graphi Documentation

Release 0.2.0

Max Fischer

Nov 17, 2017

Documentation Topics Overview:

CHAPTER 1

Quick Usage Reference

`GraphI` is primarily meant for working directly on graph data. The primitives you need to familiarise yourself with are

1. graphs, which are extended containers,
2. nodes, which are arbitrary objects in a graph,
3. edges, which are connections between objects in a graph, and
4. edge values, which are values assigned to connections in a graph.

$$\underbrace{\texttt{flighttime}}_{\texttt{graph}}[\overbrace{\underbrace{\texttt{Berlin}}_{\texttt{node}} : \underbrace{\texttt{London}}_{\texttt{node}}}^{\texttt{edge}}] = \underbrace{3900}_{\texttt{value}}$$

This documentation page gives an overview of the most important aspects. The complete interface of `GraphI` is defined and documented by *`Graph`*.

1.1 Creating Graphs and adding Nodes

You can create graphs empty, via cloning, from nodes or with nodes, edges and values. For many use-cases, it is simplest to start with a set of nodes:

```
from graphi import graph

planets = graph("Earth", "Jupiter", "Mars", "Pluto")
```

Once you have a graph, it works similar to a `set` for nodes. You can simply *`add()`* and *`discard()`* nodes:

```
planets.add("Venus")
planets.add("Mercury")
planets.discard("Pluto")
```

1.2 Working with Edges and Values

To really make use of a graph, you will want to add edges and give them values. Simply pick a connection *from* a node *to* a node and assign it a value:

```
# store the average distance between planets
planets["Earth":"Venus"] = 41400000
```

An edge is always of the form `start:end`, but values can be of arbitrary type. For example, you can easily add multiple values for a single edge using containers:

```
# add multiple values as an implicit tuple
planets["Earth":"Venus"] = 41400000, 258000000
# add multiple values as an explicit, mutable list
planets["Earth":"Mars"] = [78000000, 378000000]
```

The `:`-syntax of edges is not just pretty - it ensures that you never, ever accidentally mix up nodes and edges. This allows you to safely use the same `graph[item]` interface for nodes and edges.

If you need to define an edge outside of graph accesses, explicitly use *Edge*:

```
from graphi import Edge

if Edge["Venus":"Earth"] in planets:
    print("Wait, isn't there a pattern for this?")
```

1.3 Graphs as Python Containers

`GraphI` is all about letting you handle graphs with well-known interfaces. A graph is a container indexed by either nodes or edges:

```
print(planets["Venus":"Earth"])
del planets["Jupiter"]
```

Even though it contains nodes, edges and values, it presents its nodes first - similar to keys in a `dict`. However, you can efficiently access its various elements via views:

```
print("My father only told me about %d of our planets." % len(planets))
print("But I looked up %d distances between planets:" % len(planets.edges())
for planet_a, planet_b, distances in planets.items():
    print("  %s to %s: %s" % (planet_a, planet_b, '-'.join(distances)))
```

CHAPTER 2

Common Graph Operations

Many common graph operations map to simple operators in *graphi*. Unless parameters are needed, builtin operations usually suffice. For example, the outdegree of a node is simply its number of outgoing edges, i.e.

```
out_degree = len(graph[node])
```

in a directed graph. Since *graphi* makes heavy use of data views (instead of copies), this has optimal performance.

2.1 Pythonic Graph Operations

2.1.1 Nodes of a graph

Graphs behave like a `set` with regard to *nodes*. Note that removing a *node* also invalidates all its *edges* and their *values*.

graph[a] = True
graph.add(a)
graph.discard(a)
Safely add or remove a *node* `a` from `graph`.

del graph[a]
Remove a *node* `a` from `graph`.

a in graph
Whether there is a *node* `a` in `graph`.

list(graph)
iter(graph)
for a in graph:
List/iterate/traverse all *nodes* in `graph` .

len(graph)
The number of *nodes* in the graph.

2.1.2 Edges and values of a graph

Graphs special-case *edges*: an *edge* is a secondary key, being the value to *nodes* and the key to *edge values*.

Edge[a:b] in graph
Whether there is an *edge* from *node* `a` to *node* `b` in `graph`.

Loop[a] in graph
Edge[a:a] in graph
Whether there is a *loop* from *node* `a` to itself in `graph`.

list(graph[a])
iter(graph[a])
for b in graph[a]:
List/iterate/loop all *nodes* for which there is an edge from *node* `a`, i.e. its *neighbours*.

len(graph[a])
The number of outgoing *edges* of *node* `a`, i.e. its *outdegree*.

2.1.3 Edge values of a graph

Graphs behave similar to a `dict`, tying *values* to *edges*. Note that removing a *node* also invalidates all its *edges* and their *values*.

graph[a:b] = w
graph[Edge[a:b]] = w
Add an *edge* from *node* `a` to *node* `b` with *value* `w`.

2.2 Pythonic Graph Types

By default, every graph is a weighted, directed graph - *edges* are oriented from start to end *node* and have one *edge value*. However, other graph types can be created with standard language features.

graph[a:b] = True
Add an *edge* from *node* `a` to *node* `b` with the primitive *value* `True`.

This creates an unweighted graph edge.

graph[a:b] = [w1, w2, w3, ...]
graph[a:b] = w1, w2, w3, ...
Add an *edge* from *node* `a` to *node* `b` with multiple *values* `w1, w2, w3, ...`.

This creates a multigraph edge.

CHAPTER 3

Graph Syntax

Graphs use both key and slice notation to refer to *nodes* and *edges*, respectively. This works for both assignment and lookup of the respective value.

3.1 Nodes

A *node* is written directly. Its value is the adjacency associated with the node in the graph, i.e. a mapping to all *neighbours* and the respective *edge value*.

$$\underbrace{\texttt{flighttime}}_{\texttt{graph}}[\underbrace{\texttt{Berlin}}_{\texttt{node}}] = \overbrace{\{\underbrace{\texttt{London}}_{\texttt{node}} : \underbrace{3900}_{\texttt{value}}, ...\}}^{\texttt{adjacency}}$$

3.2 Edges

An *edge* is written using slice notation. Its value is the *edge value* associated with the edge in the graph.

$$\underbrace{\texttt{flighttime}}_{\texttt{graph}}\overbrace{[\underbrace{\texttt{Berlin}}_{\texttt{node}} : \underbrace{\texttt{London}}_{\texttt{node}}]}^{\texttt{edge}} = \underbrace{3900}_{\texttt{value}}$$

CHAPTER 4

Glossary

loop An edge from a node to itself. Counts as both an ingoing *and* outgoing edge for *outdegree*, *indegree* and *degree*.

indegree The number of ingoing edges of a node. If a node has a *loop*, it also counts as an ingoing edge.

The number of nodes to which a node is a *neighbour*.

outdegree The number of outgoing edges of a node. If a node has a *loop*, it also counts as an outgoing edge.

The number of *neighbours* of a node.

degree The number of ingoing and outgoing edges of a node. If a node has a *loop*, it counts as both an ingoing and outgoing edge.

The *degree* of a node is the sum of its *indegree* and *outdegree*.

graph A collection of *nodes*, *edges* between them and possibly *values* associated with any *edges*.

node A regular object in a *graph*.

edge

arrow A connection between two *nodes* in a *graph*.

edge value

weight The value associated with an *edge* in a *graph*.

neighbour A *node* with an *edge* from a specific node. Given an edge `a:b`, `b` is a *neighbour* of `a`.

The number of neighbours is the *outdegree*.

CHAPTER 5

graphi Changelog

5.1 prerelease 2017-??-??

Notes Added operator interface and implementations

Added graph input/output

Major Changes Added `graph[item] = True`, which is equal to `graph.add(item)`. Deprecates both `graph[node] = node` and `graph[node] = None`.

New Features Operator interface allowing graphs to provide optimized implementations

Added operators:

- `neighbours(graph, node, ..)`
- `density(graph)`

Added input/output:

- csv

Minor Changes

Graphs explicitly define `bool(graph)`. This was previously implicitly available as `bool` falls back to `__len__`.

5.2 0.2.0 2017-07-31

Notes Definition of primary interface, algorithms (`Graph.neighbours`) will be revised

New Features Added AdjacencyGraph

Major Changes Defined graph container interface

Minor Changes Added documentation

CHAPTER 6

graphi

6.1 graphi package

`graphi.graph`

Default graph type implementation

An implementation of the *Graph* interface, suitable for most purposes. Support of all graph interfaces for both reading and writing is provided. The implementation is adequate for most use-cases, and provides a balance of complexity, performance and storage.

See The corresponding class *AdjacencyGraph* for details.

alias of `AdjacencyGraph`

`graphi.GraphABC`

Graph abstract base class for type checks and virtual subclasses

The ABC is primarily needed for two cases:

- Type checking to find graph classes via `isinstance()`, as in `isinstance(candidate, GraphABC)`.
- Actual or virtual subclasses acting as implementations of the *graphi* interface for type checks.

See The corresponding class *Graph* for details.

alias of `Graph`

6.1.1 Subpackages

graphi.graph_io package

Submodules

graphi.graph_io.csv module

Utilities for loading and storing Graphs as csv

CSV Graph Format

The CSV graph format uses an optional header to define nodes, and a body storing the adjacency matrix. By default, a graph with `n` nodes is stored as a matrix literal of `n` columns and `n+1` rows:

```
a  b  c  d
0  2  1  0
2  0  3  2
1  4  0  0
0  1  3  0
```

Separators and formatting are handled by the csv `Dialect`. Value conversion and interpretation is handled by the appropriate reader/writer.

Reading Graphs

Graphs can be read using *graph_reader()* from iterables of CSV lines, such as files or str.splitlines. The csv itself is parsed using a `csv.reader()`, which allows setting the CSV dialect.

```
from graphi.graph_io import csv

literal = """\
a, b, c
0, 2, 3
2, 0, 2
1, 2, 0
"""

graph = csv.graph_reader(
    literal.splitlines(),
    skipinitialspace=True,
)
for nodes in graph:
    print(repr(node), "=>", graph[nodes])
```

class `graphi.graph_io.csv.`**`DistanceMatrixLiteral`**

Bases: `csv.Dialect`

CSV dialect for a Graph Matrix Literal, suitable for numeric data and string literals

A graph with alphabetic node names and numeric values would look like this:

```
a    b    c
0    2  1.3
```

```
 2   0  .5
16  .5   1
```

delimiter = ' '
no explicit delimeter between fields

doublequote = False

escapechar = '\\'
use regular escaping

lineterminator = '\n'

quotechar = '"'
string values are written as "foo", multi-values as '1,2,3'

quoting = 0

skipinitialspace = True
allow for alignment with arbitrary whitespace

exception `graphi.graph_io.csv.`**`ParserError`**(*error*, *row*, *column=None*)
Bases: `exceptions.Exception`

Error during parsing of a graph from a csv

`graphi.graph_io.csv.`**`graph_reader`**(*iterable*, *nodes_header=True*, *literal_type=<function stripped_literal>*, *valid_edge=<type 'bool'>*, *undirected=False*, **args*, ***kwargs*)
Load a graph from files or iterables

Parameters

- **`iterable`** – an iterable yielding lines of CSV, such as an open file
- **`nodes_header`** – whether and how to interpret a header specifying nodes
- **`literal_type`** – type callable to evaluate literals
- **`valid_edge`** – callable to test whether an edge should be inserted
- **`undirected`** – whether to mirror the underlying matrix

The `iterable` argument can be any object that returns a line of input for each iteration step, such as a file object or a list of strings.

Nodes are created depending on the value of `nodes_header`:

`False` Nodes are numbered `1` to `len(iterable[0])`. Elements in the first line of `iterable` are *not* consumed by this.

iterable Nodes are read from `node_header`.

`True` Nodes are taken as the elements of the first line of `iterable`. The first line is consumed by this, and not considered as containing graph edges. Nodes are read plainly of type :py:class:`str`, not using `literal_type`.

callable Like `True`, but nodes are not taken as plain `str()` but individually interpreted via `node_header(element)`.

The CSV is interpreted as a matrix, where the row marks the origin of an edge and the column marks the destination. Thus, loops are specified on the diagonal, while an asymmetric matrix creates different edge values for opposite directions. For an `undirected` graph, the matrix is automatically treated as symmetric. Trailing empty lines may be removed.

In the following example, the edges `a:b` and `a:c` are symmetric and there are no edges or self-loops `a:a` or `b:b`. In contrast, `b:c` is 3 whereas `c:b` is `4`, and there is a self-loop `c:c`. The node `d` only has an ingoing edge `b:d`, but no outgoing edges:

```
a   b   c   d
0   2   1   0
2   0   3   2
1   4   1   0
```

If `undirected` evaluates to `True`, the upper right corner is mirrored to the lower left. Note that the diagonal *must* be provided. The following matrices give the same output if `symmetric` is `True`:

```
a   b   c       a   b   c       a   b   c
0   2   1       0   2   1       0   2   1
2   0   3           0   3       5   0   3
1   4   1               1       7       1
```

Each value is evaluated and filtered by `literal_type` and `valid_edge`:

`graphi.graph_io.csv.`**`literal_type`**(*literal*) → object

Fields read from the csv are passed to *literal_type* directly as the sole argument. The return value is considered as final, and inserted into the graph without further conversions.

`graphi.graph_io.csv.`**`valid_edge`**(*object*) → bool

Similarly, *valid_edge* is called on the result of *literal_type*. The default is `bool()`, which should work for most data types.

The default for `literal_type` is capable of handling regular python literals, e.g. `int`, `float` and `str`. In combination with *valid_edge*, any literal of non-True values signifies a missing edge: *None*, *False*, *0* etc.

See All `*args` and `**kwargs` are passed on directly to `csv.reader` for extracting lines.

`graphi.graph_io.csv.`**`stripped_literal`**(*literal*)

evaluate literals, ignoring leading/trailing whitespace

This function is capable of handling all literals supported by `ast.literal_eval()`, even if they are surrounded by whitespace.

graphi.operators package

`graphi.operators.`**`neighbours`**(*graph*, **args*, ***kwargs*)

Yield all nodes to which there is an outgoing edge from `node` in `graph`

Parameters

- **`graph`** (*`Graph`*) – graph in which to search for edges
- **`node`** – node from which edges originate.
- **`maximum_distance`** – maximum distance to other nodes

Returns iterator of neighbour nodes

Raises ***`NodeError`*** – if `node` is not in the graph

When `maximum_distance` is not `None`, it is the maximum allowed edge value. This is interpreted using the <= operator as `graph[edge] <= distance`.

If there is a valid edge `graph[node:node] <= distance`, then `node` is part of its own neighbourhood.

Note The order of neighbours is arbitrary.

`graphi.operators.density(graph, *args, **kwargs)`

Return the density of the graph, i.e. the connectedness of its nodes

Parameters **graph** (*Graph*) – graph for which to calculate density

Raises **ValueError** – if `graph` has no nodes

The density is the ratio of actual edge count versus the maximum, non-looping edge count.

A graph without edges has a density of `0`, whereas a complete graph has a density of `1`. A graph with a *loop* may have a density bigger than `1`. The density is undefined for a graph less than two nodes, and raises `ValueError`.

Submodules

graphi.operators.interface module

`graphi.operators.interface.graph_operator(prefix='graphi')`

Implement a callable as a graph operator

Parameters **prefix** – identifier to prepend to special method names

Adds the operator lookup and fallback procedure to allow *Graph* subclasses to implement optimized algorithms. For example, a graph storing edges in a sorted data structure may prematurely end a search for neighbours given a maximum distance.

A graph can influence the evaluation of a graph operator by providing an attribute named `__<prefix>_<operator name>__`, e.g. `__graphi_neighbours__` for an operator `neighbours`. If this is the case, the attribute is called with the provided arguments as a replacement for the operator implementation.

`graphi.operators.interface.operator(graph, *args, **kwargs)`

The generic implementation of a graph operator.

`graph.__graphi_operator__(*args, **kwargs)`

The optimized implementation of a graph operator.

There are three special conditions to this procedure:

attribute is `None` The graph does not support the operation.

Attempting the operation on the graph raises `TypeError`.

attribute is `NotImplemented` The graph does not overwrite the operation. The operator implementation is always used.

calling the attribute returns `NotImplemented` The graph does not overwrite the operation for the specific parameters. The operator implementation is used.

The name of an operator is taken from `operator.__name__` or `operator.__class__.__name__`.

graphi.types package

Submodules

graphi.types.adjacency_graph module

class `graphi.types.adjacency_graph.AdjacencyGraph(*source, **kwargs)`

Bases: *graphi.abc.Graph*

Graph storing edge distances via adjacency lists

Parameters

- **source** – adjacency information
- **undirected** – whether the graph enforces symmetry

This graph provides optimal performance for random, direct access to nodes and edges. As it stores individual nodes and edges, it is optimal in both space and time for sparse graphs.

However, ordering of `nodes()`, *edges()* and *values()* is arbitrary. The expected complexity for searches is the worst case of O(len(`nodes()`) = n) and O(len(*edges()*) -> n^2), respectively.

clear()

edges()

update(*other*)

values()

class graphi.types.adjacency_graph.**EdgeView**(*graph*)
Bases: *graphi.abc.EdgeView*

View on the edges in a graph

class graphi.types.adjacency_graph.**ValueView**(*graph*)
Bases: *graphi.abc.ValueView*

View on the values of edges in a graph

6.1.2 Submodules

graphi.abc module

class graphi.abc.**AdjacencyList**
Bases: `dict`, `_abcoll.MutableMapping`

Edge values of nodes to a node in a graph

This represents edges in a `graph` originating from `node` as a mapping to their values. For example, the edge `graph[a:b] = c` corresponds to `adjacency[b] = c` for node `a`.

exception graphi.abc.**AdjacencyListTypeError**(*item*)
Bases: `exceptions.TypeError`

AdjacencyList set with an incorrect type

exception graphi.abc.**EdgeError**
Bases: `exceptions.Exception`

Graph edge not found

class graphi.abc.**EdgeView**(*graph*)
Bases: *graphi.abc.GraphView*

View on the edges in a graph

class graphi.abc.**Graph**(**source*, ***kwargs*)
Bases: `_abcoll.Container`

Abstract Base Class for graphs representing values of edges between nodes

A *Graph* is a container for primitive nodes and edges. There are three types of elements handled by a graph:

1. primitive *nodes*,
2. slice-like *edges* as pairs of nodes, and
3. primitive *edge values*.

Both nodes and edge values are conceptually similar to keys and values of `dict`. However, the concept of node pairs as edges adds additional functionality. The fixed relation between arbitrary nodes `a, b` and the directed pair `a:b` creates two value-type layers:

1. each *node* is mapped to all its outgoing edges,
2. each *edge* is mapped to the respective edge value.

In short, `graph[a]` provides a collection of edges originating at `a`, while `graph[a:b]` provides the specific edge value from `a` to `b`.

Note: Many interfaces return the rich *`Edge`* type for its added usability. To access an edge value, using `slice` such as `graph[a:b]` is sufficient, however.

Similar to `Mappings`, nodes are the primary keys of a *`Graph`*. As a result, the container interfaces, such as `iter` and `len`, operate on nodes. In general, nodes can be of arbitrary type as long as they are hashable.

By default, edges in a *`Graph`* are directed and unique: The edges represented by `graph[a:b]` and `graph[b:a]` are separate with opposite direction. Each edge is unique, i.e. there is only one edge `graph[a:b]`. A loop is represented by the edge `graph[a:a]`. The edge entities stored in the graph may be arbitrary objects.

As such, the interface of *`Graph`* defaults to describing a directed graph. However, other types of graph can be expressed as well. These generally do not form separate types in term of implementation.

Multigraphs allow for multiple edges between pairs of nodes. In this case, all edge values are containers (such as `list` or `set`) of arbitrary size. Whether a *`Graph`* is a graph of containers or a multigraph depends on the context.

Undirected Graphs do not distinguish between `graph[a:b]` and `graph[b:a]`. This can be enforced by symmetry of edge values, which guarantees that `graph[a:b] == graph[b:a]` always applies.

`g.undirected`

Indicates whether *`Graph`* `g` is guaranteed to be undirected, having only symmetric edge values. If `True`, `g[a:b] is g[b:a]` for any nodes `a` and `b` in `g`; the graph enforces this, e.g. `g[a:b] = c` implies `g[b:a] = c`. If `False`, symmetric edges are allowed but not enforced.

Read-only unless explicitly indicated otherwise.

There are several ways to initialise a new graph; their main difference is which element types are left empty.

`Graph()`

Create a new empty graph. No nodes, edges or values are filled in.

`Graph(graph)`

Create a new graph with all nodes, edges and values of `graph`. The resulting graph is a shallow copy of `graph` - the identity of elements is preserved.

`Graph(a, b, c, ...)`
`Graph([a, b, c, ...])`
`Graph({a, b, c, ...})`
`Graph(<iterable for a, b, c, ...>)`

Create a new graph with nodes `a`, `b`, `c`, `d`, and so on. No edges or values are created explicitly.

`Graph({a: {b: ab_edge, c: ...}, b: {a: ab_edge, ...}})`

Graph({a: AdjacencyList({b: ab_edge, c: ...}), b: AdjacencyList(...), ...})
Create a new graph with nodes `a`, `b`, `c`, and so on. Initialize edges to `graph[a:b] = ab_edge`, `graph[b:a] = ba_edge`, and so on.

Note: If only a single argument is provided, graph and mapping initialization is preferred over iterable initialisation. To initialize a graph with a graph or mapping as the sole node, wrap it in an iterable, e.g. `Graph([graph])`.

All implementations of this ABC guarantee the following operators:

bool(g)
Whether there are any nodes in the graph `g`.

len(g)
Return the number of nodes in the graph `g`.

g[a:b]
Return the value of the edge between nodes `a` and `b`. Raises *EdgeError* if no edge is defined for the nodes. Undirected graphs guarantee `g[a:b] == g[b:a]`.

g[a:b] = value
Set the value of the edge between nodes `a` and `b` to `value` for graph `g`.

del g[a:b]
Remove the edge and value between nodes `a` and `b` from `g`. Raises *EdgeError* if the edge is not in the graph.

g[a]
Return the edges between nodes `a` and any other node as an *AdjacencyList* corresponding to `{b: ab_edge, c: ac_edge, ...}`. Raises *NodeError* if `a` is not in `g`.

g[a] = True
g.add(a)
Add the node `a` to graph `g` if it does not exist. Do not add, remove or modify existing edges. Graphs for which edges are computed, not set, may create them implicitly.

Changed in version 0.3.0: Added `g[a] = True`, deprecated `g[a] = a` and `g[a] = None`.

g[a] = {}
g[a] = AdjacencyList()
Add the node `a` to graph `g` if it does not exist. Remove any existing edges originating at `a` from graph `g`.

g[a] = {b: ab_edge, c: ac_edge, ...}
g[a] = AdjacencyList(b=ab_edge, c=c_edge)
Add the node `a` to graph `g` if it does not exist. Set the value of the edge between nodes `a` and `b` to `ab_edge`, between `a` and `c` to `ac_edge`, and so on. Remove any other edge from `a`. Raises *NodeError* if any of `b`, `c`, etc. are not in `g`.

del g[a]
Remove the node `a` and all its edges from `g`. Raises *NodeError* if the node is not in the graph.

a in g
Return `True` if `g` has a node `a`, else `False`.

Edge[a:b] in g
Edge(a, b) in g
Return `True` if `g` has an edge from node `a` to `b`, else `False`.

iter(g)
Return an iterator over the nodes in `g`.

In addition, several methods are provided. While methods and operators for retrieving data must be implemented by all subclasses, methods for modifying data may not be applicable to certain graphs.

add(*item*)
Safely add a node or edge to the graph, without modifying existing edges

If a *node* is not part of the graph, it is added without any explicit edges. If a *edge* is not part of the graph, its value is set to `True`.

Note: Graphs which compute edges may implicitly create new edges if `node` is new to the graph.

clear()
Remove all elements from this graph

copy()
Return a shallow copy of this graph

discard(*item*)
Remove a node or edge from the graph if it is a member

Parameters **item** – node or edge to discard from the graph

edges()
Return a new view of the graph's edges

Returns view of the graph's edges

Return type *EdgeView*

get(*item*, *default=None*)
Return the value for node or edge `item` if it is in the graph, else default. If `default` is not given, it defaults to `None`, so that this method never raises a *NodeError* or *EdgeError*.

Parameters

- **item** – node or edge to look up in the graph
- **default** – default to return if `item` is not in the graph

items()
Return a new view of the graph's edges and their values

Returns view of the graph's edges and their values

Return type *ItemView*

nodes()
Return a new view of the graph's nodes

Returns view of the graph's nodes

Return type *NodeView*

undirected = False
whether this graph is undirected, having only symmetric edges

update(*other*)
Update the graph with the nodes, edges and values from `other`, overwriting existing elements.

Parameters **other** (*Graph* or *ItemView*) – graph or items from which to pull elements

values()
Return a new view of the values of the graph's edges

Returns view of the values of the graph's edges

Return type *ValueView*

class `graphi.abc.`**`GraphView`**(*graph*)

Bases: `_abcoll.Sized`

Dynamic view on the content of a *Graph*

View objects represent a portion of the content of a graph. A view allows to work with its scope without copying the viewed content. It is dynamic, meaning that any changes to the graph are reflected by the view.

Each view works only on its respective portion of the graph. For example, `edge in nodeview` will always return `False`.

`len(graphview)`
Return the number of nodes, node pairs or edges in the graph.

`x in graphview`
Return `True` if x is a node, node pair or edge of the graph.

`iter(graphview)`
Return an iterator over the nodes, node pairs or edges in the graph.

Each view strictly defines the use of nodes, edges or values. As such, edges are safely represented as a tuple of start and end node.

`undirected`

class `graphi.abc.`**`ItemView`**(*graph*)

Bases: *graphi.abc.GraphView*

View on the edges and values in a graph

Represents edges and their value as a `tuple` of `(tail, head, value)`. For example, the edge `graph[a:b] = c` corresponds to the item `(a, b, c)`.

exception `graphi.abc.`**`NodeError`**

Bases: `exceptions.Exception`

Graph node not found

class `graphi.abc.`**`NodeView`**(*graph*)

Bases: *graphi.abc.GraphView*

View on the nodes of a graph

class `graphi.abc.`**`ValueView`**(*graph*)

Bases: *graphi.abc.GraphView*

View on the values of edges in a graph

graphi.edge module

class `graphi.edge.`**`Edge`**(*start*, *stop*, *step=None*)

Bases: `object`

An edge in a graph as a pair of nodes

Parameters

- **`start`** – the start or tail of an edge
- **`stop`** – the stop or head of an edge

- **step** – currently unused

This is a verbose interface for creating edges between nodes for use in a graph. It allows using slice notation independent of a graph:

```
>>> atb = Edge[a:b]
>>> a2b = Edge(a, b)
>>> graph[a2b] = 1337
>>> graph[a:b] == graph[atb] == graph[a2b] == graph[Edge[a:b]] == graph[Edge(a,␣
↪b)]
True
```

A *Edge* can also be used for explicit containment tests:

```
>>> Edge[a:b] in graph
True
```

In addition to their slice-like nature, *Edge* is iterable and indexable. This allows for easy unpacking:

```
>>> edge = Edge[a:b]
>>> tail, head = edge
```

Note: This class creates a representation of an edge as a connection between nodes. Edge *values* can be arbitrary objects.

Warning: Even though *Edge* behaves like a `slice` in graphs, builtin containers such as `list` cannot make use of an *Edge*.

start

stop

class `graphi.edge.`**EdgeMeta**

Bases: `type`

Metaclass for *Edge* to support `Edge[a:b]`

class `graphi.edge.`**Loop**(*start*, *stop=None*, *step=None*)

Bases: *graphi.edge.Edge*

An edge in a graph from a node to itself

Parameters

- **start** – the start or tail of a loop
- **stop** – optional stop or head of a loop, same as start
- **step** – currently unused

Raises ValueError – if `stop` is given but not equal to `start`

`GraphI` is a lightweight graph library - it is suitable to model networks, connections and other relationships. Compared to other graph libraries, `GraphI` aims for being as pythonic as possible. If you are comfortable using `list`, `dict` or other types, `GraphI` is intuitive and straight-forward to use.

```
# create a graph with initial nodes
from graphi import graph
```

```
airports = graph("New York", "Rio", "Tokyo")
# add connections between nodes
airports["New York":"Rio"] = timedelta(hours=9, minutes=50)
airports["New York":"Tokyo"] = timedelta(hours=13, minutes=55)
```

At its heart, `GraphI` is built to integrate with Python's data model. It natively works with primitives, iterables, mappings and whatever you need. For example, creating a multigraph is as simple as using multiple edge values:

```
# add multiple connections between nodes -> Multigraph
airports["Rio":"Tokyo"] = timedelta(days=1, hours=2), timedelta(days=1, hours=3)
```

By design, `GraphI` is primarily optimized for general convenience over specific brute force performance. It heavily exploits lazy iteration, data views and other modern python paradigms under the hood. This allows the use of common operations without loss of performance:

```
# get number of outgoing edges of nodes -> outdegree
outgoing_flights = {city: len(airports[city]) for city in airports}
```

With its general-purpose design, `GraphI` makes no assumptions about your data. You are free to use whatever is needed to solve your problem, not please data structure.

CHAPTER 7

Frequently Asked Questions

Yet another graph library? The goal of `GraphI` is not to be another graph library, but to provide an intuitive way to work with graphs. Working with complex graphs should be as easy *for you* as working with any other primitive type.

What is this thing you call ABC? `GraphI` does not just provide graph *implementations*, but also an efficient graph *interface*. This interface is defined by the *`graphi.abc`* abstract base classes.

Any custom graph implementation can be made a *virtual* subclass of these ABCs. This allows you to adopt graph implementations optimized for your use-case without changing your code.

Where are all the algorithms? First and foremost, `GraphI` is designed for you to *work on graph data* instead of pre-sliced storybook data. `GraphI` implements only algorithms that

1. are fundamental building blocks for advanced algorithms, and/or
2. benefit from knowledge of internal data structures.

At the moment, you can find basic operators in the *`graphi.operators`* module.

What about performance? At its core, `GraphI` uses Python's native, highly optimized data structures. For any non-trivial graph algorithm, the provided performance is more than sufficient.

From our experience, performance critical code is best run with PyPy. This will not just optimize isolated pieces, but the actual combination of your algorithm and `GraphI` as a whole.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Documentation built from graphi 0.2.0 at Nov 17, 2017.

Python Module Index

g

Index

Symbols

A

C

D

E

G

I

L

N

O

P

Q

S

U

V

W