PyGFA Documentation

Release 1.0a

Diego Lobba

Sep 03, 2017

Contents:

CHAPTER 1

pygfa

pygfa package

Subpackages

pygfa.algorithms package

Submodules

pygfa.algorithms.simple_paths module

A module rewritten using the simple_paths networkx module to provide a convenient and reusable way to specificy a custom iterator to use in the algorithm (using only algorithms for multigraphs)

The same documentation for networkx is valid using this algorithms.

pygfa.algorithms.simple_paths.**all_simple_paths**(*gfa_*, *source*, *target*, *selector*, *edges=False*, *keys=True*, *cutoff=None*)

Compute the all_simple_path algorithm as described in networkx, but return the edges keys if asked and use the given selector to obtain the nodes to consider.

Parameters

- **selector** – A function or a method used to select the nodes to consider, the selector MUST give back two values at least and three values considering the keys. So the selector must be a similar networkx edges selectors (at least in behavior).
- **edges** – If True return the edges key that connect each pair of nodes in the simple path, each data is given in the format *(node_to, edge_that_connect_previous_to_node_to)*, so source node and target node will be in the form *(node, None)*.
- **args** – Optional arguments to supply to selector.

pygfa.algorithms.traversal module

pygfa.algorithms.traversal.**dfs_edges**(*gfa_*, *selector*, *source=None*, *keys=False*, ***args*)
Custom dfs_edges to select custom edges while traversing.

Parameters **keys** – If set return the keys of the edges of the dfs tree.

Module contents

pygfa.dovetail_operations package

Submodules

pygfa.dovetail_operations.iterator module

Iterators used by the GFA graph. This iterators work considering only edges representing dovetails overlaps.

class pygfa.dovetail_operations.iterator.**DovetailIterator**
Bases: object

dovetails_iter(*nbunch=None*, *keys=False*, *data=False*)
Return an iterator on edges that describe dovetail overlaps with the given node.

Notes It seems that networkx edges_iter keeps track of edges already seen, so the edge (u,v) is in the results but edge (v,u) is not.

dovetails_linear_path_iter(*source*, *keys=False*)
Return an iterator over the linear path whose source node belongs to, starting from one end of the path to another.

Parameters **source** – One of the node in the linear path.

dovetails_linear_path_traverse_edges_iter(*source*, *keys=False*)
Traverse all nodes adjacent to source node where the right degree and left degree of each node is 1.

Parameters **source** – One of the node in the linear path. It doesn't matter if it's one of the end of the linear path.

Notes If the source node it's not one of the end node of the path, the result is not an iterator over the ordered node of the linear path, but an iterator where nodes are returned by their distance from the source node.

If the source node is an isolated node, then this method returns an empty list (no edge is found). Use dovetails_linear_path_traverse_nodes_iter instead.

Same code as _plain_bfs_dovetails_with_edges function.

dovetails_linear_path_traverse_nodes_iter(*source*)
Traverse all nodes adjacent to source node where the right degree and left degree of each node is 1.

Parameters **source** – One of the node in the linear path. It doesn't matter if it's one of the end of the linear path.

Notes If the source node it's not one of the end node of the path, the result is not an iterator over the ordered node of the linear path, but an iterator where nodes are returned by their distance from the source node.

The code is the same as networkx _plain_bfs.

dovetails_nbunch_iter(*nbunch=None*)
Return an iterator checking that the given nbunch nodes are in the graphs. Consider only nodes involved into a dovetail overlap.

dovetails_neighbors(*nbunch=None*)
Return a list of all the right and left segments of the given nodes.

dovetails_neighbors_iter(*nbunch=None*, *keys=False*, *data=False*)
Return an iterator over neighbors nodes considering all nodes in nbunch as source node.

Notes This method is used to check right and left links among sequences, so from_node is needed. If only to_node in neighborhood are need, consider using 'dovetails_neighbors'.

left(*nbunch=None*)
Return all the nodes connected to the left end of the given node sequence.

left_degree(*nbunch=None*)

left_degree_iter(*nbunch=None*)

left_end_iter(*nbunch=None*, *keys=False*, *data=False*)
Return an iterator over dovetail edges where left segment-end of the nodes ids given are taken into account in the overlap

right(*nbunch=None*)
Return all the nodes connected to the right end of the given node sequence.

right_degree(*nbunch=None*)

right_degree_iter(*nbunch=None*)

right_end_iter(*nbunch*, *keys=False*, *data=False*)
Return an iterator over dovetail edges where nodes id right-segment end is taken into account in the overlap

pygfa.dovetail_operations.linear_paths module

Module that contain operation to find linearh paths in a GFA graph.

pygfa.dovetail_operations.linear_paths.**dovetails_linear_path**(*gfa_*, *node_*, *keys=False*)
Return the oriented edges involved in a linear path where that contain the given node.

pygfa.dovetail_operations.linear_paths.**dovetails_linear_paths**(*gfa_*, *components=False*, *keys=False*)

pygfa.dovetail_operations.operations module

pygfa.dovetail_operations.operations.**dovetails_remove_dead_ends**(*gfa_*, *min_length*, *safe_remove=False*)
Remove all the nodes where its right degree and its left degree are the following (0,0), (1,0), (1,0) and the length of the sequence is less than the given length. The node to remove mustn't split its connected component in two.

Parameters

- **min_length** –
- **consider_sequence** – If set try to get the sequence length where length field is not defined.

- **safe_remove** – If set the operation doesn't remove nodes where is not possible to obtain the length value.

Note Using the right and left degree, only dovetails overlaps are considered.

pygfa.dovetail_operations.operations.**dovetails_remove_small_components**(*gfa_*, *min_length*)

Remove all the connected components where the sequences length is less than min_length.

Find all the connected components nodes, for each component obtain the sum of the sequences length. If length is less than the given length remove the connected component nodes.

Parameters **min_length** – An integer describing the required length to keep a connected component.

Note

When connected components are computed only dovetail overlaps

edges are considered.

pygfa.dovetail_operations.simple_paths module

pygfa.dovetail_operations.simple_paths.**dovetails_all_simple_paths**(*gfa_*, *source*, *target*, *edges=False*, *keys=False*, *cutoff=None*)

Module contents

pygfa.graph_element package

Subpackages

pygfa.graph_element.parser package

Submodules

pygfa.graph_element.parser.containment module

class pygfa.graph_element.parser.containment.**Containment**

Bases: *pygfa.graph_element.parser.line.Line*

PREDEFINED_OPTFIELDS = {'ID': 'Z', 'NM': 'i', 'RC': 'i'}

REQUIRED_FIELDS = {'from': 'lbl', 'to': 'lbl', 'from_orn': 'orn', 'pos': 'pos', 'to_orn': 'orn', 'overlap': 'cig'}

classmethod **from_string**(*string*)

Extract the containment fields from the string.

The string can contains the C character at the begin or can only contains the fields of the containment directly.

pygfa.graph_element.parser.edge module

class pygfa.graph_element.parser.edge.**Edge**
Bases: *pygfa.graph_element.parser.line.Line*

REQUIRED_FIELDS = {'beg1': 'pos2', 'alignment': 'aln', 'sid1': 'ref', 'sid2': 'ref', 'beg2': 'pos2', 'eid': 'oid', 'end1': 'p

classmethod from_string(*string*)
Extract the Edge fields from the string.

The string can contains the E character at the begin or can only contains the fields of the Edge directly.

pygfa.graph_element.parser.field_validator module

Field validation module to check each field string against GFA1 and GFA2 specification.

exception pygfa.graph_element.parser.field_validator.**FormatError**
Bases: Exception

Exception raised when a wrong type of object is given to the validator.

exception pygfa.graph_element.parser.field_validator.**InvalidFieldError**
Bases: Exception

Exception raised when an invalid field is provided.

exception pygfa.graph_element.parser.field_validator.**UnknownDataTypeError**
Bases: Exception

Exception raised when the datatype provided is not in the *DATASTRING_VALIDATION_REGEXP* dictionary.

pygfa.graph_element.parser.field_validator.**is_dazzler_trace**(*string*)

pygfa.graph_element.parser.field_validator.**is_gfa1_cigar**(*string*)
Check if the given string is a valid CIGAR string as defined in the GFA1 specification.

pygfa.graph_element.parser.field_validator.**is_gfa2_cigar**(*string*)
Check if the given string is a valid CIGAR string as defined in the GFA2 specification.

pygfa.graph_element.parser.field_validator.**is_valid**(*string*, *datatype*)
Check if the string respects the datatype.

Parameters **datatype** – The type of data corresponding to the string.

Returns True if the string respect the type defined by the datatype.

Raises

- ***UnknownDataTypeError*** – If the datatype is not presents in *DATASTRING_VALIDATION_REGEXP*.
- **UnknownFormatError** – If string is not python string.

TODO Fix exception reference in the documentation.

pygfa.graph_element.parser.field_validator.**validate**(*string*, *datatype*)
Return a value from the given string with the type closer to the one it's represented.

pygfa.graph_element.parser.fragment module

class pygfa.graph_element.parser.fragment.**Fragment**
Bases: *pygfa.graph_element.parser.line.Line*

REQUIRED_FIELDS = {'alignment': 'aln', 'fend': 'pos2', 'sid': 'id', 'external': 'ref', 'send': 'pos2', 'sbeg': 'pos2', 'fbeg'

classmethod from_string(*string*)
Extract the fragment fields from the string.

The string can contains the F character at the begin or can only contains the fields of the fragment directly.

pygfa.graph_element.parser.gap module

class pygfa.graph_element.parser.gap.**Gap**
Bases: *pygfa.graph_element.parser.line.Line*

REQUIRED_FIELDS = {'distance': 'int', 'gid': 'oid', 'sid1': 'ref', 'sid2': 'ref', 'variance': 'oint'}

classmethod from_string(*string*)
Extract the Gap fields from the string.

The string can contains the G character at the begin or can only contains the fields of the Gap directly.

pygfa.graph_element.parser.group module

class pygfa.graph_element.parser.group.**OGroup**
Bases: *pygfa.graph_element.parser.line.Line*

REQUIRED_FIELDS = {'references': 'rfs', 'oid': 'oid'}

classmethod from_string(*string*)
Extract the OGroup fields from the string.

The string can contains the O character at the begin or can just contains the fields of the OGroup directly.

class pygfa.graph_element.parser.group.**UGroup**
Bases: *pygfa.graph_element.parser.line.Line*

REQUIRED_FIELDS = {'ids': 'ids', 'uid': 'oid'}

classmethod from_string(*string*)
Extract the UGroup fields from the string.

The string can contains the U character at the begin or can only contains the fields of the UGroup directly.

pygfa.graph_element.parser.header module

class pygfa.graph_element.parser.header.**Header**
Bases: *pygfa.graph_element.parser.line.Line*

PREDEFINED_OPTFIELDS = {'TS': 'i', 'VN': 'Z'}

classmethod from_string(*string*)
Extract the header fields from the string.

The string can contains the H character at the begin or can only contains the fields of the header directly.

pygfa.graph_element.parser.line module

class pygfa.graph_element.parser.line.**Field**(*name*, *value*)
Bases: object

This class represent any required field.

The type of field is bound to the field name.

name

value

exception pygfa.graph_element.parser.line.**InvalidLineError**
Bases: Exception

Exception raised when making a Line object from a string. The number of fields gained by splittin the string must be equal to or great than the number of required field ecluding the optional first field indicating the type of the line.

class pygfa.graph_element.parser.line.**Line**(*line_type=None*)
Bases: object

A generic Line, it's unlikely that it will be directly instantiated (but could be done so). Its subclasses should be used instead.

It's possible to instatiate a Line to save a custom line in a gfa file.

PREDEFINED_OPTFIELDS = {}

REQUIRED_FIELDS = {}

add_field(*field*)
Add a field to the line.

It's possible to add a Field if an only if its name is in the *REQUIRED_FIELDS* dictionary. Otherwise the field will be considered as an optional field and an InvalidFieldError will be raised.

Parameters **field** – The field to add to the line

Raises ***InvalidFieldError*** – If a 'name' and a 'value' attributes are not found or the field has already been added.

Note If you want to add a Field for a custom Line object be sure to add its name to the REQUIRED_FIELDS dictionary for that particular Line subclass.

fields

classmethod from_string(*string*)

classmethod get_static_fields()

classmethod is_valid(*line_*)
Check if the line is valid.

Defining the method here allows to have automatically validated all the line of the specifications.

remove_field(*field*)
If the field is contained in the line it gets removed. Otherwise it does nothing, without raising any exception.

type

class pygfa.graph_element.parser.line.**OptField**(*name*, *value*, *field_type*)
Bases: *pygfa.graph_element.parser.line.Field*

An Optional field of the form *TAG:TYPE:VALUE*, where: TAG match [A-Za-z0-9][A-Za-z0-9] TYPE match [AiZfJHB]

classmethod from_string(*string*)
Create an OptField with a given string.

type

pygfa.graph_element.parser.line.**is_field**(*field*)
Check if the given object is a valid field

A field is valid if it has at least a name and a value attribute/property.

pygfa.graph_element.parser.line.**is_optfield**(*field*)
Check if the given object is an optfield

A field is an optfield if it's a field with name that match a given expression and its type is defined.

pygfa.graph_element.parser.link module

class pygfa.graph_element.parser.link.**Link**
Bases: *pygfa.graph_element.parser.line.Line*

PREDEFINED_OPTFIELDS = {'KC': 'i', 'NM': 'i', 'RC': 'i', 'MQ': 'i', 'ID': 'Z', 'FC': 'i'}

REQUIRED_FIELDS = {'from': 'lbl', 'to': 'lbl', 'from_orn': 'orn', 'overlap': 'cig', 'to_orn': 'orn'}

classmethod from_string(*string*)
Extract the link fields from the string.

The string can contains the L character at the begin or can just contains the fields of the link directly.

pygfa.graph_element.parser.path module

class pygfa.graph_element.parser.path.**Path**
Bases: *pygfa.graph_element.parser.line.Line*

PREDEFINED_OPTFIELDS = {}

REQUIRED_FIELDS = {'overlaps': 'cgs', 'path_name': 'lbl', 'seqs_names': 'lbs'}

classmethod from_string(*string*)
Extract the path fields from the string.

The string can contains the P character at the begin or can just contains the fields of the path directly.

pygfa.graph_element.parser.segment module

class pygfa.graph_element.parser.segment.**SegmentV1**
Bases: *pygfa.graph_element.parser.line.Line*

A GFA1 Segment line.

PREDEFINED_OPTFIELDS = {'KC': 'i', 'FC': 'i', 'RC': 'i', 'LN': 'i', 'UR': 'Z', 'SH': 'H'}

REQUIRED_FIELDS = {'name': 'lbl', 'sequence': 'seq'}

classmethod from_string(*string*)
Extract the segment fields from the string.

The string can contains the S character at the begin or can only contains the fields of the segment directly.

class pygfa.graph_element.parser.segment.**SegmentV2**
Bases: *pygfa.graph_element.parser.line.Line*

A GFA2 Segment line.

REQUIRED_FIELDS = {'slen': 'int', 'sequence': 'seq2', 'sid': 'id'}

classmethod **from_string**(*string*)
Extract the segment fields from the string.

The string can contains the S character at the begin or can only contains the fields of the segment directly.

pygfa.graph_element.parser.segment.**is_segmentv1**(*line_repr*)
Check wether a given gfa line string probably belongs to a Segment of the first GFA version.

Parameters **line_repr** – A string or a Line that is supposed to represent an S line.

pygfa.graph_element.parser.segment.**is_segmentv2**(*line_repr*)
Check wether a given string or line belongs to a Segment of the second GFA version.

Parameters **line_repr** – A string or a Line that is supposed to represent an S line.

Module contents

Submodules

pygfa.graph_element.edge module

class pygfa.graph_element.edge.**Edge**(*edge_id*, *from_node*, *from_orn*, *to_node*, *to_orn*, *from_positions*, *to_positions*, *alignment*, *distance=None*, *variance=None*, *opt_fields={}*, *is_dovetail=False*)
Bases: object

alignment

distance

eid

classmethod **from_line**(*line_*)

from_node

from_orn

from_positions

from_segment_end

is_dovetail

opt_fields

to_node

to_orn

to_positions

to_segment_end

variance

exception pygfa.graph_element.edge.**InvalidEdgeError**
Bases: Exception

pygfa.graph_element.edge.**is_edge**(*obj*)

pygfa.graph_element.node module

exception pygfa.graph_element.node.**InvalidNodeError**
Bases: Exception

class pygfa.graph_element.node.**Node**(*node_id*, *sequence*, *length*, *opt_fields={}*)
Bases: object

A Node object that abstract the GFA1 and GFA2 Sequence concepts.

GFA graphs will operate on Nodes, by adding them directly to their structures.

Node accepts elements (ids, sequences, lengths and so on) from the more tolerant of the two specification. So, a sequence will be accepted if and only if is a valid GFA2 sequence, since GFA2 sequence is more tolerant than GFA1 sequence.

classmethod from_line(*segment_line*)
Given a Segment Line construct a Node from it.

If segment_line is a GFA1 Segment segment_line then the sequence length taken into account will be the value of the optional field *LN* if specified in the line fields.

Parameters **segment_line** – A valid Segment Segment_line.

Raises **InvalidSegment_lineError** – If the given segment_line is not valid.

nid

opt_fields

sequence

slen

pygfa.graph_element.node.**is_node**(*obj*)
Check wheter the given object is a Node object.

Parameters **obj** – Any Python object.

Returns **True** If obj can be treated as a Node object.

pygfa.graph_element.subgraph module

exception pygfa.graph_element.subgraph.**InvalidSubgraphError**
Bases: Exception

class pygfa.graph_element.subgraph.**Subgraph**(*graph_id*, *elements*, *opt_fields={}*)
Bases: object

as_dict()
Turn the Subgraph into a dictionary,

Put all fields and the optional fields into a dictionary.

elements

classmethod from_line(*line_*)

is_path()

opt_fields

sub_id

pygfa.graph_element.subgraph.**is_subgraph**(*obj*)

Module contents

pygfa.serializer package

Submodules

pygfa.serializer.gfa1_serializer module

GFA1 Serializer for nodes, edge, Subgraphs and networkx graphs.

Can serialize either one of the object from the group mentioned before or from a dictionary with equivalent key.

exception pygfa.serializer.gfa1_serializer.**GFA1SerializationError**
Bases: Exception

pygfa.serializer.gfa1_serializer.**point_to_node**(*gfa_*, *node_id*)
Check if the given node_id point to a node in the gfa graph.

pygfa.serializer.gfa1_serializer.**serialize_edge**(*edge_*, *identifier='no identifier given.'*)
Converts to a GFA1 line the given edge.

Fragments and Gaps cannot be represented in GFA1 specification, so they are not serialized.

pygfa.serializer.gfa1_serializer.**serialize_gfa**(*gfa_*)
Serialize a GFA object into a GFA1 file.

pygfa.serializer.gfa1_serializer.**serialize_graph**(*graph*, *write_header=True*)
Serialize a networkx.MulitGraph object.

Parameters **graph** – A networkx.MultiGraph instance.

Write_header If set to True put a GFA1 header as first line.

pygfa.serializer.gfa1_serializer.**serialize_node**(*node_*, *identifier='no identifier given.'*)
Serialize to the GFA1 specification a Graph Element Node or a dictionary that has the same informations.

Parameters

- **node** – A Graph Element Node or a dictionary.
- **identifier** – If set help gaining useful debug information.

Return "" If the object cannot be serialized to GFA.

pygfa.serializer.gfa1_serializer.**serialize_subgraph**(*subgraph_*, *identifier='no identifier given.'*, *gfa_=None*)
Serialize a Subgraph object or an equivalent dictionary.

pygfa.serializer.gfa2_serializer module

GFA2 Serializer for nodes, edge, Subgraphs and networkx graphs.

Can serialize either one of the object from the group mentioned before or from a dictionary with equivalent key.

exception pygfa.serializer.gfa2_serializer.**GFA2SerializationError**
Bases: Exception

pygfa.serializer.gfa2_serializer.**are_elements_oriented**(*subgraph_elements*)
Check wheter all the elements of a subgraph have an orientation value *[+/-]*.

pygfa.serializer.gfa2_serializer.**serialize_edge**(*edge_*, *identifier='no identifier given.'*)
Converts to a GFA2 line the given edge.

pygfa.serializer.gfa2_serializer.**serialize_gfa**(*gfa_*)
Serialize a GFA object into a GFA2 file.

TODO: maybe process the header fields here

pygfa.serializer.gfa2_serializer.**serialize_graph**(*graph*, *write_header=True*)
Serialize a networkx.MultiGraph or a derivative object.

Parameters

- **graph** – A networkx.MultiGraph instance.
- **write_header** – If set to True put a GFA2 header as first line.

pygfa.serializer.gfa2_serializer.**serialize_node**(*node_*, *identifier='no identifier given.'*)
Serialize to the GFA2 specification a graph_element Node or a dictionary that has the same informations.

If sequence length is undefined (for example, after parsing a GFA1 Sequence line) a sequence length of 0 is automatically added in the serialization process.

Parameters **node** – A Graph Element Node or a dictionary

Identifier If set help gaining useful debug information.

Returns "" If the object cannot be serialized to GFA.

pygfa.serializer.gfa2_serializer.**serialize_subgraph**(*subgraph_*, *identifier='no identifier given.'*, *gfa_=None*)
Serialize a Subgraph object or an equivalent dictionary.

Returns "" If subgraph cannot be serialized.

TODO Check with *gfa* for OGroup in UGroup. See GFA2 spec.

pygfa.serializer.utils module

Module contents

Submodules

pygfa.gfa module

GFA representation through a networkx MulitGraph.

The dovetail operations are available thanks to the dovetail_operation.Iterator class, that considers only dovetail overlaps edges.

TODO

- Rewrite pprint method.

class pygfa.gfa.**Element**
Bases: object

Represent the types of graph a GFA graph object can have.

EDGE = 1

NODE = 0

SUBGRAPH = 2

class pygfa.gfa.**GFA**(*base_graph=None*)
Bases: *pygfa.dovetail_operations.iterator.DovetailIterator*

GFA will use a networkx MultiGraph as structure to contain the elements of the specification. GFA graphs directly accept only instances coming from the graph_elements package, but can contains any kind of data undirectly by accessing the *_graph* attribute.

add_edge(*new_edge*, *safe=False*)
Add a graph_element Edge or a networkx edge to the GFA graph using the edge id as key.

If its id is * or *None* the edge will be given a **virtual_id**, in either case the original edge id will be preserved as edge attribute.

All edge attributes will be stored as netwrorkx edge attributes and all the remainders optional field will be stored individually as edge data.

add_graph_element(*element*)
Add a graph element -Node, Edge or Subgraph- object to the graph.

add_node(*new_node*, *safe=False*)
Add a graph_element Node to the GFA graph using the node id as key.

Its sequence and sequence length will be individual attributes on the graph and all the remainders optional field will be stored individually as node data.

Parameters

- **new_node** – A graph_element.Node object or a string that can represent a node (such as the Segment line).
- **safe** – If set check if the given identifier has already been added to the graph, and in that case raise an exception

add_subgraph(*subgraph*, *safe=False*)
Add a Subgraph object to the graph.

The object is not altered in any way. A deepcopy of the object given is attached to the graph.

as_graph_element(*key*)
Given a key of an existing node, edge or subgraph, return its equivalent graph element object.

clear()
Clear all GFA object elements.

Call networkx *clear* method, reset the virtual id counter and delete all the subgraphs.

dovetails_subgraph(*nbunch=None*, *copy=True*)
Given a collection of nodes return a subgraph with the nodes given and all the edges between each pair of nodes. Only dovetails overlaps are considered.

dump(*gfa_version=1*, *out=None*)

edge(*identifier=None*)
GFA edge accessor.

•If *identifier* is *None* all the graph edges are returned.

•**If *identifier* is a tuple perform a search by nodes with** the tuple values as nodes id.

•**If *identifier* is a single defined value then perform** a search by edge key, where the edge key is the given value.

edges(***kwargs*)
Return all the edges in the graph.

edges_iter(*nbunch=None*, *data=False*, *keys=False*, *default=None*)
Interface to networx edges iterator.

classmethod from_file(*filepath*)
Parse the given file and return a GFA object.

from_string(*string*)
Add a GFA string to the graph once it has been converted.

TODO Maybe this could be used instead of checking for line type in the add_xxx methods...

get(*key*)
Return the element pointed by the specified key.

get_subgraph(*sub_key*)
Return a GFA subgraph from the parent graph.

Return a new GFA graph structure with the nodes, edges and subgraphs specified in the elements attributes of the subgraph object pointed by the id.

The returned GFA is *independent* from the original object.

Parameters **sub_key** – The id of a subgraph present in the GFA graph.

Returns None if the subgraph id doesn't exist.

nbunch_iter(*nbunch=None*)
Return an iterator of nodes contained in nbunch that are also in the graph.

Interface to the networkx method.

neighbors(*nid*)
Return all the nodes id of the nodes connected to the given node.

Return all the predecessors and successors of the given source node.

Params nid The id of the selected node

node(*identifier=None*)
An interface to access the node method of the netwrokx graph.

If *identifier* is *None* all the graph nodes are returned.

nodes(*data=False*, *with_sequence=False*)
Return a list of the nodes in the graph.

Parameters **with_sequence** – If set return only nodes with a *sequence* property.

nodes_iter(*data=False*, *with_sequence=False*)
Return an iterator over nodes in the graph.

Para with_sequence If set return only nodes with a sequence property.

pprint()
A basic pretty print function for nodes and edges.

remove_edge(*identifier*)
Remove an edge or all edges identified by an id or by a tuple with end node, respectively.

- **If *identifier* is a two elements tuple remove all the** all the edges between the two nodes.
- **If *identifier* is a three elements tuple remove the edge** specified by the third element of the tuple with end nodes given by the first two elements of the tuple itself.

•**If *identifier* is not a tuple, treat it as it should be** an edge id.

Raises ***InvalidEdgeError*** – If *identifier* is not in the cases described above.

remove_edges(*from_node*, *to_node*)
Remove all the direct edges between the two nodes given.

Call iteratively remove_edge (remove a not specified edge from *from_node* and *to_node*) for n-times where n is the number of edges between the given nodes, removing all the edges indeed.

remove_node(*nid*)
Remove a node with nid as its node id.

Edges containing nid as end node will be automatically deleted.

Parameters **nid** – The id belonging to the node to delete.

Raises ***InvalidNodeError*** – If *nid* doesn't point to any node.

remove_subgraph(*subgraph_id*)
Remove the Subgraph object identified by the given id.

search(*comparator*, *limit_type=None*)
Perform a query applying the comparator on each graph element.

search_on_edges(*comparator*)

search_on_nodes(*comparator*)

search_on_subgraph(*comparator*)

subgraph(*nbunch*, *copy=True*)
Given a bunch of nodes return a graph with all the given nodes and the edges between them.

The returne object is not a GFA Graph, but a MultiGraph. To create a new GFA graph, just use the GFA initializer an give the subgraph to it.

Interface to the networkx subgraph method. Given a collection of nodes return a subgraph with the nodes given and all the edges between each pair of nodes.

Parameters

- **nbunch** – The nodes.
- **copy** – If set to True return a copy of the subgraph.

subgraphs(*identifier=None*)
An interface to access to the subgraphs inside the GFA object.

If *identifier* is *None* all the graph Subgraph objects are returned.

subgraphs_iter(*data=False*)
Return an iterator over subgraphs elements in the GFA graph.

exception pygfa.gfa.**GFAError**
Bases: Exception

exception pygfa.gfa.**InvalidElementError**
Bases: Exception

exception pygfa.gfa.**InvalidSearchParameters**
Bases: Exception

pygfa.operations module

pygfa.operations.**nodes_connected_component**(*gfa_*, *nid*)
Return the connected component belonging to the given node.

Parameters **nid** – The id of the node to find the reachable nodes.

pygfa.operations.**nodes_connected_components**(*gfa_*)
Return a generator of sets with nodes of each weakly connected component in the graph.

Module contents

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

G

H

I

L

N

O

P

R