number of dimensions. If `None`, the number of dimensions is determined automatically from the length of the first item of the coordinate list. If there are no entries in the coordinate list, the default will be 2. Generally, this should be given if the length of the coordinate list is zero, otherwise it should be left as is.

Overrides: object.__init__

__len__(*self*)

__getitem__(*self*, *idx*)

__setitem__(*self*, *idx*, *value*)

__delitem__(*self*, *idx*)

__copy__(*self*)

__repr__(*self*)

repr(x)

Overrides: object.__repr__ extit(inherited documentation)

append(*self*, *value*)

Appends a new point to the layout

mirror(*self*, *dim*)

Mirrors the layout along the given dimension(s)

Parameters

`dim`: the list of dimensions or a single dimension

rotate(*self*, *angle*, *dim1*=`0`, *dim2*=`1`, ***kwds*)

Rotates the layout by the given degrees on the plane defined by the given two dimensions.

Parameters

`angle`: the angle of the rotation, specified in degrees.

`dim1`: the first axis of the plane of the rotation.

`dim2`: the second axis of the plane of the rotation.

`origin`: the origin of the rotation. If not specified, the origin will be the origin of the coordinate system.

scale(*self*, **args*, ***kwds*)

Scales the layout.

Scaling parameters can be provided either through the `scale` keyword argument or through plain unnamed arguments. If a single integer or float is given, it is interpreted as a uniform multiplier to be applied on all dimensions. If it is a list or tuple, its length must be equal to the number of dimensions in the layout, and each element must be an integer or float describing the scaling coefficient in one of the dimensions.

Parameters

`scale`: scaling coefficients (integer, float, list or tuple)

`origin`: the origin of scaling (this point will stay in place). Optional, defaults to the origin of the coordinate system being used.

translate(*self*, **args*, ***kwds*)

Translates the layout.

The translation vector can be provided either through the `v` keyword argument or through plain unnamed arguments. If unnamed arguments are used, the vector can be supplied as a single list (or tuple) or just as a series of arguments. In all cases, the translation vector must have the same number of dimensions as the layout.

Parameters

`v`: the translation vector

to_radial(*self*, *min_angle*=`100`, *max_angle*=`80`, *min_radius*=`0.0`, *max_radius*=`1.0`)

Converts a planar layout to a radial one

This method applies only to 2D layouts. The X coordinate of the layout is transformed to an angle, with min(x) corresponding to the parameter called *min_angle* and max(y) corresponding to *max_angle*. Angles are given in degrees, zero degree corresponds to the direction pointing upwards. The Y coordinate is interpreted as a radius, with min(y) belonging to the minimum and max(y) to the maximum radius given in the arguments.

This is not a fully generic polar coordinate transformation, but it is fairly useful in creating radial tree layouts from ordinary top-down ones (that's why the Y coordinate belongs to the radius). It can also be used in conjunction with the Fruchterman-Reingold layout algorithm via its *miny* and *maxy* parameters (see `Graph.layout_fruchterman_reingold`) to produce radial layouts where the radius belongs to some property of the vertices.

Parameters

`min_angle`: the angle corresponding to the minimum X value

`max_angle`: the angle corresponding to the maximum X value

`min_radius`: the radius corresponding to the minimum Y value

`max_radius`: the radius corresponding to the maximum Y value

transform(*self*, *function*, **args*, ***kwds*)

Performs an arbitrary transformation on the layout

Additional positional and keyword arguments are passed intact to the given function.

Parameters

`function:` a function which receives the coordinates as a tuple and returns the transformed tuple.

centroid(*self*)

Returns the centroid of the layout.

The centroid of the layout is the arithmetic mean of the points in the layout.

Return Value

the centroid as a list of floats

boundaries(*self*, *border*=`0`)

Returns the boundaries of the layout.

The boundaries are the minimum and maximum coordinates along all dimensions.

Parameters

`border:` this value gets subtracted from the minimum bounds and gets added to the maximum bounds before returning the coordinates of the box. Defaults to zero.

Return Value

the minimum and maximum coordinates along all dimensions, in a tuple containing two lists, one for the minimum coordinates, the other one for the maximum.

Raises

`ValueError` if the layout contains no layout items

bounding_box(*self*, *border*=0)

Returns the bounding box of the layout.

The bounding box of the layout is the smallest box enclosing all the points in the layout.

Parameters

`border`: this value gets subtracted from the minimum bounds and gets added to the maximum bounds before returning the coordinates of the box. Defaults to zero.

Return Value

the coordinates of the lower left and the upper right corner of the box. "Lower left" means the minimum coordinates and "upper right" means the maximum. These are encapsulated in a `BoundingBox` object.

center(*self*, **args*, ***kwds*)

Centers the layout around the given point.

The point itself can be supplied as multiple unnamed arguments, as a simple unnamed list or as a keyword argument. This operation moves the centroid of the layout to the given point. If no point is supplied, defaults to the origin of the coordinate system.

Parameters

`p`: the point where the centroid of the layout will be after the operation.

copy(*self*)

Creates an exact copy of the layout.

fit_into(*self*, *bbox*, *keep_aspect_ratio*=`True`)

Fits the layout into the given bounding box.

The layout will be modified in-place.

Parameters

`bbox`: the bounding box in which to fit the layout. If the dimension of the layout is d, it can either be a d-tuple (defining the sizes of the box), a 2d-tuple (defining the coordinates of the top left and the bottom right point of the box), or a `BoundingBox` object (for 2D layouts only).

`keep_aspect_ratio`: whether to keep the aspect ratio of the current layout. If `False`, the layout will be rescaled to fit exactly into the bounding box. If `True`, the original aspect ratio of the layout will be kept and it will be centered within the bounding box.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

20.2.2 Properties

Name	Description
dim	Returns the number of dimensions
coords	The coordinates as a list of lists
Inherited from object	
__class__	

21 Module igraph.matching

Classes representing matchings on graphs.

21.1 Variables

Name	Description
__package__	**Value:** `'igraph'`

21.2 Class Matching

object
└ **igraph.matching.Matching**

A matching of vertices in a graph.

A matching of an undirected graph is a set of edges such that each vertex is incident on at most one matched edge. When each vertex is incident on *exactly* one matched edge, the matching called *perfect*. This class is used in `igraph` to represent non-perfect and perfect matchings in undirected graphs.

This class is usually not instantiated directly, everything is taken care of by the functions that return matchings.

Examples:

```
>>> from igraph import Graph
>>> g = Graph.Famous("noperfectmatching")
>>> matching = g.maximum_matching()
```

21.2.1 Methods

__init__(*self*, *graph*, *matching*, *types*=`None`)

Initializes the matching.

Parameters

`graph`: the graph the matching belongs to

`matching`: a numeric vector where element i corresponds to vertex i of the graph. Element i is -1 or if the corresponding vertex is unmatched, otherwise it contains the index of the vertex to which vertex i is matched.

`types`: the types of the vertices if the graph is bipartite. It must either be the name of a vertex attribute (which will be retrieved for all vertices) or a list. Elements in the list will be converted to boolean values `True` or `False`, and this will determine which part of the bipartite graph a given vertex belongs to.

Raises

`ValueError` if the matching vector supplied does not describe a valid matching of the graph.

Overrides: object.__init__

__len__(*self*)

__repr__(*self*)

repr(x)

Overrides: object.__repr__ extit(inherited documentation)

__str__(*self*)

str(x)

Overrides: object.__str__ extit(inherited documentation)

edges(*self*)

Returns an edge sequence that contains the edges in the matching.

If there are multiple edges between a pair of matched vertices, only one of them will be returned.

is_maximal(*self*)

Returns whether the matching is maximal.

A matching is maximal when it is not possible to extend it any more with extra edges; in other words, unmatched vertices in the graph must be adjacent to matched vertices only.

is_matched(*self*, *vertex*)

Returns whether the given vertex is matched to another one.

match_of(*self*, *vertex*)

Returns the vertex a given vertex is matched to.

Parameters

`vertex`: the vertex we are interested in; either an integer index or an instance of `Vertex`.

Return Value

the index of the vertex matched to the given vertex, either as an integer index (if *vertex* was integer) or as an instance of `Vertex`. When the vertex is unmatched, returns `None`.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __subclasshook__()

21.2.2 Properties

Name	Description
graph	Returns the graph corresponding to the matching.
matching	Returns the matching vector where element i contains the ID of the vertex that vertex i is matched to. The matching vector will contain `-1` for unmatched vertices.
types	Returns the type vector if the graph is bipartite. Element i of the type vector will be `False` or `True` depending on which side of the bipartite graph vertex i belongs to. For non-bipartite graphs, this property returns `None`.
Inherited from object __class__	

22 Module igraph.operators

Implementation of union, disjoint union and intersection operators. **License:** Copyright (C) 2006-2012 Tamás Nepusz <ntamas@gmail.com> Pázmány Péter sétány 1/a, 1117 Budapest, Hungary

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

22.1 Functions

disjoint_union(*graphs*)

Graph disjoint union.

The disjoint union of two or more graphs is created.

This function keeps the attributes of all graphs. All graph, vertex and edge attributes are copied to the result. If an attribute is present in multiple graphs and would result a name clash, then this attribute is renamed by adding suffixes: _1, _2, etc.

An error is generated if some input graphs are directed and others are undirected.

@param graph: list of graphs. A lazy sequence is not acceptable. @return: the disjoint union graph

union(*graphs*, *byname*='auto')

```
Graph union.

The union of two or more graphs is created. The graphs may have identical
or overlapping vertex sets. Edges which are included in at least one graph
will be part of the new graph.

This function keeps the attributes of all graphs. All graph, vertex and
edge attributes are copied to the result. If an attribute is present in
multiple graphs and would result a name clash, then this attribute is
renamed by adding suffixes: _1, _2, etc.

The 'name' vertex attribute is treated specially if the operation is
performed based on symbolic vertex names. In this case 'name' must be
present in all graphs, and it is not renamed in the result graph.

An error is generated if some input graphs are directed and others are
undirected.

@param graph: list of graphs. A lazy sequence is not acceptable.
@param byname: bool or 'auto' specifying the function behaviour with
  respect to names vertices (i.e. vertices with the 'name' attribute). If
  False, ignore vertex names. If True, merge vertices based on names. If
  'auto', use True if all graphs have named vertices and False otherwise
  (in the latter case, a warning is generated too).
@return: the union graph
```

intersection(*graphs*, *byname*='auto', *keep_all_vertices*=True)

```
Graph intersection.

The intersection of two or more graphs is created. The graphs may have
identical or overlapping vertex sets. Edges which are included in all
graphs will be part of the new graph.

This function keeps the attributes of all graphs. All graph, vertex and
edge attributes are copied to the result. If an attribute is present in
multiple graphs and would result a name clash, then this attribute is
renamed by adding suffixes: _1, _2, etc.

The 'name' vertex attribute is treated specially if the operation is
performed based on symbolic vertex names. In this case 'name' must be
present in all graphs, and it is not renamed in the result graph.

An error is generated if some input graphs are directed and others are
undirected.

@param graph: list of graphs. A lazy sequence is not acceptable.
@param byname: bool or 'auto' specifying the function behaviour with
  respect to names vertices (i.e. vertices with the 'name' attribute). If
  False, ignore vertex names. If True, merge vertices based on names. If
  'auto', use True if all graphs have named vertices and False otherwise
  (in the latter case, a warning is generated too).
@keep_all_vertices: bool specifying if vertices that are not present in all
  graphs should be kept in the intersection.
@return: the intersection graph
```

23 Package igraph.remote

Classes that help igraph communicate with remote applications.

23.1 Modules

- **gephi**: Classes that help igraph communicate with Gephi (http://www.gephi.org). *(Section 24, p. 387)*

23.2 Variables

Name	Description
__package__	**Value:** `None`

24 Module igraph.remote.gephi

Classes that help igraph communicate with Gephi (http://www.gephi.org). **License:** Copyright (C) 2006-2012 Tamás Nepusz <ntamas@gmail.com> Pázmány Péter sétány 1/a, 1117 Budapest, Hungary

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

24.1 Class GephiConnection

object
└ **igraph.remote.gephi.GephiConnection**

Object that represents a connection to a Gephi master server.

24.1.1 Methods

__init__(*self*, *url*=`None`, *host*=`'127.0.0.1'`, *port*=`8080`, *workspace*=`1`)

Constructs a connection to a Gephi master server.

The connection object can be constructed either by specifying the `url` directly, or by specifying the `host`, `port` and `workspace` arguments. The latter three are evaluated only if `url` is None; otherwise the `url` will take precedence.

The `url` argument does not have to include the operation (e.g., `?operation=updateGraph`); the connection will take care of it. E.g., if you wish to connect to workspace 2 in a local Gephi instance on port 7341, the correct form to use for the `url` is as follows:

```
http://localhost:7341/workspace0
```

Overrides: object.__init__

__del__(*self*)

close(*self*)

Flushes all the pending operations to the Gephi master server in a single request.

flush(*self*)

Flushes all the pending operations to the Gephi master server in a single request.

write(*self*, *data*)

Sends the given raw data to the Gephi streaming master server in an HTTP POST request.

__repr__(*self*)

repr(x)

Overrides: object.__repr__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

24.1.2 Properties

Name	Description
url	The URL of the Gephi workspace where the data will be sent.
Inherited from object __class__	

24.2 Class GephiGraphStreamingAPIFormat

object
└ **igraph.remote.gephi.GephiGraphStreamingAPIFormat**

Object that implements the Gephi graph streaming API format and returns Python objects

corresponding to the events defined in the API.

24.2.1 Methods

get_add_node_event(*self*, *identifier*, *attributes*={})

Generates a Python object corresponding to the event that adds a node with the given identifier and attributes in the Gephi graph streaming API.

Example:

```
>>> api = GephiGraphStreamingAPIFormat()
>>> api.get_add_node_event("spam")
{'an': {'spam': {}}}
>>> api.get_add_node_event("spam", dict(ham="eggs"))
{'an': {'spam': {'ham': 'eggs'}}}
```

get_add_edge_event(*self*, *identifier*, *source*, *target*, *directed*, *attributes*={})

Generates a Python object corresponding to the event that adds an edge with the given source, target, directednessr and attributes in the Gephi graph streaming API.

get_change_node_event(*self*, *identifier*, *attributes*)

Generates a Python object corresponding to the event that changes the attributes of some node in the Gephi graph streaming API. The given attributes are merged into the existing ones; use C{None} as the attribute value to delete a given attribute.

Example:

```
>>> api = GephiGraphStreamingAPIFormat()
>>> api.get_change_node_event("spam", dict(ham="eggs"))
{'cn': {'spam': {'ham': 'eggs'}}}
>>> api.get_change_node_event("spam", dict(ham=None))
{'cn': {'spam': {'ham': None}}}
```

get_change_edge_event(*self*, *identifier*, *attributes*)

Generates a Python object corresponding to the event that changes the attributes of some edge in the Gephi graph streaming API. The given attributes are merged into the existing ones; use C{None} as the attribute value to delete a given attribute.

Example:

```
>>> api = GephiGraphStreamingAPIFormat()
>>> api.get_change_edge_event("spam", dict(ham="eggs"))
{'ce': {'spam': {'ham': 'eggs'}}}
>>> api.get_change_edge_event("spam", dict(ham=None))
{'ce': {'spam': {'ham': None}}}
```

get_delete_node_event(*self*, *identifier*)

Generates a Python object corresponding to the event that deletes a node with the given identifier in the Gephi graph streaming API.

Example:

```
>>> api = GephiGraphStreamingAPIFormat()
>>> api.get_delete_node_event("spam")
{'dn': {'spam': {}}}
```

get_delete_edge_event(*self*, *identifier*)

Generates a Python object corresponding to the event that deletes an edge with the given identifier in the Gephi graph streaming API.

Example:

```
>>> api = GephiGraphStreamingAPIFormat()
>>> api.get_delete_edge_event("spam:ham")
{'de': {'spam:ham': {}}}
```

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

24.2.2 Properties

Name	Description
Inherited from object	
__class__	

24.3 Class GephiGraphStreamer

object
└ **igraph.remote.gephi.GephiGraphStreamer**

Class that produces JSON event objects that stream an igraph graph to Gephi using the Gephi Graph Streaming API.

The Gephi graph streaming format is a simple JSON-based format that can be used to post mutations to a graph (i.e. node and edge additions, removals and updates) to a remote component. For instance, one can open up Gephi (http://www.gephi.org}), install the Gephi graph streaming plugin and then send a graph from igraph straight into the Gephi window by using `GephiGraphStreamer` with the appropriate URL where Gephi is listening.

Example:

```
>>> from cStringIO import StringIO
>>> from igraph import Graph
>>> buf = StringIO()
>>> streamer = GephiGraphStreamer()
>>> graph = Graph.Formula("A --> B, B --> C")
>>> streamer.post(graph, buf)
>>> print buf.getvalue()         # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
{"an": {"igraph:...:v:0": {"name": "A"}}}
{"an": {"igraph:...:v:1": {"name": "B"}}}
{"an": {"igraph:...:v:2": {"name": "C"}}}
{"ae": {"igraph:...:e:0:1": {...}}}
{"ae": {"igraph:...:e:1:2": {...}}}
<BLANKLINE>
```

24.3.1 Methods

__init__(*self*, *encoder*=`None`)

Constructs a Gephi graph streamer that will post graphs to a given file-like object or a Gephi connection.

`encoder` must either be `None` or an instance of `json.JSONEncoder` and it must contain the JSON encoder to be used when posting JSON objects. Overrides: object.__init__

iterjsonobj(*self*, *graph*)

Iterates over the JSON objects that build up the graph using the Gephi graph streaming API. The objects returned from this function are Python objects; they must be formatted with `json.dumps` before sending them to the destination.

post(*self*, *graph*, *destination*, *encoder*=`None`)

Posts the given graph to the destination of the streamer using the given JSON encoder. When `encoder` is `None`, it falls back to the default JSON encoder of the streamer in the `encoder` property.

`destination` must be a file-like object or an instance of `GephiConnection`.

send_event(*self*, *event*, *destination*, *encoder*=`None`, *flush*=`True`)

Sends a single JSON event to the given destination using the given JSON encoder. When `encoder` is `None`, it falls back to the default JSON encoder of the streamer in the `encoder` property.

`destination` must be a file-like object or an instance of `GephiConnection`.

The method flushes the destination after sending the event. If you want to avoid this (e.g., because you are sending many events), set `flush` to `False`.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(),

__str__(), __subclasshook__()

24.3.2 Properties

Name	Description
Inherited from object	
__class__	

25 Module igraph.statistics

Statistics related stuff in igraph

25.1 Functions

mean(*xs*)

Returns the mean of an iterable.

Example:

```
>>> mean([1, 4, 7, 11])
5.75
```

Parameters

`xs`: an iterable yielding numbers.

Return Value

the mean of the numbers provided by the iterable.

See Also: RunningMean() if you also need the variance or the standard deviation

median(*xs*, *sort*=`True`)

Returns the median of an unsorted or sorted numeric vector.

Parameters

`xs`: the vector itself.

`sort`: whether to sort the vector. If you know that the vector is sorted already, pass `False` here.

Return Value

the median, which will always be a float, even if the vector contained integers originally.

percentile(*xs*, *p*=`(25, 50, 75)`, *sort*=`True`)

Returns the pth percentile of an unsorted or sorted numeric vector.

This is equivalent to calling quantile(xs, p/100.0); see `quantile` for more details on the calculation.

Example:

```
>>> round(percentile([15, 20, 40, 35, 50], 40), 2)
26.0
>>> for perc in percentile([15, 20, 40, 35, 50], (0, 25, 50, 75, 100)):
...     print "%.2f" % perc
...
15.00
17.50
35.00
45.00
50.00
```

Parameters

`xs`: the vector itself.

`p`: the percentile we are looking for. It may also be a list if you want to calculate multiple quantiles with a single call. The default value calculates the 25th, 50th and 75th percentile.

`sort`: whether to sort the vector. If you know that the vector is sorted already, pass `False` here.

Return Value

the pth percentile, which will always be a float, even if the vector contained integers originally. If p is a list, the result will also be a list containing the percentiles for each item in the list.

power_law_fit(*data*, *xmin*=`None`, *method*=`'auto'`, *return_alpha_only*=`False`)

Fitting a power-law distribution to empirical data

Parameters

`data`: the data to fit, a list containing integer values

`xmin`: the lower bound for fitting the power-law. If `None`, the optimal xmin value will be estimated as well. Zero means that the smallest possible xmin value will be used.

`method`: the fitting method to use. The following methods are implemented so far:

- `continuous`, `hill`: exact maximum likelihood estimation when the input data comes from a continuous scale. This is known as the Hill estimator. The statistical error of this estimator is *(alpha-1) / sqrt(n)*, where alpha is the estimated exponent and n is the number of data points above *xmin*. The estimator is known to exhibit a small finite sample-size bias of order *O(n^-1)*, which is small when $n > 100$. igraph will try to compensate for the finite sample size if n is small.
- `discrete`: exact maximum likelihood estimation when the input comes from a discrete scale (see Clauset et al among the references).
- `auto`: exact maximum likelihood estimation where the continuous method is used if the input vector contains at least one fractional value and the discrete method is used if the input vector contains integers only.

Return Value

a `FittedPowerLaw` object. The fitted `xmin` value and the power-law exponent can be queried from the `xmin` and `alpha` properties of the returned object.

Reference:

- MEJ Newman: Power laws, Pareto distributions and Zipf's law. Contemporary Physics 46, 323-351 (2005)
- A Clauset, CR Shalizi, MEJ Newman: Power-law distributions in empirical data. E-print (2007). arXiv:0706.1062

quantile(*xs*, *q*=(0.25, 0.5, 0.75), *sort*=True)

Returns the qth quantile of an unsorted or sorted numeric vector.

There are a number of different ways to calculate the sample quantile. The method implemented by igraph is the one recommended by NIST. First we calculate a rank n as q(N+1), where N is the number of items in xs, then we split n into its integer component k and decimal component d. If k <= 1, we return the first element; if k >= N, we return the last element, otherwise we return the linear interpolation between xs[k-1] and xs[k] using a factor d.

Example:

```
>>> round(quantile([15, 20, 40, 35, 50], 0.4), 2)
26.0
```

Parameters

`xs`: the vector itself.

`q`: the quantile we are looking for. It may also be a list if you want to calculate multiple quantiles with a single call. The default value calculates the 25th, 50th and 75th percentile.

`sort`: whether to sort the vector. If you know that the vector is sorted already, pass `False` here.

Return Value

the qth quantile, which will always be a float, even if the vector contained integers originally. If q is a list, the result will also be a list containing the quantiles for each item in the list.

25.2 Class FittedPowerLaw

object
└ **igraph.statistics.FittedPowerLaw**

Result of fitting a power-law to a vector of samples

Example:

```
>>> result = power_law_fit([1, 2, 3, 4, 5, 6])
>>> result                      # doctest:+ELLIPSIS
FittedPowerLaw(continuous=False, alpha=2.425828..., xmin=3.0, L=-7.54633..., D=0.2138..., p=0.99311...)
>>> print result                # doctest:+ELLIPSIS
Fitted power-law distribution on discrete data
<BLANKLINE>
Exponent (alpha)  = 2.425828
Cutoff (xmin)     = 3.000000
```

```
<BLANKLINE>
Log-likelihood    = -7.546337
<BLANKLINE>
H0: data was drawn from the fitted distribution
<BLANKLINE>
KS test statistic = 0.213817
p-value           = 0.993111
<BLANKLINE>
H0 could not be rejected at significance level 0.05
>>> result.alpha               # doctest:+ELLIPSIS
2.425828...
>>> result.xmin
3.0
>>> result.continuous
False
```

25.2.1 Methods

__init__(*self*, *continuous*, *alpha*, *xmin*, *L*, *D*, *p*)

x.__init__(...) initializes x; see help(type(x)) for signature

Overrides: object.__init__ extit(inherited documentation)

__repr__(*self*)

repr(x)

Overrides: object.__repr__ extit(inherited documentation)

__str__(*self*)

str(x)

Overrides: object.__str__ extit(inherited documentation)

summary(*self*, *significance*=0.05)

Returns the summary of the power law fit.

Parameters

`significance`: the significance level of the Kolmogorov-Smirnov test used to decide whether the input data could have come from the fitted distribution

Return Value

the summary as a string

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __subclasshook__()

25.2.2 Properties

Name	Description
Inherited from object	
__class__	

25.3 Class Histogram

object
└
igraph.statistics.Histogram

Generic histogram class for real numbers

Example:

```
>>> h = Histogram(5)       # Initializing, bin width = 5
>>> h << [2,3,2,7,8,5,5,0,7,9]      # Adding more items
>>> print h
N = 10, mean +- sd: 4.8000 +- 2.9740
[ 0,  5): **** (4)
[ 5, 10): ****** (6)
```

25.3.1 Methods

__init__(*self*, *bin_width*=1, *data*=None)

Initializes the histogram with the given data set.

Parameters

`bin_width`: the bin width of the histogram.

`data`: the data set to be used. Must contain real numbers.

Overrides: object.__init__

add(*self*, *num*, *repeat*=1)

Adds a single number to the histogram.

Parameters

`num`: the number to be added

`repeat`: number of repeated additions

add_many(*self*, *data*)

Adds a single number or the elements of an iterable to the histogram.

Parameters

`data`: the data to be added

__lshift__(*self*, *data*)

Adds a single number or the elements of an iterable to the histogram.

Parameters

`data`: the data to be added

clear(*self*)

Clears the collected data

bins(*self*)

Generator returning the bins of the histogram in increasing order

Return Value

a tuple with the following elements: left bound, right bound, number of elements in the bin

__plot__(*self*, *context*, *bbox*, *_*, ***kwds*)

Plotting support

to_string(*self*, *max_width*=`78`, *show_bars*=`True`, *show_counts*=`True`)

Returns the string representation of the histogram.

Parameters

`max_width`: the maximal width of each line of the string This value may not be obeyed if it is too small.

`show_bars`: specify whether the histogram bars should be shown

`show_counts`: specify whether the histogram counts should be shown. If both *show_bars* and *show_counts* are `False`, only a general descriptive statistics (number of elements, mean and standard deviation) is shown.

__str__(*self*)

str(x)

Overrides: object.__str__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

25.3.2 Properties

Name	Description
n	Returns the number of elements in the histogram
mean	Returns the mean of the elements in the histogram
sd	Returns the standard deviation of the elements in the histogram
var	Returns the variance of the elements in the histogram
Inherited from object __class__	

25.4 Class RunningMean

object
└ **igraph.statistics.RunningMean**

Running mean calculator.

This class can be used to calculate the mean of elements from a list, tuple, iterable or any other data source. The mean is calculated on the fly without explicitly summing the values, so it can be used for data sets with arbitrary item count. Also capable of returning the standard deviation (also calculated on the fly)

25.4.1 Methods

__init__(*items*=`None`, *n*=`0.0`, *mean*=`0.0`, *sd*=`0.0`)

Initializes the running mean calculator.

There are two possible ways to initialize the calculator. First, one can provide an iterable of items; alternatively, one can specify the number of items, the mean and the standard deviation if we want to continue an interrupted calculation.

Parameters

`items`: the items that are used to initialize the running mean calcuator. If `items` is given, `n`, `mean` and `sd` must be zeros.

`n`: the initial number of elements already processed. If this is given, `items` must be `None`.

`mean`: the initial mean. If this is given, `items` must be `None`.

`sd`: the initial standard deviation. If this is given, `items` must be `None`.

Overrides: object.__init__

add(*RunningMean*, *value*, *repeat*=`1`)

Adds the given value to the elements from which we calculate the mean and the standard deviation.

Parameters

`value`: the element to be added

`repeat`: number of repeated additions

add_many(*RunningMean*, *values*)

Adds the values in the given iterable to the elements from which we calculate the mean. Can also accept a single number. The left shift (`<<`) operator is aliased to this function, so you can use it to add elements as well:

```
>>> rm=RunningMean()
>>> rm << [1,2,3,4]
>>> rm.result                  # doctest:+ELLIPSIS
(2.5, 1.290994...)
```

Parameters

`values`: the element(s) to be added

(type=iterable)

clear(*self*)

Resets the running mean calculator.

__repr__(*self*)

repr(x)

Overrides: object.__repr__ extit(inherited documentation)

__str__(*self*)

str(x)

Overrides: object.__str__ extit(inherited documentation)

__lshift__(*RunningMean*, *values*)

Adds the values in the given iterable to the elements from which we calculate the mean. Can also accept a single number. The left shift (<<) operator is aliased to this function, so you can use it to add elements as well:

```
>>> rm=RunningMean()
>>> rm << [1,2,3,4]
>>> rm.result                  # doctest:+ELLIPSIS
(2.5, 1.290994...)
```

Parameters

`values`: the element(s) to be added
(type=iterable)

__float__(*self*)

__int__(*self*)

__long__(*self*)

__complex__(*self*)

__len__(*self*)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __subclasshook__()

25.4.2 Properties

Name	Description
result	Returns the current mean and standard deviation as a tuple
mean	Returns the current mean
sd	Returns the current standard deviation
var	Returns the current variation
Inherited from object __class__	

26 Module igraph.summary'

Summary representation of a graph.

26.1 Class GraphSummary

object
└ **igraph.summary'.GraphSummary**

Summary representation of a graph.

The summary representation includes a header line and the list of edges. The header line consists of `IGRAPH`, followed by a four-character long code, the number of vertices, the number of edges, two dashes (`-`) and the name of the graph (i.e. the contents of the `name` attribute, if any). For instance, a header line may look like this:

```
IGRAPH U--- 4 5 --
```

The four-character code describes some basic properties of the graph. The first character is `U` if the graph is undirected, `D` if it is directed. The second letter is `N` if the graph has a vertex attribute called `name`, or a dash otherwise. The third letter is `W` if the graph is weighted (i.e. it has an edge attribute called `weight`), or a dash otherwise. The fourth letter is `B` if the graph has a vertex attribute called `type`; this is usually used for bipartite graphs.

Edges may be presented as an ordinary edge list or an adjacency list. By default, this depends on the number of edges; however, you can control it with the appropriate constructor arguments.

26.1.1 Methods

__init__(*self*, *graph*, *verbosity*=0, *width*=78, *edge_list_format*='auto', *max_rows*=99999, *print_graph_attributes*=False, *print_vertex_attributes*=False, *print_edge_attributes*=False, *full*=False)

Constructs a summary representation of a graph.

Parameters

`verbosity:`	the verbosity of the summary. If zero, only the header line will be returned. If one, the header line and the list of edges will both be returned.
`width:`	the maximal width of each line in the summary. `None` means that no limit will be enforced.
`max_rows:`	the maximal number of rows to print in a single table (e.g., vertex attribute table or edge attribute table)
`edge_list_format:`	format of the edge list in the summary. Supported formats are: `compressed`, `adjlist`, `edgelist`, `auto`, which selects automatically from the other three based on some simple criteria.
`print_graph_attributes:`	whether to print graph attributes if there are any.
`print_vertex_attributes:`	whether to print vertex attributes if there are any.
`print_edge_attributes:`	whether to print edge attributes if there are any.
`full:`	False has no effect; True turns on the attribute printing for graph, vertex and edge attributes with verbosity 1.

Overrides: object.__init__

__str__(*self*)

Returns the summary representation as a string.

Overrides: object.__str__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(),

__reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

26.1.2 Properties

Name	Description
Inherited from object __class__	

27 Module igraph.utils

Utility functions that cannot be categorised anywhere else.

@undocumented: _is_running_in_ipython **License:** Copyright (C) 2006-2012 Tamás Nepusz <ntamas@gmail.com> Pázmány Péter sétány 1/a, 1117 Budapest, Hungary

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

27.1 Functions

named_temporary_file(**args*, ***kwds*)

Context manager that creates a named temporary file and returns its name.

All parameters are passed on to `tempfile.mkstemp`, see its documentation for more info.

numpy_to_contiguous_memoryview(*obj*)

Converts a NumPy array or matrix into a contiguous memoryview object that is suitable to be forwarded to the Graph constructor.

This is used internally to allow us to use a NumPy array or matrix directly when constructing a Graph.

rescale(*values*, *out_range*=(0.0, 1.0), *in_range*=None, *clamp*=False, *scale*=None)

Rescales a list of numbers into a given range.

`out_range` gives the range of the output values; by default, the minimum of the original numbers in the list will be mapped to the first element in the output range and the maximum will be mapped to the second element. Elements between the minimum and maximum values in the input list will be interpolated linearly between the first and second values of the output range.

`in_range` may be used to override which numbers are mapped to the first and second values of the output range. This must also be a tuple, where the first element will be mapped to the first element of the output range and the second element to the second.

If `clamp` is `True`, elements which are outside the given `out_range` after rescaling are clamped to the output range to ensure that no number will be outside `out_range` in the result.

If `scale` is not `None`, it will be called for every element of `values` and the rescaling will take place on the results instead. This can be used, for instance, to transform the logarithm of the original values instead of the actual values. A typical use-case is to map a range of values to color identifiers on a logarithmic scale. Scaling also applies to the `in_range` parameter if present.

Examples:

```
>>> rescale(range(5), (0, 8))
[0.0, 2.0, 4.0, 6.0, 8.0]
>>> rescale(range(5), (2, 10))
[2.0, 4.0, 6.0, 8.0, 10.0]
>>> rescale(range(5), (0, 4), (1, 3))
[-2.0, 0.0, 2.0, 4.0, 6.0]
>>> rescale(range(5), (0, 4), (1, 3), clamp=True)
[0.0, 0.0, 2.0, 4.0, 4.0]
>>> rescale([0]*5, (1, 3))
[2.0, 2.0, 2.0, 2.0, 2.0]
>>> from math import log10
>>> rescale([1, 10, 100, 1000, 10000], (0, 8), scale=log10)
[0.0, 2.0, 4.0, 6.0, 8.0]
>>> rescale([1, 10, 100, 1000, 10000], (0, 4), (10, 1000), scale=log10)
[-2.0, 0.0, 2.0, 4.0, 6.0]
```

safemax(*iterable*, *default*=0)

Safer variant of `max()` that returns a default value if the iterable is empty.

Example:

```
>>> safemax([-5, 6, 4])
6
>>> safemax([])
0
>>> safemax((), 2)
2
```

safemin(*iterable*, *default*=0)

Safer variant of `min()` that returns a default value if the iterable is empty.

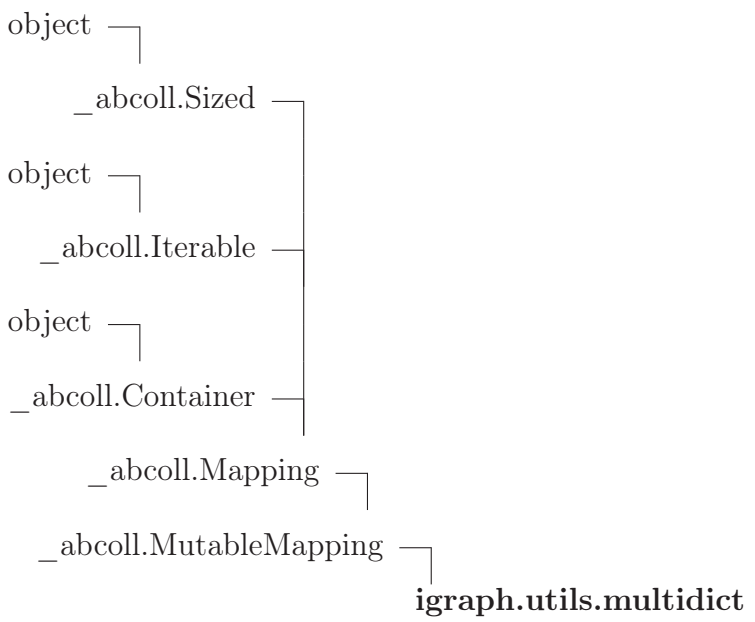
Example:

```
>>> safemin([-5, 6, 4])
-5
>>> safemin([])
0
>>> safemin((), 2)
2
```

27.2 Variables

Name	Description
dbl_epsilon	**Value:** `2.22044604925e-16`

27.3 Class multidict



A dictionary-like object that is customized to deal with multiple values for the same key.

Each value in this dictionary will be a list. Methods which emulate the methods of a standard Python `dict` object will return or manipulate the first items of the lists only. Special methods are provided to deal with keys having multiple values.

27.3.1 Methods

__init__(*self*, **args*, ***kwds*)

x.__init__(...) initializes x; see help(type(x)) for signature

Overrides: object.__init__ extit(inherited documentation)

__contains__(*self*, *key*)

Returns whether there are any items associated to the given `key`. Overrides: _abcoll.Container.__contains__

__delitem__(*self*, *key*)

Removes all the items associated to the given `key`. Overrides: _abcoll.MutableMapping.__delitem__

__getitem__(*self*, *key*)

Returns an arbitrary item associated to the given key. Raises `KeyError` if no such key exists.

Example:

```
>>> d = multidict([("spam", "eggs"), ("spam", "bacon")])
>>> d["spam"]
'eggs'
```

Overrides: _abcoll.Mapping.__getitem__

__iter__(*self*)

Iterates over the keys of the multidict. Overrides: _abcoll.Iterable.__iter__

__len__(*self*)

Returns the number of distinct keys in this multidict. Overrides: _abcoll.Sized.__len__

__setitem__(*self*, *key*, *value*)

Sets the item associated to the given `key`. Any values associated to the key will be erased and replaced by `value`.

Example:

```
>>> d = multidict([("spam", "eggs"), ("spam", "bacon")])
>>> d["spam"] = "ham"
>>> d["spam"]
'ham'
```

Overrides: _abcoll.MutableMapping.__setitem__

add(*self*, *key*, *value*)

Adds `value` to the list of items associated to `key`.

Example:

```
>>> d = multidict()
>>> d.add("spam", "ham")
>>> d["spam"]
'ham'
>>> d.add("spam", "eggs")
>>> d.getlist("spam")
['ham', 'eggs']
```

clear(*self*)

Removes all the items from the multidict. **Return Value**
None

Overrides: _abcoll.MutableMapping.clear

get(*self*, *key*, *default*=`None`)

Returns an arbitrary item associated to the given `key`. If `key` does not exist or has zero associated items, `default` will be returned. **Return Value**
D[k] if k in D, else d

Overrides: _abcoll.Mapping.get

getlist(*self*, *key*)

Returns the list of values for the given `key`. An empty list will be returned if there is no such key.

iterlists(*self*)

Iterates over `(key, values)` pairs where `values` is the list of values associated with `key`.

lists(*self*)

Returns a list of `(key, values)` pairs where `values` is the list of values associated with `key`.

update(*self*, *arg*, ***kwds*)

Update D from mapping/iterable E and F. If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

Return Value

None

Overrides: _abcoll.MutableMapping.update extit(inherited documentation)

Inherited from _abcoll.MutableMapping

pop(), popitem(), setdefault()

Inherited from _abcoll.Mapping

__eq__(), __ne__(), items(), iteritems(), iterkeys(), itervalues(), keys(), values()

Inherited from _abcoll.Sized

__subclasshook__()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__()

27.3.2 Properties

Name	Description
Inherited from object	
__class__	

27.3.3 Class Variables

Name	Description
__abstractmethods__	**Value:** `frozenset([])`
Inherited from _abcoll.Mapping	
__hash__	

28 Module igraph.version

Version: 0.8.3

28.1 Variables

Name	Description
__version_info__	**Value:** `(0, 8, 3)`
__package__	**Value:** `None`

Index

igraph *(package)*, 2–225
  igraph._igraph *(module)*, 226–230
    igraph._igraph.convex_hull *(function)*, 226
    igraph._igraph.InternalError *(class)*, 229–230
    igraph._igraph.is_degree_sequence *(function)*, 226
    igraph._igraph.is_graphical_degree_sequence *(function)*, 226
    igraph._igraph.set_progress_handler *(function)*, 227
    igraph._igraph.set_random_number_generator *(function)*, 227
    igraph._igraph.set_status_handler *(function)*, 227
  igraph.app *(package)*, 231
    igraph.app.shell *(module)*, 232–240
  igraph.ARPACKOptions *(class)*, 103–104
  igraph.autocurve *(function)*, 3
  igraph.BFSIter *(class)*, 104–105
    igraph.BFSIter.__iter__ *(method)*, 105
    igraph.BFSIter.next *(method)*, 105
  igraph.clustering *(module)*, 241–268
    igraph.clustering.Clustering *(class)*, 245–248
    igraph.clustering.CohesiveBlocks *(class)*, 265–268
    igraph.clustering.compare_communities *(function)*, 243
    igraph.clustering.Cover *(class)*, 259–262
    igraph.clustering.Dendrogram *(class)*, 254–257
    igraph.clustering.split_join_distance *(function)*, 243
    igraph.clustering.VertexClustering *(class)*, 248–254
    igraph.clustering.VertexCover *(class)*, 262–265
    igraph.clustering.VertexDendrogram *(class)*, 257–259
  igraph.configuration *(module)*, 269–273
    igraph.configuration.Configuration *(class)*, 270–273
    igraph.configuration.get_platform_image_viewer *(function)*, 269
    igraph.configuration.get_user_config_file *(function)*, 269
    igraph.configuration.init *(function)*, 269
  igraph.cut *(module)*, 274–280
    igraph.cut.Cut *(class)*, 274–277
    igraph.cut.Flow *(class)*, 277–280
  igraph.datatypes *(module)*, 281–291
    igraph.datatypes.DyadCensus *(class)*, 286–288
    igraph.datatypes.Matrix *(class)*, 281–286
    igraph.datatypes.TriadCensus *(class)*, 288–289
    igraph.datatypes.UniqueIdGenerator *(class)*, 289–291
  igraph.DFSIter *(class)*, 105–106
    igraph.DFSIter.__iter__ *(method)*, 105
    igraph.DFSIter.next *(method)*, 105
  igraph.drawing *(package)*, 292–311
    igraph.drawing.baseclasses *(module)*, 312–314
    igraph.drawing.colors *(module)*, 315–327
    igraph.drawing.coord *(module)*, 328–330
    igraph.drawing.edge *(module)*, 331–338
    igraph.drawing.graph *(module)*, 339–345
    igraph.drawing.metamagic *(module)*, 346–349
    igraph.drawing.Plot *(class)*, 307–311
    igraph.drawing.plot *(function)*, 294
    igraph.drawing.shapes *(module)*, 349–351
    igraph.drawing.text *(module)*, 352–356
    igraph.drawing.utils *(module)*, 357–367
    igraph.drawing.vertex *(module)*, 368–372
  igraph.Edge *(class)*, 106–109
    igraph.Edge.__delitem__ *(method)*, 106
    igraph.Edge.__eq__ *(method)*, 106
    igraph.Edge.__ge__ *(method)*, 106
    igraph.Edge.__getitem__ *(method)*, 106
    igraph.Edge.__gt__ *(method)*, 106

igraph.Edge.__le__ *(method)*, 107
igraph.Edge.__len__ *(method)*, 107
igraph.Edge.__lt__ *(method)*, 107
igraph.Edge.__ne__ *(method)*, 107
igraph.Edge.__setitem__ *(method)*, 107
igraph.Edge.attribute_names *(method)*, 107
igraph.Edge.attributes *(method)*, 107
igraph.Edge.count_multiple *(method)*, 107
igraph.Edge.delete *(method)*, 108
igraph.Edge.is_loop *(method)*, 108
igraph.Edge.is_multiple *(method)*, 108
igraph.Edge.is_mutual *(method)*, 108
igraph.Edge.update_attributes *(method)*, 108
igraph.EdgeSeq *(class)*, 98–103
igraph.EdgeSeq.__call__ *(method)*, 101
igraph.EdgeSeq.attributes *(method)*, 100
igraph.EdgeSeq.count_multiple *(method)*, 102
igraph.EdgeSeq.delete *(method)*, 102
igraph.EdgeSeq.edge_betweenness *(method)*, 102
igraph.EdgeSeq.is_loop *(method)*, 102
igraph.EdgeSeq.is_multiple *(method)*, 102
igraph.EdgeSeq.is_mutual *(method)*, 102
igraph.EdgeSeq.subgraph *(method)*, 103
igraph.get_include *(function)*, 3
igraph.Graph *(class)*, 11–90
igraph.Graph.__add__ *(method)*, 74
igraph.Graph.__and__ *(method)*, 74
igraph.Graph.__coerce__ *(method)*, 75
igraph.Graph.__iadd__ *(method)*, 73
igraph.Graph.__isub__ *(method)*, 74
igraph.Graph.__mul__ *(method)*, 74
igraph.Graph.__nonzero__ *(method)*, 75
igraph.Graph.__or__ *(method)*, 75
igraph.Graph.__plot__ *(method)*, 75
igraph.Graph.__sub__ *(method)*, 74
igraph.Graph.add_edge *(method)*, 22
igraph.Graph.add_vertex *(method)*, 23
igraph.Graph.adjacent *(method)*, 23
igraph.Graph.alpha *(method)*, 13, 173
igraph.Graph.as_directed *(method)*, 24
igraph.Graph.as_undirected *(method)*, 24
igraph.Graph.biconnected_components *(method)*, 26
igraph.Graph.Bipartite *(class method)*, 66
igraph.Graph.clear *(method)*, 26
igraph.Graph.clusters *(method)*, 27
igraph.Graph.community_leading_eigenvector_naiv *(method)*, 40
igraph.Graph.count_automorphisms_vf2 *(method)*, 38
igraph.Graph.cut_vertices *(method)*, 13, 130
igraph.Graph.DataFrame *(class method)*, 70
igraph.Graph.degree_distribution *(method)*, 28
igraph.Graph.dfs *(method)*, 73
igraph.Graph.DictList *(class method)*, 63
igraph.Graph.disjoint_union *(method)*, 77
igraph.Graph.edge_disjoint_paths *(method)*, 16, 18, 160
igraph.Graph.evcent *(method)*, 14, 161
igraph.Graph.Formula *(class method)*, 65
igraph.Graph.from_graph_tool *(class method)*, 54
igraph.Graph.from_networkx *(class method)*, 53
igraph.Graph.Full_Bipartite *(class method)*, 67
igraph.Graph.get_adjacency_sparse *(method)*, 29
igraph.Graph.get_adjedgelist *(method)*, 30
igraph.Graph.get_adjlist *(method)*, 29
igraph.Graph.get_all_simple_paths *(method)*, 30
igraph.Graph.get_automorphisms_vf2 *(method)*, 39

igraph.Graph.get_inclist *(method)*, 31
igraph.Graph.GRG *(class method)*, 69
igraph.Graph.Incidence *(class method)*, 69
igraph.Graph.indegree *(method)*, 24
igraph.Graph.intersection *(method)*, 78
igraph.Graph.is_named *(method)*, 32
igraph.Graph.is_weighted *(method)*, 32
igraph.Graph.k_core *(method)*, 48
igraph.Graph.layout *(method)*, 49
igraph.Graph.layout_auto *(method)*, 50
igraph.Graph.layout_fruchterman_reingold_3d *(method)*, 78
igraph.Graph.layout_grid_3d *(method)*, 78
igraph.Graph.layout_grid_fruchterman_reingold *(method)*, 51
igraph.Graph.layout_kamada_kawai_3d *(method)*, 78
igraph.Graph.layout_random_3d *(method)*, 78
igraph.Graph.layout_sphere *(method)*, 78
igraph.Graph.layout_sugiyama *(method)*, 51
igraph.Graph.maximum_bipartite_matching *(method)*, 52
igraph.Graph.omega *(method)*, 13, 136
igraph.Graph.outdegree *(method)*, 25
igraph.Graph.pagerank *(method)*, 35
igraph.Graph.Random_Bipartite *(class method)*, 68
igraph.Graph.Read *(class method)*, 59, 60
igraph.Graph.Read_Adjacency *(class method)*, 55
igraph.Graph.Read_GraphMLz *(class method)*, 57
igraph.Graph.Read_Pickle *(class method)*, 58
igraph.Graph.Read_Picklez *(class method)*, 58
igraph.Graph.shell_index *(method)*, 13, 152
igraph.Graph.shortest_paths_dijkstra *(method)*, 19, 208
igraph.Graph.spanning_tree *(method)*, 36
igraph.Graph.subgraph *(method)*, 20, 174
igraph.Graph.summary *(method)*, 77
igraph.Graph.to_graph_tool *(method)*, 53
igraph.Graph.to_networkx *(method)*, 53
igraph.Graph.TupleList *(class method)*, 64
igraph.Graph.union *(method)*, 77
igraph.Graph.vertex_disjoint_paths *(method)*, 15, 17, 221
igraph.Graph.write *(method)*, 61, 62
igraph.Graph.write_adjacency *(method)*, 54
igraph.Graph.write_graphmlz *(method)*, 56
igraph.Graph.write_pickle *(method)*, 57
igraph.Graph.write_picklez *(method)*, 58
igraph.Graph.write_svg *(method)*, 58
igraph.GraphBase *(class)*, 109–225
igraph.GraphBase.__delitem__ *(method)*, 128
igraph.GraphBase.__getitem__ *(method)*, 128
igraph.GraphBase.__invert__ *(method)*, 128
igraph.GraphBase.__setitem__ *(method)*, 129
igraph.GraphBase.add_edges *(method)*, 129
igraph.GraphBase.add_vertices *(method)*, 129
igraph.GraphBase.Adjacency *(method)*, 110
igraph.GraphBase.all_minimal_st_separators *(method)*, 129
igraph.GraphBase.all_st_cuts *(method)*, 129
igraph.GraphBase.all_st_mincuts *(method)*, 130
igraph.GraphBase.are_connected *(method)*,

130
igraph.GraphBase.assortativity *(method)*, 130
igraph.GraphBase.assortativity_degree *(method)*, 131
igraph.GraphBase.assortativity_nominal *(method)*, 132
igraph.GraphBase.Asymmetric_Preference *(method)*, 110
igraph.GraphBase.Atlas *(method)*, 111
igraph.GraphBase.attributes *(method)*, 132
igraph.GraphBase.authority_score *(method)*, 132
igraph.GraphBase.average_path_length *(method)*, 133
igraph.GraphBase.Barabasi *(method)*, 111
igraph.GraphBase.betweenness *(method)*, 133
igraph.GraphBase.bfs *(method)*, 134
igraph.GraphBase.bfsiter *(method)*, 134
igraph.GraphBase.bibcoupling *(method)*, 135
igraph.GraphBase.biconnected_components *(method)*, 135
igraph.GraphBase.bipartite_projection *(method)*, 135
igraph.GraphBase.bipartite_projection_size *(method)*, 135
igraph.GraphBase.bridges *(method)*, 136
igraph.GraphBase.canonical_permutation *(method)*, 136
igraph.GraphBase.cliques *(method)*, 137
igraph.GraphBase.closeness *(method)*, 137
igraph.GraphBase.clusters *(method)*, 138
igraph.GraphBase.cocitation *(method)*, 139
igraph.GraphBase.cohesive_blocks *(method)*, 139
igraph.GraphBase.community_edge_betweenness *(method)*, 139
igraph.GraphBase.community_fastgreedy *(method)*, 140
igraph.GraphBase.community_infomap *(method)*, 141
igraph.GraphBase.community_label_propagation *(method)*, 142
igraph.GraphBase.community_leading_eigenvector *(method)*, 143
igraph.GraphBase.community_leiden *(method)*, 144
igraph.GraphBase.community_multilevel *(method)*, 145
igraph.GraphBase.community_optimal_modularity *(method)*, 146
igraph.GraphBase.community_spinglass *(method)*, 147
igraph.GraphBase.community_walktrap *(method)*, 148
igraph.GraphBase.complementer *(method)*, 149
igraph.GraphBase.compose *(method)*, 149
igraph.GraphBase.constraint *(method)*, 149
igraph.GraphBase.contract_vertices *(method)*, 150
igraph.GraphBase.convergence_degree *(method)*, 151
igraph.GraphBase.convergence_field_size *(method)*, 151
igraph.GraphBase.copy *(method)*, 151
igraph.GraphBase.count_isomorphisms_vf2 *(method)*, 152
igraph.GraphBase.count_multiple *(method)*, 153
igraph.GraphBase.count_subisomorphisms_vf2 *(method)*, 154
igraph.GraphBase.De_Bruijn *(method)*, 112
igraph.GraphBase.decompose *(method)*, 155
igraph.GraphBase.degree *(method)*, 156
igraph.GraphBase.Degree_Sequence *(method)*, 113
igraph.GraphBase.delete_edges *(method)*, 156
igraph.GraphBase.delete_vertices *(method)*, 156

igraph.GraphBase.density *(method)*, 157
igraph.GraphBase.dfsiter *(method)*, 157
igraph.GraphBase.diameter *(method)*, 157
igraph.GraphBase.difference *(method)*, 158
igraph.GraphBase.diversity *(method)*, 158
igraph.GraphBase.dominator *(method)*, 158
igraph.GraphBase.dyad_census *(method)*, 159
igraph.GraphBase.eccentricity *(method)*, 159
igraph.GraphBase.ecount *(method)*, 159
igraph.GraphBase.edge_attributes *(method)*, 160
igraph.GraphBase.edge_betweenness *(method)*, 160
igraph.GraphBase.eigen_adjacency *(method)*, 161
igraph.GraphBase.Erdos_Renyi *(method)*, 114
igraph.GraphBase.Establishment *(method)*, 115
igraph.GraphBase.Famous *(method)*, 115
igraph.GraphBase.farthest_points *(method)*, 162
igraph.GraphBase.feedback_arc_set *(method)*, 163
igraph.GraphBase.Forest_Fire *(method)*, 115
igraph.GraphBase.Full *(method)*, 116
igraph.GraphBase.Full_Citation *(method)*, 116
igraph.GraphBase.get_adjacency *(method)*, 164
igraph.GraphBase.get_all_shortest_paths *(method)*, 164
igraph.GraphBase.get_diameter *(method)*, 165
igraph.GraphBase.get_edgelist *(method)*, 165
igraph.GraphBase.get_eid *(method)*, 165
igraph.GraphBase.get_eids *(method)*, 166
igraph.GraphBase.get_incidence *(method)*, 166
igraph.GraphBase.get_isomorphisms_vf2 *(method)*, 167
igraph.GraphBase.get_shortest_paths *(method)*, 168
igraph.GraphBase.get_subisomorphisms_lad *(method)*, 169
igraph.GraphBase.get_subisomorphisms_vf2 *(method)*, 170
igraph.GraphBase.girth *(method)*, 171
igraph.GraphBase.gomory_hu_tree *(method)*, 172
igraph.GraphBase.Growing_Random *(method)*, 116
igraph.GraphBase.has_multiple *(method)*, 172
igraph.GraphBase.hub_score *(method)*, 172
igraph.GraphBase.incident *(method)*, 173
igraph.GraphBase.independent_vertex_sets *(method)*, 173
igraph.GraphBase.is_bipartite *(method)*, 174
igraph.GraphBase.is_connected *(method)*, 175
igraph.GraphBase.is_dag *(method)*, 175
igraph.GraphBase.is_directed *(method)*, 175
igraph.GraphBase.is_loop *(method)*, 175
igraph.GraphBase.is_minimal_separator *(method)*, 176
igraph.GraphBase.is_multiple *(method)*, 176
igraph.GraphBase.is_mutual *(method)*, 176
igraph.GraphBase.is_separator *(method)*, 177
igraph.GraphBase.is_simple *(method)*, 177
igraph.GraphBase.Isoclass *(method)*, 117
igraph.GraphBase.isoclass *(method)*, 177
igraph.GraphBase.isomorphic *(method)*, 178
igraph.GraphBase.isomorphic_bliss *(method)*,

178
igraph.GraphBase.isomorphic_vf2 *(method)*, 179
igraph.GraphBase.K_Regular *(method)*, 117
igraph.GraphBase.Kautz *(method)*, 117
igraph.GraphBase.knn *(method)*, 180
igraph.GraphBase.laplacian *(method)*, 181
igraph.GraphBase.largest_cliques *(method)*, 181
igraph.GraphBase.largest_independent_vertex_sets *(method)*, 182
igraph.GraphBase.Lattice *(method)*, 118
igraph.GraphBase.layout_bipartite *(method)*, 182
igraph.GraphBase.layout_circle *(method)*, 182
igraph.GraphBase.layout_davidson_harel *(method)*, 183
igraph.GraphBase.layout_drl *(method)*, 184
igraph.GraphBase.layout_fruchterman_reingold *(method)*, 185
igraph.GraphBase.layout_graphopt *(method)*, 186
igraph.GraphBase.layout_grid *(method)*, 187
igraph.GraphBase.layout_kamada_kawai *(method)*, 188
igraph.GraphBase.layout_lgl *(method)*, 189
igraph.GraphBase.layout_mds *(method)*, 190
igraph.GraphBase.layout_random *(method)*, 191
igraph.GraphBase.layout_reingold_tilford *(method)*, 191
igraph.GraphBase.layout_reingold_tilford_circular *(method)*, 192
igraph.GraphBase.layout_star *(method)*, 193
igraph.GraphBase.LCF *(method)*, 118
igraph.GraphBase.linegraph *(method)*, 193
igraph.GraphBase.maxdegree *(method)*, 193
igraph.GraphBase.maxflow *(method)*, 194
igraph.GraphBase.maxflow_value *(method)*, 194
igraph.GraphBase.maximal_cliques *(method)*, 195
igraph.GraphBase.maximal_independent_vertex_sets *(method)*, 195
igraph.GraphBase.mincut *(method)*, 196
igraph.GraphBase.mincut_value *(method)*, 197
igraph.GraphBase.minimum_size_separators *(method)*, 198
igraph.GraphBase.modularity *(method)*, 198
igraph.GraphBase.motifs_randesu *(method)*, 199
igraph.GraphBase.motifs_randesu_estimate *(method)*, 200
igraph.GraphBase.motifs_randesu_no *(method)*, 201
igraph.GraphBase.neighborhood *(method)*, 201
igraph.GraphBase.neighborhood_size *(method)*, 202
igraph.GraphBase.neighbors *(method)*, 203
igraph.GraphBase.path_length_hist *(method)*, 203
igraph.GraphBase.permute_vertices *(method)*, 204
igraph.GraphBase.personalized_pagerank *(method)*, 204
igraph.GraphBase.predecessors *(method)*, 205
igraph.GraphBase.Preference *(method)*, 118
igraph.GraphBase.radius *(method)*, 206
igraph.GraphBase.random_walk *(method)*, 206
igraph.GraphBase.Read_DIMACS *(method)*, 119
igraph.GraphBase.Read_DL *(method)*, 119

igraph.GraphBase.Read_Edgelist *(method)*, 120
igraph.GraphBase.Read_GML *(method)*, 120
igraph.GraphBase.Read_GraphDB *(method)*, 120
igraph.GraphBase.Read_GraphML *(method)*, 120
igraph.GraphBase.Read_Lgl *(method)*, 120
igraph.GraphBase.Read_Ncol *(method)*, 121
igraph.GraphBase.Read_Pajek *(method)*, 122
igraph.GraphBase.Recent_Degree *(method)*, 122
igraph.GraphBase.reciprocity *(method)*, 206
igraph.GraphBase.rewire *(method)*, 207
igraph.GraphBase.rewire_edges *(method)*, 207
igraph.GraphBase.Ring *(method)*, 123
igraph.GraphBase.SBM *(method)*, 123
igraph.GraphBase.similarity_dice *(method)*, 208
igraph.GraphBase.similarity_inverse_log_weighted *(method)*, 209
igraph.GraphBase.similarity_jaccard *(method)*, 210
igraph.GraphBase.simplify *(method)*, 211
igraph.GraphBase.st_mincut *(method)*, 212
igraph.GraphBase.Star *(method)*, 124
igraph.GraphBase.Static_Fitness *(method)*, 124
igraph.GraphBase.Static_Power_Law *(method)*, 125
igraph.GraphBase.strength *(method)*, 213
igraph.GraphBase.subcomponent *(method)*, 213
igraph.GraphBase.subgraph_edges *(method)*, 214
igraph.GraphBase.subisomorphic_lad *(method)*, 214
igraph.GraphBase.subisomorphic_vf2 *(method)*, 215
igraph.GraphBase.successors *(method)*, 216
igraph.GraphBase.to_directed *(method)*, 217
igraph.GraphBase.to_prufer *(method)*, 217
igraph.GraphBase.to_undirected *(method)*, 217
igraph.GraphBase.topological_sorting *(method)*, 217
igraph.GraphBase.transitivity_avglocal_undirected *(method)*, 218
igraph.GraphBase.transitivity_local_undirected *(method)*, 218
igraph.GraphBase.transitivity_undirected *(method)*, 219
igraph.GraphBase.Tree *(method)*, 126
igraph.GraphBase.triad_census *(method)*, 220
igraph.GraphBase.unfold_tree *(method)*, 220
igraph.GraphBase.vcount *(method)*, 221
igraph.GraphBase.vertex_attributes *(method)*, 221
igraph.GraphBase.Watts_Strogatz *(method)*, 127
igraph.GraphBase.Weighted_Adjacency *(method)*, 127
igraph.GraphBase.write_dimacs *(method)*, 222
igraph.GraphBase.write_dot *(method)*, 223
igraph.GraphBase.write_edgelist *(method)*, 223
igraph.GraphBase.write_gml *(method)*, 223
igraph.GraphBase.write_graphml *(method)*, 223
igraph.GraphBase.write_leda *(method)*, 224
igraph.GraphBase.write_lgl *(method)*, 224
igraph.GraphBase.write_ncol *(method)*,

224
igraph.GraphBase.write_pajek *(method)*, 225
igraph.layout *(module)*, 373–379
igraph.layout.Layout *(class)*, 373–379
igraph.matching *(module)*, 380–383
igraph.matching.Matching *(class)*, 380–383
igraph.operators *(module)*, 383–385
igraph.operators.disjoint_union *(function)*, 383
igraph.operators.intersection *(function)*, 384
igraph.operators.union *(function)*, 383
igraph.read *(function)*, 4
igraph.remote *(package)*, 386
igraph.remote.gephi *(module)*, 387–393
igraph.statistics *(module)*, 394–404
igraph.statistics.FittedPowerLaw *(class)*, 397–399
igraph.statistics.Histogram *(class)*, 399–401
igraph.statistics.mean *(function)*, 394
igraph.statistics.median *(function)*, 394
igraph.statistics.percentile *(function)*, 395
igraph.statistics.power_law_fit *(function)*, 395
igraph.statistics.quantile *(function)*, 396
igraph.statistics.RunningMean *(class)*, 401–404
igraph.summary *(function)*, 4
igraph.summary' *(module)*, 405–407
igraph.summary'.GraphSummary *(class)*, 405–407
igraph.utils *(module)*, 408–414
igraph.utils.multidict *(class)*, 410–414
igraph.utils.named_temporary_file *(function)*, 408
igraph.utils.numpy_to_contiguous_memoryview *(function)*, 408
igraph.utils.rescale *(function)*, 408
igraph.utils.safemax *(function)*, 409
igraph.utils.safemin *(function)*, 410
igraph.version *(module)*, 415
igraph.Vertex *(class)*, 5–11
igraph.Vertex.__delitem__ *(method)*, 6
igraph.Vertex.__eq__ *(method)*, 6
igraph.Vertex.__ge__ *(method)*, 6
igraph.Vertex.__getitem__ *(method)*, 6
igraph.Vertex.__gt__ *(method)*, 6
igraph.Vertex.__le__ *(method)*, 6
igraph.Vertex.__len__ *(method)*, 6
igraph.Vertex.__lt__ *(method)*, 6
igraph.Vertex.__ne__ *(method)*, 6
igraph.Vertex.__setitem__ *(method)*, 7
igraph.Vertex.all_edges *(method)*, 7
igraph.Vertex.attribute_names *(method)*, 7
igraph.Vertex.attributes *(method)*, 7
igraph.Vertex.betweenness *(method)*, 7
igraph.Vertex.closeness *(method)*, 7
igraph.Vertex.constraint *(method)*, 7
igraph.Vertex.degree *(method)*, 8
igraph.Vertex.delete *(method)*, 8
igraph.Vertex.diversity *(method)*, 8
igraph.Vertex.eccentricity *(method)*, 8
igraph.Vertex.get_shortest_paths *(method)*, 8
igraph.Vertex.in_edges *(method)*, 8
igraph.Vertex.incident *(method)*, 9
igraph.Vertex.indegree *(method)*, 9
igraph.Vertex.is_minimal_separator *(method)*, 9
igraph.Vertex.is_separator *(method)*, 9
igraph.Vertex.neighbors *(method)*, 9
igraph.Vertex.out_edges *(method)*, 9
igraph.Vertex.outdegree *(method)*, 10
igraph.Vertex.pagerank *(method)*, 10
igraph.Vertex.personalized_pagerank *(method)*, 10
igraph.Vertex.predecessors *(method)*, 10
igraph.Vertex.shortest_paths *(method)*, 10
igraph.Vertex.strength *(method)*, 10
igraph.Vertex.successors *(method)*, 11

igraph.Vertex.update_attributes *(method)*, 11
igraph.VertexSeq *(class)*, 90–98
igraph.VertexSeq.__call__ *(method)*, 93
igraph.VertexSeq.attributes *(method)*, 92
igraph.VertexSeq.betweenness *(method)*, 94
igraph.VertexSeq.bibcoupling *(method)*, 94
igraph.VertexSeq.closeness *(method)*, 94
igraph.VertexSeq.cocitation *(method)*, 94
igraph.VertexSeq.constraint *(method)*, 94
igraph.VertexSeq.degree *(method)*, 94
igraph.VertexSeq.delete *(method)*, 95
igraph.VertexSeq.diversity *(method)*, 95
igraph.VertexSeq.eccentricity *(method)*, 95
igraph.VertexSeq.get_shortest_paths *(method)*, 95
igraph.VertexSeq.indegree *(method)*, 95
igraph.VertexSeq.is_minimal_separator *(method)*, 96
igraph.VertexSeq.is_separator *(method)*, 96
igraph.VertexSeq.isoclass *(method)*, 96
igraph.VertexSeq.maxdegree *(method)*, 96
igraph.VertexSeq.outdegree *(method)*, 96
igraph.VertexSeq.pagerank *(method)*, 97
igraph.VertexSeq.personalized_pagerank *(method)*, 97
igraph.VertexSeq.shortest_paths *(method)*, 97
igraph.VertexSeq.similarity_dice *(method)*, 97
igraph.VertexSeq.similarity_jaccard *(method)*, 97
igraph.VertexSeq.subgraph *(method)*, 98
igraph.write *(function)*, 4