Another Python Graph Library Documentation

Release 0.8.1

Charanpal Dhanjal

August 09, 2014
## CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 News</td>
<td>3</td>
</tr>
<tr>
<td>2 Downloading</td>
<td>5</td>
</tr>
<tr>
<td>3 Installation</td>
<td>7</td>
</tr>
<tr>
<td>4 User Guide</td>
<td>9</td>
</tr>
<tr>
<td>4.1 Class Documentation</td>
<td>9</td>
</tr>
<tr>
<td>5 Support</td>
<td>69</td>
</tr>
<tr>
<td>6 Indices and tables</td>
<td>71</td>
</tr>
<tr>
<td>Index</td>
<td>73</td>
</tr>
</tbody>
</table>
Another Python Graph Library is a simple, fast and easy to use graph library. The main characteristics are as follows:

- Directed, undirected and multi-graphs using numpy and scipy matrices for fast linear algebra computations. The PySparseGraph, CsArray and SparseGraph classes can scale up to 1,000,000s of vertices and edges on a standard PC.
- Set operations including finding subgraphs, complements, unions, and intersections of graphs.
- Graph properties such as diameter, geodesic distance, degree distributions, eigenvector betweenness, and eigenvalues.
- Other algorithms: search, Floyd-Warshall, Dijkstra’s algorithm.
- Configuration Model, Erdos-Renyi, Small-World, Albert-Barabasi and Kronecker graph generation
- Write to Pajek and simple CSV files
- Unit tested using the Python unittest framework
NEWS

- 17/10/13 - Version 0.8 released.
- 29/12/12 - Version 0.7.3 released.
- 17/11/12 - Version 0.7.2 released.

See https://github.com/charanpald/APGL for the changelog.
Download for Windows, Linux or Mac OS using:

- Sourceforge here
- The Python Package Index (PyPI) here

To use this library, you must have Python, NumPy and SciPy. The code has been verified on Python 2.7.2, Numpy 1.5.1 and Scipy 0.9.0, but should work with other versions. The automatic testing routine requires Python 2.7 or later, or the unittest2 testing framework for Python 2.3-2.6.

The source code repository is available here for those that want the bleeding edge, or are interested in development.
Ensure that pip is installed, and then install apgl in the following way:

```bash
pip install apgl
```

If installing from source unzip the apgl-x.y.z.tar.gz file and then run setup.py as follows:

```bash
python setup.py install
```

In order to test the library (recommended), use the following commands in python

```python
import apgl
apgl.test()
```

and check that all tested pass.
A short introduction to the main features of the library is available in the PDF document “An Introduction to APGL”. This is the best way to learn the key features of APGL. In the meanwhile, here is small example of how to create a graph using the SparseGraph class which is based on scipy.sparse matrices.

```python
>>> from apgl.graph import GeneralVertexList, SparseGraph
>>> import numpy

>>> numVertices = 5
>>> graph = SparseGraph(numVertices)
>>> graph[0,1] = 1
>>> graph[0,2] = 3
>>> graph[1,2] = 0.1
>>> graph[3,4] = 2
>>> graph.setVertex(0, "abc")
>>> graph.setVertex(1, 123)
>>> graph.findConnectedComponents()
[[0, 1, 2], [3, 4]]
>>> graph.getWeightMatrix()
array([[ 0. , 1. , 3. , 0. , 0. ],
       [ 1. , 0. , 0.1, 0. , 0. ],
       [ 3. , 0.1, 0. , 0. , 0. ],
       [ 0. , 0. , 0. , 0. , 0. ],
       [ 0. , 0. , 0. , 2. , 0. ]])
>>> graph.degreeDistribution()
array([0, 2, 3])
>>> graph.neighbours(0)
array([2, 1], dtype=int32)
>>> print(graph)
SparseGraph: vertices 5, edges 4, undirected, vertex list type: GeneralVertexList
```

The `SparseGraph` is initialised as an undirected graph with 5 vertices of which the labels are stored in `GeneralVertexList`. Edges are added between vertices (0, 1), (0, 2), (1, 2) and (3, 4). Following, the first and second vertices (indexed by 0 and 1 respectively) are initialised with “abc” and 123 respectively, and we then compute some properties over the resulting graph.

To learn more consult the reference documentation:

## 4.1 Class Documentation

Warning: this class documentation is currently incomplete.
4.1.1 Graph Generators

These classes represent different ways to generate graphs in random and non-random ways.

**BarabasiAlbertGenerator**

A class to generate random graphs according to the Barabasi-Albert method. For example the following code generates a random graph with 10 vertices, starting with 2 initial vertices and adding 1 edge for each additional vertex.

```python
from apgl.graph import *
from apgl.generator.BarabasiAlbertGenerator import BarabasiAlbertGenerator

ell = 2
m = 1
graph = SparseGraph(VertexList(10, 1))
generator = BarabasiAlbertGenerator(ell, m)
graph = generator.generate(graph)
```

**Methods**

```python
class apgl.generator.BarabasiAlbertGenerator.BarabasiAlbertGenerator(ell, m)
    Create a random graph generator according to the Barabasi-Albert Model

    Parameters
    • ell (int) – the initial number of vertices.
    • m (int) – the number of edges to be added at each step

    generate (graph)
    Create a random graph using the input graph according to the Barabasi-Albert model. Note that the input graph is modified.

    Parameters graph (apgl.graph.AbstractMatrixGraph) – the empty input graph.

    Returns The modified input graph.

    setEll (ell)
    Parameters ell (int) – the initial number of vertices.

    setM (m)
    Parameters m (int) – the number of edges to be added at each step
```

**ConfigModelGenerator**

A class to generate random graphs according to the Configuration Model. In this model, one specifies a particular degree sequence and a random graph is generated which fits the sequence as closely as possible. The following code generates a random graph with 10 vertices such that the ith vertex has degree i.

```python
import numpy
from apgl.graph import *
from apgl.generator.ConfigModelGenerator import ConfigModelGenerator

degreeSequence = numpy.arange(10)
graph = SparseGraph(VertexList(10, 1))
```
generator = ConfigModelGenerator(degreeSequence)
graph = generator.generate(graph)

Methods

class apgl.generator.ConfigModelGenerator.ConfigModelGenerator(outDegSequence,
inDegreeSequence=None)

Create a ConfigModelGenerator object with the given degree sequence. If inDegreeSequence is None then we assume an undirected graph, otherwise it is directed. Note that the sum of the values in inDegreeSequence and outDegreeSequence should ideally be the same to avoid unconnected spokes.

Parameters

• outDegreeSequence (numpy.ndarray) – a vector of (out)degrees for each vertex in the graph.

• inDegreeSequence (numpy.ndarray) – a vector of in-degrees for each vertex in the graph or None for undirected graphs.

generate (graph, requireEmpty=True)

Create an Configuration Model graph. Note the the degree sequence(s) given in the constructor cannot be guaranteed. The algorithm randomly selects two free “spokes” and then tries to connect them. If two vertices are already connected the corresponding spokes are not used again. In the case that requireEmpty is False then a non-empty graph can be used and the given degree sequence(s) is(are) the difference(s) in degrees between the output graph and input one.

Parameters

• graph (apgl.graph.AbstractMatrixGraph) – a graph to populate with edges

• requireEmpty (bool) – if this is set to true then we require an empty graph.

Returns The modified input graph.

getInDegreeSequence ()

Returns A vector of integers corresponding to the (in)degree sequence.

getOutDegreeSequence ()

Returns A vector of integers corresponding to the (out)degree sequence.

setInDegreeSequence (inDegreeSequence)

Set the (in)degree sequence of this object.

Parameters inDegreeSequence (numpy.ndarray) – a vector of degrees for each vertex in the graph.

setOutDegreeSequence (outDegreeSequence)

Set the (out)degree sequence of this object.

Parameters outDegreeSequence (numpy.ndarray) – a vector of degrees for each vertex in the graph.

ErdosRenyiGenerator

A class to generate random graphs according to the Erdos-Renyi method. In this model, one specifies a probability p for the existence of an edge between two vertices. The following code generates a random graph with 10 vertices and edge probability 0.1.
import numpy
from apgl.graph import *
from apgl.generator.ErdosRenyiGenerator import ErdosRenyiGenerator

p = 0.1
graph = SparseGraph(VertexList(10, 1))
generator = ErdosRenyiGenerator(p)
graph = generator.generate(graph)

Methods

class apgl.generator.ErdosRenyiGenerator.ErdosRenyiGenerator (p, selfEdges=False)

Create an Erdos-Renyi generator with edge probability p. For all pairs of vertices in the graph, an edge exists with probability p. If selfEdges is False edges from a vertex to itself are not created.

Parameters

• p (float) – the probability of an edge
• selfEdges (bool) – whether to allow self edges

clusteringCoefficient ()

Returns the clustering coefficient for the generator.

generate (graph, requireEmpty=True)

Create an Erdos-Renyi graph from the given input graph.

Parameters

• graph (apgl.graph.AbstractMatrixGraph) – an empty graph to populate with edges
• requireEmpty (bool) – whether to allow non empty graphs.

Returns The modified input graph.

generate2 (graph, requireEmpty=True)

An alternative way of generating random edges which might work better than generate.

setP (p)

Parameters p (float) – the probability of an edge

setSelfEdges (selfEdges)

Parameters selfEdges (bool) – whether to allow self edges

KroneckerGenerator

A class to generate random graphs according to the Kronecker method. One specifies an input graph and the generated graph is grown using k successive kronecker multiplications of the adjacency matrix.

import numpy
from apgl.graph import *
from apgl.generator.KroneckerGenerator import KroneckerGenerator

initialGraph = SparseGraph(VertexList(5, 1))
initialGraph.addEdge(1, 2)
initialGraph.addEdge(2, 3)
for i in range(5):
    initialGraph.addEdge(i, i)

k = 2
generator = KroneckerGenerator(initialGraph, k)
graph = generator.generate()

Methods

class apgl.generator.KroneckerGenerator.KroneckerGenerator(initialGraph, k)

Initialise with a starting graph, and number of iterations k. Note that the starting graph must have self edges on every vertex. Only the adjacency matrix of the starting graph is used.

Parameters

• initialGraph (apgl.graph.AbstractMatrixGraph) – The initial graph to use.
• k (int) – The number of iterations.

generate()

Generate a Kronecker graph using the adjacency matrix of the input graph.

Returns

The generate graph as a SparseGraph object.

setK(k)

Set the number of iterations k.

Parameters

k (int) – The number of iterations.

SmallWorldGenerator

A class to generate random graphs according to the Small World method. The vertices are arranged on a two-dimensional lattice (a circle) and then edges are formed to their k closest neighbours in the lattice. Following, a random edge endpoint is selected and then rewired with probability p. The following code demonstrates the usage of this class.

import numpy
from apgl.graph import *
from apgl.generator.SmallWorldGenerator import SmallWorldGenerator

k = 2
p = 0.1
graph = SparseGraph(VertexList(10, 1))
generator = SmallWorldGenerator(p, k)
graph = generator.generate(graph)

Methods

class apgl.generator.SmallWorldGenerator.SmallWorldGenerator(p, k)

Create a small-world generator with k initial neighbours and a re-wiring probability p

Parameters

• p (float) – the probability of rewiring an edge.
• k (int) – the number of neighbours in the regular lattice.
• **edgeWeight** *(float)* – the non-zero weight of newly created edges.

**clusteringCoefficient** ()

Returns the clustering coefficient for the generator.

**generate** *(graph)*

Create a small-world graph using the given input graph.

*Parameters* graph *(apgl.graph.AbstractMatrixGraph)* – The graph to use.

*Returns* The modified input graph.

**setK** *(k)*

The number of neighbours of each vertex.

*Parameters* k *(int)* – the number of neighbours in the regular lattice.

**setP** *(p)*

Set the rewiring probability.

*Parameters* p *(float)* – the probability of rewiring an edge.

### 4.1.2 Graph Classes

This package contains a number of graph types and corresponding utility classes. Graph structure (a set of edges) is stored using adjacency/weight matrices: DenseGraph uses numpy.ndarray, SparseGraph uses scipy.sparse and PySparseGraph uses pysparse. The advantage of using matrices is that one can store edges efficiently as sparse matrices and many graph algorithms can be computed efficiently and with relatively simple code. Furthermore, vertex labels are recorded in the VertexList and GeneralVertexList classes which store values as numpy.ndarrays and list elements respectively. However, the number of vertices remains fixed and one can only store non-zero floating point values on edges.

In the case that one wishes to label vertices and edges with anything and efficiently add vertices, the DictGraph class can be used. To access the functionality of the other graph class classes it can be efficiently converted to one of the other graph types.

**CsArrayGraph**

The CsArrayGraph object represents a graph with an underlying sparse weight matrix representation of csarray (see http://pythonhosted.org/sppy/). This has the advantage of being efficient in memory usage for graphs with few edges. Graphs of a 1,000,000 vertices or more can be created with minimal memory overheads. The following is a very simple example of how to use CsArrayGraph:

```python
from apgl.graph.CsArrayGraph import CsArrayGraph
import numpy

numVertices = 10

graph = CsArrayGraph(numVertices)
graph.addEdge(0, 1)
# Note can also use the notation e.g. graph[0,1] = 1 to create an edge
graph[0, 2] = 1
graph[0, 3] = 1
graph[2, 1] = 1
graph[2, 5] = 1
graph[2, 6] = 1
graph[6, 9] = 1
```
subgraph = graph.subgraph([0, 1, 2, 3])

graph.vlist[0] = "abc"
graph.vlist[1] = 123

The code creates a new CsArrayGraph with 10 vertices, after which edges are added and a subgraph is extracted using vertices 0, 1, 2, and 3. Notice that numpy.array vertices can be added to a CsArrayGraph using the VertexList class in the constructor. Finally, the first and second vertices are initialised with “abc” and 123 respectively.

In order to speed up certain operations on the graph, CsArrayGraph can be initialised with an empty sparse matrix of several types. For example, the csr_matrix allows fast out-degree computations whereas csc_matrix is faster for finding in-degrees of directed graphs.

```python
from apgl.graph.CsArrayGraph import CsArrayGraph
import numpy
import scipy.sparse

numVertices = 10

weightMatrix = scipy.sparse.lil_matrix((numVertices, numVertices))

graph = CsArrayGraph(numVertices, W=weightMatrix)
graph[0, 1] = 1
graph[0, 2] = 1

# Output the number of vertices
print(graph.size)
```

Here, the CsArrayGraph is initialised with 10 vertices and the sparse matrix weightMatrix passed to the constructor is used to store edge weights.

Methods

```python
class apgl.graph.CsArrayGraph.CsArrayGraph (vertices, undirected=True, dtype=<type 'float'>)
    Bases: apgl.graph.AbstractMatrixGraph.AbstractMatrixGraph

    Create a sparse graph using sppy csarray with a given AbstractVertexList, and specify whether directed.

    Parameters
    • vertices – the initial set of vertices as a AbstractVertexList object, or an int to specify the number of vertices in which case vertices are stored in a GeneralVertexList.
    • undirected (boolean) – a boolean variable to indicate if the graph is undirected.
    • dtype – the data type for the weight matrix, e.g numpy.int8.

    W = None
    add (graph)
    Add the edge weights of the input graph to the current one. Results in a union of the edges.

    Parameters
    • graph (apgl.graph.CsArrayGraph) – the input graph.

    Returns A new graph with same vertex list and addition of edge weights

    addEdge (vertexIndex1, vertexIndex2, edge=1)
    Add a non-zero edge between two vertices.

    Parameters
```

4.1. Class Documentation
• `vertexIndex1` (`int`) – The index of the first vertex.
• `vertexIndex2` (`int`) – The index of the second vertex.
• `edge` (`float`) – The value of the edge.

`addEdges` (`edgeIndexArray`, `edgeValues=[])`  
Takes a numpy array of edge index pairs, and edge values and adds them to this graph. The array is 2 dimensional such that each row is a pair of edge indices.

Parameters

• `edgeIndexArray` (`numpy.ndarray`) – The array of edge indices with each being a pair of indices.
• `edgeValues` (`list`) – The list of edge values

`addVertices` (`n`)  
Adds n vertices to the current graph. This is not an efficient operation as we create a new weight matrix and copy the old one. The old vertices are the first m at the start of the new graph.

`adjacencyList` (`useWeights=True`)  
Returns an adjacency list representation L of the graph, in which L[i] is the list of all neighbours of vertex i. Furthermore, the method returns W in which W[i] which is the corresponding set of weights.

Parameters `useWeights` (`bool`) – If true uses weights of edges as opposed to just adjacencies.

Returns `L` A list whose ith element is a list of neighbours for vertex i.

Returns `W` A list whose ith element is a list of neighbour weights/adjacencies for vertex i.

`adjacencyMatrix()`  
Return the adjacency matrix in numpy.ndarray format. Warning: should not be used unless sufficient memory is available to store the dense matrix.

Returns The adjacency matrix in dense format

`betweenness()`  
Return the betweenness of each vertex in the graph. The betweenness is defined as the number of shortest paths passing through each vertex.

Returns A vector of betweenness values of the same length as the number of vertices in the graph.

`breadthFirstSearch` (`root`)  
Breadth first search starting from a particular vertex. Returns a list of connected vertices in the order they were found.

Parameters `root` (`int`) – The index of the root vertex.

Returns A list of vertices connected to the input one via a path in the graph.

`clusteringCoefficient()`  
Find the global clustering coefficient of this graph as defined here http://en.wikipedia.org/wiki/Clustering_coefficient

Returns The clustering coefficient of this graph.

`complement()`  
Returns a graph with identical vertices (same reference) to the current one, but with the complement of the set of edges. Edges that do not exist have weight 1.

`copy()`  
Returns a copy of this object, which also has a copy of the VertexList.
**degreeDistribution()**
Return a vector of (out)degree distributions. The ith element of the vector corresponds to the frequency of degree i.

**Returns** A vector of (out)degree distributions.

**degreeSequence()**
Returns a vector of the degrees (including self edges) for each vertex for an undirected graph.

**density()**
The density of the graph is the number of edges/number of possible edges, which does not include self loops. The density of a graph with no vertices is zero.

**Returns** The density of the graph.

**depthFirstSearch(root)**
Depth first search starting from a particular vertex. Returns a list of connected vertices in the order they were found.

**Parameters**
- **root** (int) – The index of the root vertex.

**Returns** A list of vertices connected to the input one via a path in the graph.

**diameter(useWeights=False, P=None)**
Finds the diameter of a graph i.e. the longest shortest path. If useWeights is True then the weights in the adjacency matrix are used if P is not provided.

**Parameters**
- **useWeights** (bool) – Whether to use edge weights to compute a diameter.
- **P** (ndarray) – An optional nxn matrix whose ijth entry is the shortest path from i to j.

**Returns** The diameter of this graph.

**diameter2()**
Use Dijkstras Algorithm to compute the diameter of the graph.

**Returns** The diameter of the graph.

**dijkstrasAlgorithm(vertexIndex, neighbourLists=None)**
Run Dijkstras Algorithm on the graph for a given source vertex. The parameter neighbourLists is a tuple containing two lists. The first of this lists contains at the ith position all the neighbours of vertex i. The second list contains the corresponding weight on the edge. If neighbourLists=None, then it is computed automatically and all edge weights are set to 1. Returns an array with the distance to all vertices (including itself).

**Parameters**
- **vertexIndex** (int) – the index of the source vertex.
- **neighbourLists** (list) – A tuple of two lists containing vertex adjacencies and edge weights respectively.

**Returns** An array whose ith element is the distance to vertex i.

**effectiveDiameter(q, P=None)**
The effective diameter is the minimum d such that for a fraction q of reachable node pairs, the path length is at most d. This is more robust than the standard diameter method. One can optionally pass in a matrix P whose ijth entry is the shortest path from i to j.

**Parameters**
- **q** (float) – The fraction of node pairs to consider.
- **P** (ndarray) – An optional nxn matrix whose ijth entry is the shortest path from i to j.

**Returns** The effective diameter of this graph.

**egoGraph** *(vertexIndex)*

Returns the subgraph composed of the given vertex and its immediate neighbours. In the new graph, the ego is index 0 and neighbours are indexed in order after 0.

**Parameters** *vertexIndex* (int) – the index of the source vertex.

**Returns** A subgraph of the current one consisting of only immediate neighbours.

**findAllDistances** *(useWeights=True)*

Use the repeated calls to Dijkstra’ algorithm to find the shortest path between all pairs of vertices. If useWeights is true, then the weights are used to compute the path, otherwise adjacencies are used. Note that the shortest path of a vertex to itself is always zero. Returns a matrix whose ijth entry is the shortest path between vertices i and j.

**Parameters** *useWeights* (bool) – Whether to use the edge weight to compute path cost.

**Returns** A matrix of shortest paths between all vertices.

**findConnectedComponents** *()*

Finds a list of all connected components of the graph, in order of size with the smallest first.

**Returns** A list of lists of component indices.

**findTrees** *()*

Returns a list of trees for a directed graph. The reason for only allowing directed graphs is that the root of a tree in an undirected graph is ambiguous. Each tree is represented by an list of indices of vertices in the graph.

**Returns** A list of trees (vertex indices) in the current graph sorted in descending order by size.

**fitPowerLaw** *()*

Fits the out-degree probabilities of this graph using the power law $p_d \sim d^{-\gamma}$. The value of xmin is the point to start taking examples.

**Returns** *alpha* The power law exponent.

**Returns** *ks* A fit of the power law curve to the data using KS.

**Returns** *xmin* The minimum value of x.

**floydWarshall** *(useWeights=True)*

Use the Floyd-Warshall algorithm to find the shortest path between all pairs of vertices. If useWeights is true, then the weights are used to compute the path, otherwise adjacencies are used. Note that the shortest path of a vertex to itself is always zero. Returns a matrix whose ijth entry is the shortest path between vertices i and j. This algorithm scales as $O(n^3)$ with the number of vertices n, and is not recommended for very large graphs.

**Parameters** *useWeights* (bool) – Whether to use the edge weight to compute path cost.

**Returns** A matrix of shortest paths between all vertices.

**classmethod** **fromNetworkXGraph** *(networkXGraph)*

Take a networkx Graph or DiGraph object, and return a subclass of AbstractMatrixGraph. Notice that networkx must be installed to use this function. The networkXGraph graph dict must have an attribute VListType which is the type of the VertexList used to construct the SparseGraph. Furthermore, only node attributes index by “label” are stored in the VertexList, and edge values are currently ignored.

**Returns** A networkx Graph or DiGraph object.
**geodesicDistance** *(P=None, vertexInds=None)*

Compute the mean geodesic distance for a graph. This is denoted for an undirected graph by \(1/(1/2 n(n+1)) \sum_{i<j} d_{ij}\) where \(d_{ij}\) is the shortest path length between \(i\) and \(j\). Note that if \(i\) and \(j\) are not connected we assume a path length of 0. If the graph is directed then the geodesic distance is \(1/(n^2) \sum_{i,j} d_{ij}\).

**Parameters**

- **P** *(ndarray)* – An optional nxn matrix whose ijth entry is the shortest path from \(i\) to \(j\).
- **vertexInds** *(list)* – An optional list of vertices used to compute the mean geodesic distance. If this list is none, then all vertices are used.

**Returns** The mean geodesic distance of this graph.

**getAllDirEdges** *

Returns the set of directed edges of the current graph as a matrix in which each row corresponds to an edge. For an undirected graph, there is an edge from \(v1\) to \(v2\) and from \(v2\) to \(v1\) if \(v2!=v1\).

**Returns** A matrix with 2 columns, and each row corresponding to an edge.

**getAllEdges** *

Returns the set of edges of the current graph as a matrix in which each row corresponds to an edge. For an undirected graph, \(v1>=v2\).

**Returns** A matrix with 2 columns, and each row corresponding to an edge.

**getAllVertexIds** *

Returns a list of all the vertex IDs of this graph.

**getEdge** *(vertexIndex1, vertexIndex2)*

Get the value of an edge, or None if no edge exists.

**Parameters**

- **vertexIndex1** *(int)* – The index of the first vertex.
- **vertexIndex2** *(int)* – The index of the second vertex.

**Returns** The value of the edge between the given vertex indices.

**getEdgeValues** *(edgeArray)*

Take an array of \(n \times 2\) of vertex indices and return the corresponding edge values.

**Parameters**

- **edgeArray** *(numpy.ndarray)* – An array with an edge on each row

**Returns** A vector of \(n\) values corresponding to the edge weights of edgeArray

**getNumDirEdges** *

Returns the number of edges, taking this graph as a directed graph.

**getNumEdges** *

Returns the total number of edges in this graph.

**getNumVertices** *

Returns the number of vertices in this graph.

**getVertex** *(vertexIndex)*

Returns the vertex associated with the given vertex index.

**Parameters**

- **vertexIndex** *(int)* – The index of the vertex.

**Returns** The value of the vertex at the given index.

**getVertexList** *

Returns the AbstractVertexList object of this graph.
getVertices(vertexIndices)
    Takes a list of vertex indices and returns the corresponding vertex values.
    
    Parameters vertexIndices (list) – A list of vertex indices
    
    Returns A list of vertices corresponding to the indices

getWeightMatrix()
    Return the weight matrix as a numpy array.

harmonicGeodesicDistance(P=None, vertexInds=None)
    Compute the “harmonic mean” geodesic distance for a graph. This is denoted by the inverse of
    1/(1/2 n(n+1)) sum_{[i<j]} d_{ij}^-1 where d_{ij} is the shortest path length between i and j for an undirected graph. The distance from a node to itself is infinite. For a directed graph, the inverse distance is 1/n^2 sum_{[i,j]} d_{ij}^-1.
    
    Parameters
    - P (ndarray) – An optional nxn matrix whose ijth entry is the shortest path from i to j.
    - vertexInds (list) – An optional list of vertices used to compute the mean geodesic distance. If this list is none, then all vertices are used.
    
    Returns The mean harmonic geodesic distance of this graph.

hopCount(P=None)
    Returns an array such that the ith element is the number of pairs of vertices reachable within i hops. This includes self pairs, and all other pairs are counted twice in the undirected case otherwise once.
    
    Parameters P (ndarray) – An optional nxn matrix whose ijth entry is the shortest unweighted path from i to j.
    
    Returns An array of hop counts.

inDegreeDistribution()
    Returns a vector of in-degree distributions. The ith element of the vector corresponds to the frequency of degree i.
    
    Returns A vector of (in)degree distributions.

inDegreeSequence()
    Return a vector of the (out)degree for each vertex.

incidenceMatrix()
    Return the incidence matrix of this graph as a scipy sparse matrix. The incidence matrix X is of size numVertices x numEdges, and has a 1 in element Xij = -1 if edge j leaves vertex i, and Xij = 1 if edge j enters vertex i. Notice that for an undirected graph XX^T is the laplacian matrix.

intersect(graph)
    Take the intersection of the edges of this graph and the input graph. Resulting edge weights are ignored and only adjacencies are stored.
    
    Parameters graph (apgl.graph.CsArrayGraph) – the input graph.
    
    Returns A new graph with the intersection of edges of the current plus graph

isTree()
    Returns true if this graph is a tree. Every vertex must have an in-degree of 1 (i.e. one parent), except the root which has an in-degree of zero and non-zero out-degree.
    
    Returns A boolean indicating whether the current graph is a tree.

isUndirected()
    Returns true if this graph is undirected otherwise false.
laplacianMatrix(outDegree=True)
Return the Laplacian matrix of this graph, which is defined as \(L_{ii} = \deg(i)\) \(L_{ij} = -1\) if an edge between i and j, otherwise \(L_{ij} = 0\). For a directed graph one can specify whether to use the out-degree or in-degree.

Parameters outDegree (bool) – whether to use the out-degree for the computation of the degree matrix

Returns A laplacian adjacency matrix as numpy array.

laplacianWeightMatrix(outDegree=True)
Return the Laplacian matrix of this graph, \(L = D - W\), where \(D\) is the degree matrix and \(W\) is the weight matrix. For a directed graph one can specify whether to use the out-degree or in-degree.

Parameters outDegree (bool) – whether to use the out-degree for the computation of the degree matrix

Returns A laplacian weight matrix.

classmethod load(filename)
Load the graph object from the corresponding file. Data is loaded in a zip format as created using save().

Parameters filename (str) – The name of the file to load.

Returns A graph corresponding to the one saved in filename.

static loadMatrix(filename)

maxEigenvector()
Returns the eigenvector of maximum eigenvalue of the adjacency matrix. The eigenvector is of unit length, and measures the centrality of the corresponding vertex. It is based on the principle that connections to high-scoring nodes contribute more to the score of the node in question than equal connections to low-scoring nodes.

Returns The maximum eigenvector of the adjacency matrix.

maxProductPaths()
Find the maximum product paths between all pairs of vertices using a modified version of the Floyd-Warshall algorithm.

Returns A matrix P whose ijth entry corresponds to the maximal product of edge weights between them.

maybeIsomorphicWith(graph)
Returns false if graph is definitely not isomorphic with the current graph, however a True may mean the graphs are not isomorphic. Makes a comparison with the eigenvalues of the Laplacian matrices.

Returns True if the current graph is maybe isomorphic with the input one.

multiply(graph)
Multiply the edge weights of the input graph to the current one. Results in an intersection of the edges.

Parameters graph (apgl.graph.CsArrayGraph) – the input graph.

Returns A new graph with edge weights which are multiples of the current and graph

neighbourOf(vertexIndex)
Return an array of the indices of vertices than have an edge going to the input vertex.

Parameters vertexIndex (int) – the index of a vertex.

neighbours(vertexIndex)
Return an array of the indices of the neighbours of the given vertex.

Parameters vertexIndex (int) – the index of a vertex.
normalisedLaplacianRw (outDegree=True)
Compute the normalised random walk laplacian matrix with $L = I - D^{-1} W$ in which $W$ is the weight matrix and $D_{ii}$ is the sum of the $i$th vertices weights.

Parameters outDegree (bool) – whether to use the out-degree for the computation of the degree matrix

Returns A normalised random-walk laplacian matrix as a numpy array.

normalisedLaplacianSym (outDegree=True)
Compute the normalised symmetric laplacian matrix using $L = I - D^{-1/2} W D^{-1/2}$, in which $W$ is the weight matrix and $D_{ii}$ is the sum of the $i$th vertices weights.

Parameters outDegree (bool) – whether to use the out-degree for the computation of the degree matrix

Returns A normalised symmetric laplacian matrix as a numpy array.

outDegreeSequence ()
Return a vector of the (out)degree for each vertex.

removeAllEdges ()
Removes all edges from this graph.

removeEdge (vertexIndex1, vertexIndex2)
Remove an edge between two vertices.

Parameters
• vertexIndex1 (int) – The index of the first vertex.
• vertexIndex2 (int) – The index of the second vertex.

save (filename)
Save the graph object to the corresponding filename under the .zip extension. The adjacency matrix is stored in matrix market format and the AbstractVertexList decides how to store the vertex labels.

Parameters filename (str) – The name of the file to save.

Returns The name of the saved zip file.

saveMatrix (W, filename)

setDiff (graph)
Find the edges in the current graph which are not present in the input graph. Replaces the edges in the current graph with adjacencies.

Parameters graph (apgl.graph.CsArrayGraph) – the input graph.

Returns The graph which is the set difference of the edges of this graph and graph.

setVertex (vertexIndex, vertex)
Set the vertex with given index to a particular value.

Parameters
• vertexIndex (int) – the index of the vertex.
• vertex – the value of the vertex.

setVertexList (vList)
Assign a new VertexList object to this graph. The number of vertices in the VertexList must be the same as in the graph.

Parameters vList (apgl.graph.AbstractVertexList) – A new subclass of AbstractVertexList to assign to this graph.
setVertices \((vertexIndices, vertices)\)
Assign new values to the vertices corresponding to vertexIndices

Parameters

- vertexIndices (list) – A list of vertex indices
- vertices (list) – A list of vertex values

setWeightMatrix \((W)\)
Set the weight matrix of this graph. Requires as input an ndarray or a scipy sparse matrix with the same dimensions as the current weight matrix. Edges are represented by non-zero edges.

Parameters \(W\) (ndarray or scipy.sparse matrix) – The weight matrix to use.

setWeightMatrixSparse \((W)\)
Set the weight matrix of this graph. Requires as input a scipy sparse matrix with the same dimensions as the current weight matrix. Edges are represented by non-zero edges.

Parameters \(W\) – The scipy sparse weight matrix to use.

size
The number of vertices in the graph

subgraph \((vertexIndices)\)
Pass in a list or set of vertexIndices and returns the subgraph containing those vertices only, and edges between them.

Parameters vertexIndices (list) – the indices of the subgraph vertices.

toDictGraph ()
Convert to a DictGraph object. Currently ignores vertex labels.

Return graph A DictGraph object.

toIGraph ()
Convert this graph into a igraph Graph object, which requires igraph to be installed. Edge values are stored under the “value” index. Vertices are stored as indices with a “label” value being the corresponding vertex value.

Returns An igraph Graph object.

toNetworkXGraph ()
Convert this graph into a networkx Graph or DiGraph object, which requires networkx to be installed. Edge values are stored under the “value” index. Vertices are stored as indices with a “label” value being the corresponding vertex value. The type of vertex list is stored as a graph attribute under the index “VListType”

Returns A networkx Graph or DiGraph object.

triangleSequence ()
Computes the number of triangles each vertex participates in using the diagonal of the adjacency matrix. In an undirected graph, each triangle is counted twice (once for each direction). Note that self loops are not used to form triangles.

Returns An array of triangle counts for each vertex.

undirected = None

union \((graph)\)
Take the union of the edges of this graph and the input graph. Resulting edge weights are ignored and only adjacencies are stored.

Parameters graph (apgl.graph.CsArrayGraph) – the input graph.
**Returns** A new graph with the union of edges of the current one.

\[ vList = None \]

\[ vlist \]

The vertex list

\[ weightMatrixDType() \]

**Returns** the dtype of the matrix used to store edge weights.

**DenseGraph**

A graph who edges are represented by a dense (numpy.ndarray) weight matrix, and is otherwise similar to SparseGraph. The following is a very simple example of how to use DenseGraph:

```python
from apgl.graph import DenseGraph
import numpy

numVertices = 10

graph = DenseGraph(numVertices)
graph[0, 2] = 1
graph[0, 3] = 1
graph[2, 1] = 1
graph[2, 5] = 1
graph[2, 6] = 1
graph[6, 9] = 1

subgraph = graph.subgraph([0,1,2,3])

graph.vlist[0] = "abc"
graph.vlist[1] = 123
```

The code creates a new DenseGraph with 10 vertices, after which edges are added and a subgraph is extracted using vertices 0, 1, 2, and 3. Notice that numpy.array vertices can be added to a DenseGraph using the VertexList class in the constructor. Finally, the first and second vertices are initialised with “abc” and 123 respectively.

**Methods**

```python
class apgl.graph.DenseGraph.DenseGraph(vertices, undirected=True, W=None, dtype=<type 'float'>)
```

Bases: `apgl.graph.AbstractMatrixGraph.AbstractMatrixGraph`

Create a DenseGraph with a given AbstractVertexList or number of vertices, and specify whether it is directed. One can optionally pass in a numpy array W which is used as the weight matrix of the graph.

**Parameters**

- `vertices` – the initial set of vertices as a AbstractVertexList object, or an int to specify the number of vertices in which case vertices are stored in a GeneralVertexList.
- `undirected` (boolean) – a boolean variable to indicate if the graph is undirected.
- `W` – a numpy array of the same size as vertices, or None to create the default one.
- `dtype` – the data type of the weight matrix if W is not specified e.g numpy.int8.

\[ W = None \]
add(graph)
   Add the edge weights of the input graph to the current one. Results in a union of the edges.
   Parameters graph (apgl.graph.DenseGraph) – the input graph.
   Returns A new graph with same vertex list and addition of edge weights
addEdge (vertexIndex1, vertexIndex2, edge=1)
   Add a non-zero edge between two vertices.
   Parameters
      • vertexIndex1 (int) – The index of the first vertex.
      • vertexIndex2 (int) – The index of the second vertex.
      • edge (float) – The value of the edge.
addEdges (edgeIndexArray, edgeValues=[])
   Takes a numpy array of edge index pairs, and edge values and adds them to this graph. The array is 2
dimensional such that each row is a pair of edge indices.
   Parameters
      • edgeIndexArray (numpy.ndarray) – The array of edge indices with each being a pair
        of indices.
      • edgeValues (list) – The list of edge values

adjacencyList (useWeights=True)
   Returns an adjacency list representation L of the graph, in which L[i] is the list of all neighbours of vertex
   i. Furthermore, the method returns W in which W[i] which is the corresponding set of weights.
   Parameters useWeights (bool) – If true uses weights of edges as opposed to just adjacencies.
   Returns L A list whose ith element is a list of neighbours for vertex i.
   Returns W A list whose ith element is a list of neighbour weights/adjacencies for vertex i.

adjacencyMatrix()
   Return the adjacency matrix in numpy.ndarray format. Warning: should not be used unless sufficient
   memory is available to store the dense matrix.
   Returns The adjacency matrix in dense format

betweenness()
   Return the betweenness of each vertex in the graph. The betweenness is defined as the number of shortest
   paths passing through each vertex.
   Returns A vector of betweenness values of the same length as the number of vertices in the
   graph.
breadthFirstSearch (root)
   Breadth first search starting from a particular vertex. Returns a list of connected vertices in the order they
   were found.
   Parameters root (int) – The index of the root vertex.
   Returns A list of vertices connected to the input one via a path in the graph.

clusteringCoefficient()
   Find the global clustering coefficient of this graph as defined here
   http://en.wikipedia.org/wiki/Clustering_coefficient
   Returns The clustering coefficient of this graph.
complement()
Returns a graph with identical vertices (same reference) to the current one, but with the complement of the set of edges. Edges that do not exist have weight 1.

copy()
Returns a copy of this object, which also has a copy of the VertexList.

degreeDistribution()
Return a vector of (out)degree distributions. The ith element of the vector corresponds to the frequency of degree i.

    Returns A vector of (out)degree distributions.

degreeSequence()
Returns a vector of the degrees (including self edges) for each vertex for an undirected graph.

density()
The density of the graph is the number of edges/number of possible edges, which does not include self loops. The density of a graph with no vertices is zero.

    Returns The density of the graph.

depthFirstSearch(root)
Depth first search starting from a particular vertex. Returns a list of connected vertices in the order they were found.

    Parameters root (int) – The index of the root vertex.

    Returns A list of vertices connected to the input one via a path in the graph.

diameter(useWeights=False, P=None)
Finds the diameter of a graph i.e. the longest shortest path. If useWeights is True then the weights in the adjacency matrix are used if P is not provided.

    Parameters

        • useWeights (bool) – Whether to use edge weights to compute a diameter.

        • P (ndarray) – An optional nxn matrix whose ijth entry is the shortest path from i to j.

    Returns The diameter of this graph.

diameter2()
Use Dijkstras Algorithm to compute the diameter of the graph.

    Returns The diameter of the graph.

dijkstrasAlgorithm(vertexIndex, neighbourLists=None)
Run Dijkstras Algorithm on the graph for a given source vertex. The parameter neighbourLists is a tuple containing two lists. The first of this lists contains at the ith position all the neighbours of vertex i. The second list contains the corresponding weight on the edge. If neighbourLists=None, then it is computed automatically and all edge weights are set to 1. Returns an array with the distance to all vertices (including itself).

    Parameters

        • vertexIndex (int) – the index of the source vertex.

        • neighbourLists (list) – A tuple of two lists containing vertex adjacencies and edge weights respectively.

    Returns An array whose ith element is the distance to vertex i.
effectiveDiameter \((q, P=None)\)

The effective diameter is the minimum \(d\) such that for a fraction \(q\) of reachable node pairs, the path length is at most \(d\). This is more robust than the standard diameter method. One can optionally pass in a matrix \(P\) whose \(ij\)th entry is the shortest path from \(i\) to \(j\).

**Parameters**

- \(q\) \((\text{float})\) – The fraction of node pairs to consider.
- \(P\) \((\text{ndarray})\) – An optional nxn matrix whose \(ij\)th entry is the shortest path from \(i\) to \(j\).

**Returns**  The effective diameter of this graph.

egoGraph \((\text{vertexIndex})\)

Returns the subgraph composed of the given vertex and its immediate neighbours. In the new graph, the ego is index 0 and neighbours are indexed in order after 0.

**Parameters**  vertexIndex \((\text{int})\) – the index of the source vertex.

**Returns**  A subgraph of the current one consisting of only immediate neighbours.

findAllDistances \((\text{useWeights}=\text{True})\)

Use the repeated calls to Dijkstra’s algorithm to find the shortest path between all pairs of vertices. If useWeights is true, then the weights are used to compute the path, otherwise adjacencies are used. Note that the shortest path of a vertex to itself is always zero. Returns a matrix whose \(ij\)th entry is the shortest path between vertices \(i\) and \(j\).

**Parameters**  useWeights \((\text{bool})\) – Whether to use the edge weight to compute path cost.

**Returns**  A matrix of shortest paths between all vertices.

findConnectedComponents()

Finds a list of all connected components of the graph, in order of size with the smallest first.

**Returns**  A list of lists of component indices.

findTrees()

Returns a list of trees for a directed graph. The reason for only allowing directed graphs is that the root of a tree in an undirected graph is ambiguous. Each tree is represented by an list of indices of vertices in the graph.

**Returns**  A list of trees (vertex indices) in the current graph sorted in descending order by size.

fitPowerLaw()

Fits the out-degree probabilities of this graph using the power law \(p_d \sim d^{-\gamma}\). The value of \(xmin\) is the point to start taking examples.

**Returns** alpha  The power law exponent.

**Returns** ks  A fit of the power law curve to the data using KS.

**Returns** xmin  The minimum value of \(x\).

floydWarshall \((\text{useWeights}=\text{True})\)

Use the Floyd-Warshall algorithm to find the shortest path between all pairs of vertices. If useWeights is true, then the weights are used to compute the path, otherwise adjacencies are used. Note that the shortest path of a vertex to itself is always zero. Returns a matrix whose \(ij\)th entry is the shortest path between vertices \(i\) and \(j\). This algorithm scales as \(O(n^3)\) with the number of vertices \(n\), and is not recommended for very large graphs.

**Parameters**  useWeights \((\text{bool})\) – Whether to use the edge weight to compute path cost.

**Returns**  A matrix of shortest paths between all vertices.
classmethod `fromNetworkXGraph`(networkXGraph)

Take a networkx Graph or DiGraph object, and return a subclass of AbstractMatrixGraph. Notice that networkx must be installed to use this function. The networkXGraph graph dict must have an attribute VListType which is the type of the VertexList used to construct the SparseGraph. Furthermore, only node attributes index by “label” are stored in the VertexList, and edge values are currently ignored.

**Returns** A networkx Graph or DiGraph object.

`geodesicDistance`(P=None, vertexInds=None)

Compute the mean geodesic distance for a graph. This is denoted for an undirected graph by $1/(1/2 n(n+1)) \sum_{i \leq j} d_{ij}$ where $d_{ij}$ is the shortest path length between $i$ and $j$. Note that if $i$ and $j$ are not connected we assume a path length of 0. If the graph is directed then the geodesic distance is $1/(n^2) \sum_{i,j} d_{ij}$.

**Parameters**

- P (ndarray) – An optional nxn matrix whose ijth entry is the shortest path from i to j.
- vertexInds (list) – An optional list of vertices used to compute the mean geodesic distance. If this list is none, then all vertices are used.

**Returns** The mean geodesic distance of this graph.

`getAllDirEdges`()

Returns the set of directed edges of the current graph as a matrix in which each row corresponds to an edge. For an undirected graph, there is an edge from v1 to v2 and from v2 to v1 if v2!=v1.

**Returns** A matrix with 2 columns, and each row corresponding to an edge.

`getAllEdges`()

Returns the set of edges of the current graph as a matrix in which each row corresponds to an edge. For an undirected graph, v1>=v2.

**Returns** A matrix with 2 columns, and each row corresponding to an edge.

`getAllVertexIds`()

Returns a list of all the vertex IDs of this graph.

`getEdge`(vertexIndex1, vertexIndex2)

Get the value of an edge, or None if no edge exists.

**Parameters**

- vertexIndex1 (int) – The index of the first vertex.
- vertexIndex2 (int) – The index of the second vertex.

**Returns** The value of the edge between the given vertex indices.

`getEdgeValues`(edgeArray)

Take an array of n x 2 of vertex indices and return the corresponding edge values.

**Parameters**

- edgeArray (numpy.ndarray) – An array with an edge on each row

**Returns** A vector of n values corresponding to the edge weights of edgeArray

`getNumDirEdges`()

Returns the number of edges, taking this graph as a directed graph.

`getNumEdges`()

Returns the total number of edges in this graph.

`getNumVertices`()

**Returns** the number of vertices in this graph.
getVertex (vertexIndex)
Returns the vertex associated with the given vertex index.

Parameters vertexIndex (int) – the index of the vertex.
Returns The value of the vertex at the given index.

getVertexList ()
Returns the AbstractVertexList object of this graph.

gVertices (vertexIndices)
Takes a list of vertex indices and returns the corresponding vertex values.

Parameters vertexIndices (list) – A list of vertex indices
Returns A list of vertices corresponding to the indices

getWeightMatrix ()
Return the weight matrix as a numpy array.

harmonicGeodesicDistance (P=None, vertexInds=None)
Compute the “harmonic mean” geodesic distance for a graph. This is denoted by the inverse of 1/(1/2 n(n+1)) \sum_{i<j} d_{ij}^{-1} where d_{ij} is the shortest path length between i and j for an undirected graph. The distance from a node to itself is infinite. For a directed graph, the inverse distance is 1/n^2 \sum_{i,j} d_{ij}^{-1}.

Parameters
• P (ndarray) – An optional nxn matrix whose ijth entry is the shortest path from i to j.
• vertexInds (list) – An optional list of vertices used to compute the mean geodesic distance. If this list is none, then all vertices are used.

Returns The mean harmonic geodesic distance of this graph.

hopCount (P=None)
Returns an array such that the ith element is the number of pairs of vertices reachable within i hops. This includes self pairs, and all other pairs are counted twice in the undirected case otherwise once.

Parameters P (ndarray) – An optional nxn matrix whose ijth entry is the shortest unweighted path from i to j.

Returns An array of hop counts.

inDegreeDistribution ()
Returns a vector of in-degree distributions. The ith element of the vector corresponds to the frequency of degree i.

Returns A vector of (in)degree distributions.

inDegreeSequence ()
Return a vector of the (out)degree for each vertex.

incidenceMatrix ()
Return the incidence matrix of this graph as a scipy sparse matrix. The incidence matrix X is of size numVertices x numEdges, and has a 1 in element Xij = -1 if edge j leaves vertex i, and Xij = 1 if edge j enters vertex i. Notice that for an undirected graph XX^T is the laplacian matrix.

intersect (graph)
Take the intersection of the edges of this graph and the input graph. Resulting edge weights are ignored and only adjacencies are stored.

Parameters graph (apgl.graph.DenseGraph) – the input graph.
Returns A new graph with the intersection of edges of the current plus graph

isTree()  
Returns true if this graph is a tree. Every vertex must have an in-degree of 1 (i.e. one parent), except the  
root which has an in-degree of zero and non-zero out-degree.

Returns A boolean indicating whether the current graph is a tree.

isUndirected()  
Returns true if this graph is undirected otherwise false.

laplacianMatrix(outDegree=True)  
Return the Laplacian matrix of this graph, which is defined as $L_{ii} = \deg(i)$ $L_{ij} = -1$ if an edge  
between i and j, otherwise $L_{ij} = 0$. For a directed graph one can specify whether to use the out-degree  
or in-degree.

Parameters outDegree (bool) – whether to use the out-degree for the computation of the de-  
gree matrix

Returns A laplacian adjacency matrix as numpy array.

laplacianWeightMatrix(outDegree=True)  
Return the Laplacian matrix of this graph, $L = D - W$, where D is the degree matrix and W is the weight  
matrix. For a directed graph one can specify whether to use the out-degree or in-degree.

Parameters outDegree (bool) – whether to use the out-degree for the computation of the de-  
gree matrix

Returns A laplacian weight matrix.

classmethod load(filename)  
Load the graph object from the corresponding file. Data is loaded in a zip format as created using save().

Parameters filename (str) – The name of the file to load.

Returns A graph corresponding to the one saved in filename.

static loadMatrix(filename)  

maxEigenvector()  
Returns the eigenvector of maximum eigenvalue of the adjacency matrix. The eigenvector is of unit length,  
and measures the centrality of the corresponding vertex. It is based on the principle that connections to  
high-scoring nodes contribute more to the score of the node in question than equal connections to low-  
scoring nodes.

Returns The maximum eigenvector of the adjacency matrix.

maxProductPaths()  
Find the maximum product paths between all pairs of vertices using a modified version of the Floyd-  
Warshall algorithm.

Returns A matrix P whose ijth entry corresponds to the maximal product of edge weights be-  
tween them.

maybeIsomorphicWith(graph)  
Returns false if graph is definitely not isomorphic with the current graph, however a True may mean the  
graphs are not isomorphic. Makes a comparison with the eigenvalues of the Laplacian matrices.

Returns True if the current graph is maybe isomorphic with the input one.

multiply(graph)  
Multiply the edge weights of the input graph to the current one. Results in an intersection of the edges.

Parameters graph (apgl.graph.DenseGraph) – the input graph.
Returns A new graph with edge weights which are multiples of the current and graph

neighbourOf(\text{vertexIndex})
Return an array of the indices of vertices than have an edge going to the input vertex.

Parameters vertexIndex (int) – the index of a vertex.

neighbours(\text{vertexIndex})
Return an array of the indices of the neighbours of the given vertex.

Parameters vertexIndex (int) – the index of a vertex.

normalisedLaplacianRw(outDegree=True)
Compute the normalised random walk laplacian matrix with \( L = I - D^{-1} \) \( W \) in which \( W \) is the weight matrix and \( D_{ii} \) is the sum of the \( i \)th vertices weights.

Parameters outDegree (bool) – whether to use the out-degree for the computation of the degree matrix

Returns A normalised random-walk laplacian matrix as a numpy array.

normalisedLaplacianSym(outDegree=True)
Compute the normalised symmetric laplacian matrix using \( L = I - D^{-1/2} \) \( W \) \( D^{-1/2} \), in which \( W \) is the weight matrix and \( D_{ii} \) is the sum of the \( i \)th vertices weights.

Parameters outDegree (bool) – whether to use the out-degree for the computation of the degree matrix

Returns A normalised symmetric laplacian matrix as a numpy array.

outDegreeSequence()
Return a vector of the (out)degree for each vertex.

removeAllEdges()
Removes all edges from this graph.

removeEdge(\text{vertexIndex1}, \text{vertexIndex2})
Remove an edge between two vertices.

Parameters

• vertexIndex1 (int) – The index of the first vertex.

• vertexIndex2 (int) – The index of the second vertex.

save(filename)
Save the graph object to the corresponding filename under the .zip extension. The adjacency matrix is stored in matrix market format and the AbstractVertexList decides how to store the vertex labels.

Parameters filename (str) – The name of the file to save.

Returns The name of the saved zip file.

saveMatrix(W, filename)

setDiff(graph)
Find the edges in the current graph which are not present in the input graph. Replaces the edges in the current graph with adjacencies.

Parameters graph (apgl.graph.DenseGraph) – the input graph.

Returns The graph which is the set difference of the edges of this graph and graph.

setVertex(\text{vertexIndex}, \text{vertex})
Set the vertex with given index to a particular value.
Parameters

- **vertexIndex** *(int)* – the index of the vertex.
- **vertex** – the value of the vertex.

`setVertexList(vList)`

Assign a new VertexList object to this graph. The number of vertices in the VertexList must be the same as in the graph.

**Parameters**

- **vList** *(apgl.graph.AbstractVertexList)* – A new subclass of AbstractVertexList to assign to this graph.

`setVertices(vertexIndices, vertices)`

Assign new values to the vertices corresponding to vertexIndices

**Parameters**

- **vertexIndices** *(list)* – A list of vertex indices
- **vertices** *(list)* – A list of vertex values

`setWeightMatrix(W)`

Set the weight matrix of this graph. Requires as input an ndarray or a scipy sparse matrix with the same dimensions as the current weight matrix. Edges are represented by non-zero edges.

**Parameters**

- **W** *(ndarray or scipy.sparse matrix)* – The weight matrix to use.

`size`  

The number of vertices in the graph

`subgraph(vertexIndices)`

Pass in a list or set of vertexIndices and returns the subgraph containing those vertices only, and edges between them.

**Parameters**

- **vertexIndices** *(list)* – the indices of the subgraph vertices.

`toDictGraph()`

Convert to a DictGraph object. Currently ignores vertex labels.

**Returns**

- **graph** A DictGraph object.

`toIGraph()`

Convert this graph into a igraph Graph object, which requires igraph to be installed. Edge values are stored under the “value” index. Vertices are stored as indices with a “label” value being the corresponding vertex value.

**Returns**

- **graph** An igraph Graph object.

`toNetworkXGraph()`

Convert this graph into a networkx Graph or DiGraph object, which requires networkx to be installed. Edge values are stored under the “value” index. Vertices are stored as indices with a “label” value being the corresponding vertex value. The type of vertex list is stored as a graph attribute under the index “VListType”

**Returns**

- **graph** A networkx Graph or DiGraph object.

`triangleSequence()`

Computes the number of triangles each vertex participates in using the diagonal of the adjacency matrix. In an undirected graph, a each triangle is counted twice (once for each direction). Note that self loops are not used to form triangles.

**Returns**

- **triangle_counts** An array of triangle counts for each vertex.

```
undirected = None
```
union (graph)
    Take the union of the edges of this graph and the input graph. Resulting edge weights are ignored and only
    adjacencies are stored.

    Parameters graph (apgl.graph.DenseGraph) – the input graph.

    Returns A new graph with the union of edges of the current one.

vList = None

vlist
    The vertex list

weightMatrixDType()

    Returns the dtype of the matrix used to store edge weights.

DictGraph

A graph with nodes stored in a dictionary. In particular the graph data structure is a dict of dicts and edges and vertices
 can be labeled with anything. This class is useful because unlike the graphs represented using adjacency/weight
 matrices, one can efficiently add vertices. For example:

from apgl.graph import *

g = DictGraph()
g.addEdge("a", "b")
g.addEdge("a", "c")
g.addEdge("a", "d")
g["d", "e"] = 1

g.addEdge("d", "f", "abc")

In this code snippet, we construct an undirected graph and then add 5 edges to it with vertex labels as the strings a-e
and edge labels 1. In the final line we have attached an edge label of “abc” to the edge going from “d” to “f”. Note
that one can easily take a DictGraph and copy the edges to one of the other graph classes:

from apgl.graph import *

g = DictGraph()
g.addEdge("a", "b")
g.addEdge("a", "c")
g.addEdge("a", "d")
edgeIndices = g.getAllEdgeIndices()

g2 = SparseGraph(GeneralVertexList(g.getNumVertices()))
g2.addEdges(edgeIndices)

In this case the edges from graph are added to graph2 which is a SparseGraph object and represented using a
scipy.sparse matrix. The corresponding mapping from vertex labels in graph and graph2 is found using getAllVer-
texIds.

Methods

class apgl.graph.DictGraph.DictGraph (undirected=True)
    Create a new DictGraph.

        Parameters undirected (bool) – Specify whether the graph has directed or undirected edges
addEdge \((vertex1, vertex2, value=1.0)\)
Add an edge between two vertices.

Parameters
- vertex1 – The name of the first vertex.
- vertex2 – The name of the second vertex.
- value – The value of the edge.

addEdges \((edgeList, edgeValues=None)\)
Add a set of edges to this graph given by a list of tuples of vertex ids. The value of each edge is simply set to 1 if edgeValues=None otherwise it is set to the corresponding entry of edgeValues.

Parameters
- edgeList – A list of pairs of verted ids
- edgeValues – A corresponding list of vertex values.

adjacencies = None

degrees = None
adjacencyList ()
Returns an adjacency list representation L of the graph, in which L[i] is the list of all neighbour indices of vertex index i. Furthermore, the method returns W in which W[i] which is the corresponding set of weights. In essence, this method is a way to map vertex ids to indices.

Returns L A list whose ith element is a list of neighbours for vertex i.
Returns W A list whose ith element is a list of neighbour weights for vertex i.

breadthFirstSearch (root)
Breadth first search starting from a particular vertex. Returns a list of connected vertices in the order they were found.

Parameters root (int) – The index of the root vertex.
Returns A list of vertices connected to the input one via a path in the graph.

degreeSequence ()
Returns a vector of the degrees (including self edges) for each vertex for an undirected graph.

density ()
The density of the graph is the number of edges/number of possible edges, which does not include self loops. The density of a graph with no vertices is zero.

Returns The density of the graph.

depthFirstSearch (root)
Depth first search starting from a particular vertex. Returns a list of connected vertices in the order they were found.

Parameters root (int) – The index of the root vertex.
Returns A list of vertices connected to the input one via a path in the graph.

dijkstra'sAlgorithm (vertexId, neighbourLists=None)
Run Dijkstra's Algorithm on the graph for a given source vertex. Returns an array with the distance to all vertices (including itself).

Parameters vertexId – the index of the source vertex.
Returns An array whose ith element is the distance to vertex i.
edgeExists (vertexId1, vertexId2)
Return true if the edge exists between two vertices

findAllDistances (useWeights=True)
Use the repeated calls to Dijkstra's algorithm to find the shortest path between all pairs of vertices. Note
that the shortest path of a vertex to itself is always zero. Returns a matrix whose ij th entry is the shortest
path between vertices i and j.

Returns A matrix of shortest paths between all vertices.

findConnectedComponents ()
Finds a list of all connected components of the graph, in order of size with the smallest first.

Returns A list of lists of component indices.

getAllEdgeIndices ()
Returns a numpy array of size (numEdges x 2) of edge index pairs V. The ith row of V, V[i, :], corresponds
to an edge from V[i, 0] to V[i, 1]. The corresponding vertex names are found using getAllVertexIds().

gGetAllEdges ()
Returns a list of tuples of all the edges of this graph.

gGetAllVertexIds ()
Returns a list of the vertex ids (or names) in this graph.

gGetEdge (vertex1, vertex2)
Returns the value of the edge between two vertices. If there is no edge between the vertices None is
returned.

Parameters
• vertex1 – The name of the first vertex.
• vertex2 – The name of the second vertex.

gGetNumDirEdges ()
Returns the number of edges, taking this graph as a directed graph.

gGetNumEdges ()
Returns the total number of edges in graph.

gGetNumVertices ()
Returns the number of vertices in the graph.

gGetSparseWeightMatrix (format='lil')
Returns a weight matrix representation of the graph as a scipy sparse lil_matrix by default. The in-
dices in the matrix correspond to the keys returned by getAllVertexIds. Available formats are: lil for
scipy.sparse.lil_matrix, csr for scipy.sparse.csr_matrix, csc for scipy.sparse.csc_matrix, and pysparse for
pysparse's ll_mat.

Parameters format – The format of the sparse matrix.

gGetVertex (vertexId)
Returns the label of the given vertex, or None if no label.

Parameters vertex – The name of the first vertex.

gGetVertices (vertexIndices)
Takes a list of vertex indices and returns the corresponding vertex values.

Parameters vertexIndices (list) – A list of vertex indices

Returns A list of vertices corresponding to the indices
getWeightMatrix()  
Returns a weight matrix representation of the graph as a numpy array. The indices in the matrix correspond to the keys returned by getAllVertexIds.

inDegreeSequence()  
Find the in degree sequence. Return the sequence as a vector along with the corresponding vertices in a list.

isUndirected()  
Returns true if the current graph is undirected, otherwise false.

neighbourOf(vertex)  
Returns the list of neighbours of the current neighbour.

neighbours(vertexId)  
Find a list of neighbours of the current vertex. In a directed graph, it is the list of all vertices with an edge from the current vertex.

Parameters  
vertexId – The id of the vertex.

outDegreeSequence()  
Find the out degree sequence. Return the sequence as a vector along with the corresponding vertices in a list.

removeEdge(vertex1, vertex2)  
Remove an edge. Does not remove the vertices.

Parameters  
• vertex1 – The name of the first vertex.  
• vertex2 – The name of the second vertex.

removeVertex(vertexId)  
Remove a vertex and all its edges.

Parameters  
vertexId – The id of the vertex to remove.

setVertex(vertexId, vertex)  
Sets the vertexId with the value. Overwrites value if already present.

setVertices(vertexIndices, vertices)  
Assign new values to the vertices corresponding to vertexIndices.

Parameters  
• vertexIndices (list) – A list of vertex indices  
• vertices (list) – A list of vertex values

size  
The number of vertices in the graph

subgraph(vertexIds)  
Compute the subgraph containing only the corresponding vertexIds and the edges between them.

toIGraph()  
Convert this graph into a igraph Graph object, which requires igraph to be installed. Edge values are stored under the “value” index. Vertices are stored as indices with a “label” value being the corresponding vertex value.

Returns  An igraph Graph object.
Another Python Graph Library Documentation, Release 0.8.1

**toNetworkXGraph()**
Convert this graph into a networkx Graph or DiGraph object, which requires networkx to be installed. Notice that the edge value must be hashable, which is the case for AbstractMatrixGraph subclasses. Edge values are stored under the “value” index. Vertices are stored as indices with a “label” value being the corresponding vertex value.

Returns A networkx Graph or DiGraph object.

**toPySparseGraph()**
Convert the current graph to a PySparseGraph. Currently, vertex labels are not converted.

**toSparseGraph()**
Convert the current graph to a SparseGraph. Currently, vertex labels are not converted.

**undirected** = None

**vertexExists** (vertexId)
Returns true if the vertex with the given name exists, otherwise false.

**vertices** = None

**DictTree**

A tree structure with adjacencies stored in a dictionary based on DictGraph. A tree is essentially a directed graph in which all vertices (except for the root) can only have one incoming edge. The below code is a simple example showing how a tree is created and the output is the number of edges in the tree.

```python
dictTree = DictTree()
dictTree.addEdge("a", "b")
dictTree.addEdge("a", "c")
dictTree.addEdge("d", "a")
print(dictTree.depth())
```

**Methods**

**class** apgl.graph.DictTree.DictTree
Create an empty tree.

**addChild** (parentId, childId, childVertex=None)
This is basically a convenience function to allow one to easily add children.

**addEdge** (vertex1Id, vertex2Id, value=1.0)
Add an edge from vertex1 to vertex2 with a given value.

Parameters
- **vertex1Id** – The parent vertex name
- **vertex2Id** – The child vertex name
- **value** – The value along the edge.

**addEdges** (edgeList, edgeValues=None)
Add a set of edges to the tree.

Parameters
- **edgeList** (list) – A list of pairs of vertex names

4.1. Class Documentation
• edgeValues (list) – A list of corresponding vertex values

adjacencies = None

adjacencyList ()
   Returns an adjacency list representation L of the graph, in which L[i] is the list of all neighbour indices of vertex index i. Furthermore, the method returns W in which W[i] which is the corresponding set of weights. In essence, this method is a way to map vertex ids to indices.

   Returns L  A list whose ith element is a list of neighbours for vertex i.
   Returns W  A list whose ith element is a list of neighbour weights for vertex i.

breadthFirstSearch (root)
   Breadth first search starting from a particular vertex. Returns a list of connected vertices in the order they were found.

   Parameters root (int) – The index of the root vertex.
   Returns  A list of vertices connected to the input one via a path in the graph.

children (vertexId)
   Returns the children of the current vertex. This is the same as neighbours.

   Parameters startVertexId – The vertex id of the parent node.

copy ()
   Return a copied version of this tree. This is a shallow copy in that vertex values are not copied.

cut (d)
   Return a new tree containing all the vertices of the current one up to a depth of d. The edge and vertex labels are copied by reference only.

   Parameters d (int) – The depth of the new cut tree

depthFirstSearch (root)
   Depth first search starting from a particular vertex. Returns a list of connected vertices in the order they were found.

   Parameters root (int) – The index of the root vertex.
   Returns  A list of vertices connected to the input one via a path in the graph.

dijkstrasAlgorithm (vertexId, neighbourLists=None)
   Run Dijkstras Algorithm on the graph for a given source vertex. Returns an array with the distance to all vertices (including itself).
Parameters vertexId – the index of the source vertex.

Returns An array whose ith element is the distance to vertex i.

edgeExists (vertexId1, vertexId2)
Return true if the edge exists between two vertices

findAllDistances (useWeights=True)
Use the repeated calls to Dijkstra’ algorithm to find the shortest path between all pairs of vertices. Note that the shortest path of a vertex to itself is always zero. Returns a matrix whose ij th entry is the shortest path between vertices i and j.

Returns A matrix of shortest paths between all vertices.

findConnectedComponents ()
Finds a list of all connected components of the graph, in order of size with the smallest first.

Returns A list of lists of component indices.

getAllEdgeIndices ()
Returns a numpy array of size (numEdges x 2) of edge index pairs V. The ith row of V, V[i, :], corresponds to an edge from V[i, 0] to V[i, 1]. The corresponding vertex names are found using getAllVertexIds().

getAllEdges ()
Returns a list of tuples of all the edges of this graph.

getAllVertexIds ()
Returns a list of the vertex ids (or names) in this graph.

gEdge (vertex1, vertex2)
Returns the value of the edge between two vertices. If there is no edge between the vertices None is returned.

Parameters

• vertex1 – The name of the first vertex.

• vertex2 – The name of the second vertex.

gNumDirEdges ()
Returns the number of edges, taking this graph as a directed graph.

gNumEdges ()
Returns the total number of edges in graph.

gNumVertices ()
Returns the number of vertices in the graph.

gRoot ()
Return the value of the root vertex.

gRootId ()
Find the id of the root vertex.

gSparseWeightMatrix (format=’lil’)
Returns a weight matrix representation of the graph as a scipy sparse lil_matrix by default. The indices in the matrix correspond to the keys returned by getAllVertexIds. Available formats are: lil for scipy.sparse.lil_matrix, csr for scipy.sparse.csr_matrix, csc for scipy.sparse.csc_matrix, and pysparse for pysparse’s ll_mat.

Parameters format – The format of the sparse matrix.

gVertex (vertexId)
Returns the label of the given vertex, or None if no label.
Another Python Graph Library Documentation, Release 0.8.1

Parameters

- **vertex** – The name of the first vertex.

**getVertices(vertexIndices)**

Takes a list of vertex indices and returns the corresponding vertex values.

Parameters **vertexIndices** (list) – A list of vertex indices

Returns A list of vertices corresponding to the indices

**getWeightMatrix**()

Returns a weight matrix representation of the graph as a numpy array. The indices in the matrix correspond to the keys returned by getAllVertexIds.

**inDegreeSequence**()

Find the in degree sequence. Return the sequence as a vector along with the corresponding vertices in a list.

**isLeaf(vertexId)**

Returns true if the input vertex id is a leaf otherwise false.

Parameters **vertexId** – The vertex id to test

**isNonLeaf(vertexId)**

Returns true if the input vertex id is not a leaf otherwise false.

Parameters **vertexId** – The vertex id to test

**isSubtree(superTree)**

Test if this tree is a subtree of the input tree superTree. This is based on the vertexIds and structure alone.

Parameters **superTree** (apgl.graph.DictTree) – A DictTree object to compare against

**isUndirected**()

Returns true if the current graph is undirected, otherwise false.

**leaves(startVertexId=None)**

Return a list of the vertex ids of all the leaves of this tree. One can specify a starting vertex id (if None, assume the root) and in this case, we return the leaves of the corresponding subtree.

Parameters **startVertexId** – The vertex id of the subtree to find leaves of.

Returns The vertex ids of the leaves.

**neighbourOf(vertex)**

Returns the list of neighbours of the current neighbour.

**neighbours(vertexId)**

Find a list of neighbours of the current vertex. In a directed graph, it is the list of all vertices with an edge from the current vertex.

Parameters **vertexId** – The id of the vertex.

**nonLeaves(startVertexId=None)**

Return a list of the vertex ids of all the non-leaves of this tree.

Parameters **startVertexId** – The vertex id of the parent node.

Returns The vertex ids of the non-leaves.

**outDegreeSequence**()

Find the out degree sequence. Return the sequence as a vector along with the corresponding vertices in a list.

**pruneVertex(vertexId)**

Remove all the descendants of the current vertex.

Parameters **vertexId** – The vertex id of the parent node.
**removeEdge** *(vertex1, vertex2)*

Remove an edge. Does not remove the vertices.

**Parameters**
- **vertex1** – The name of the first vertex.
- **vertex2** – The name of the second vertex.

**removeEdges** *(vertexId1, vertexId2)*

This method is not currently implemented.

**removeVertex** *(vertexId)*

Remove a vertex and all its edges.

**Parameters**
- **vertexId** – The id of the vertex to remove.

**setVertex** *(vertexId, vertex=None)*

Assign a value to a vertex with given name

**Parameters**
- **vertexId** – The id of the vertex.
- **vertex** – The value of the vertex.

**setVertices** *(vertexIndices, vertices)*

Assign new values to the vertices corresponding to vertexIndices

**Parameters**
- **vertexIndices** *(list)* – A list of vertex indices
- **vertices** *(list)* – A list of vertex values

**size**

The number of vertices in the graph

**subgraph** *(vertexIds)*

Compute the subgraph containing only the corresponding vertexIds and the edges between them.

**subtreeAt** *(vertexId)*

Compute the subtree starting from a particular vertex id

**subtreeIds** *(vertexId)*

Return a list of all vertex ids that are descendants of this one, and include this one.

**Parameters**
- **vertexId** – A vertex id

**toIGraph** ()

Convert this graph into a igraph Graph object, which requires igraph to be installed. Edge values are stored under the “value” index. Vertices are stored as indices with a “label” value being the corresponding vertex value.

**Returns**
An igraph Graph object.

**toNetworkXGraph** ()

Convert this graph into a networkx Graph or DiGraph object, which requires networkx to be installed. Notice that the edge value must be hashable, which is the case for AbstractMatrixGraph subclasses. Edge values are stored under the “value” index. Vertices are stored as indices with a “label” value being the corresponding vertex value.

**Returns**
A networkx Graph or DiGraph object.

**toPySparseGraph** ()

Convert the current graph to a PySparseGraph. Currently, vertex labels are not converted.
toSparseGraph()

Convert the current graph to a SparseGraph. Currently, vertex labels are not converted.

undirected = None

vertexExists(vertexId)

Returns true if the vertex with the given name exists, otherwise false.

vertices = None

GeneralVertexList

The GeneralVertexList object represents an indexed list of vertices which may be anything, and is often used in conjunction with either SparseGraph, PySparseGraph or DenseGraph. The following example demonstrates its usage:

```python
from apgl.graph import *
import numpy
	numVertices = 10
	features = 3

tList = GeneralVertexList(numVertices)
tList.setVertex(0, "abc")
tList.setVertex(1, "def")
```

This code creates a GeneralVertexList object, and assigns strings to the first and second vertices.

Methods

class apgl.graph.GeneralVertexList. GeneralVertexList(numVertices)

Create an empty GeneralVertexList with the specified number of features for each vertex (initialised as None) and number of vertices.

Parameters numVertices (int) – The number of vertices.

V = None

addVertices(n)

Adds n vertices to this object.

clearVertex(index)

Sets a vertex to None

Parameters index (int) – the index of the vertex to assign a value.

copy()

Returns a copy of this object.

ext = '.gvl'

getNumVertices()

Returns the number of vertices contained in this object.

getVertex(index)

Returns the value of a vertex.

Parameters index (int) – the index of the vertex.
**getVertices** *(vertexIndices=None)*

Returns a list of vertices specified by vertexIndices, or all vertices if vertexIndices == None.

**Parameters**
- **vertexIndices** *(list)* – a list of vertex indices.

**Returns**
A set of vertices corresponding to the input indices.

**static load** *(filename)*

Load this object from filename.pkl.

**Parameters**
- **filename** *(str)* – The name of the file to load.

**save** *(filename)*

Save this object to filename.nvl.

**Parameters**
- **filename** *(str)* – The name of the file to save to.

**Returns**
The name of the saved file including extension.

**setVertex** *(index, value)*

Set a vertex to the corresponding value.

**Parameters**
- **index** *(int)* – the index of the vertex to assign a value.
- **value** – the value to assign to the vertex.

**setVertices** *(vertices, indices=None)*

Set the vertices to the given list of vertices. If indices = None then all vertices are replaced, and if not the given indices are used.

**Parameters**
- **vertices** *(list)* – a list of vertices.
- **indices** *(list)* – a list of indices of the same length as vertices or None for all indices in this object.

**subList** *(indices)*

Returns a subset of this object, indicated by the given indices.

**PySparseGraph**

The PySparseGraph object represents a graph with an underlying sparse adjacency matrix representation implemented using PySparse. Therefore you must install Pysparse *(http://pysparse.sourceforge.net/)* in order to use this class. PySparseGraph has the advantage of being efficient in memory usage for graphs with few edges, and also relatively fast as Pysparse is implemented using C. Graphs of a 1,000,000 vertices or more can be created with minimal memory overheads. The following is a very simple example of how to use PySparseGraph:

```python
from apgl.graph import PySparseGraph

numVertices = 10

graph = PySparseGraph(numVertices)
graph[0, 2] = 1
graph[0, 3] = 1
graph[2, 1] = 1
graph[2, 5] = 1
graph[2, 6] = 1
graph[6, 9] = 1
```
subgraph = graph.subgraph([0,1,2,3])

graph.vlist[0] = "abc"
graph.vlist[1] = 123

The code creates a new PySparseGraph with 10 vertices, after which edges are added and a subgraph is extracted using vertices 0, 1, 2, and 3. Notice that numpy.array vertices can be added to a PySparseGraph using the VertexList class in the constructor. Finally, the first and second vertices are initialised with “abc” and 123 respectively.

Methods

class apgl.graph.PySparseGraph

Create a PySparseGraph with a given AbstractVertexList or number of vertices, and specify whether it is directed. One can optionally pass in a sparse matrix W which is used as the weight matrix of the graph. Different kinds of sparse matrix can impact the speed of various operations. The currently supported sparse matrix types are: ll_mat.

Parameters

- **vertices** – the initial set of vertices as a AbstractVertexList object, or an int to specify the number of vertices in which case vertices are stored in a GeneralVertexList.
- **undirected** (boolean) – a boolean variable to indicate if the graph is undirected.
- **W** – a square sparse matrix of the same size as the number of vertices, or None to create the default one.
- **sizeHint** (int) – the expected number of edges in the graph for efficient memory usage.

add(graph)

Add the edge weights of the input graph to the current one. Results in a union of the edges.

Parameters

- **graph** (apgl.graph.PySparseGraph) – the input graph.

Returns

A new graph with same vertex list and addition of edge weights

addEdge(vertexIndex1, vertexIndex2, edge=1)

Add a non-zero edge between two vertices.

Parameters

- **vertexIndex1** (int) – The index of the first vertex.
- **vertexIndex2** (int) – The index of the second vertex.
- **edge** (float) – The value of the edge.

addEdges(edgeIndexArray, edgeValues=[]) 

Takes a numpy array of edge index pairs, and edge values and adds them to this graph. The array is 2 dimensional such that each row is a pair of edge indices.

Parameters

- **edgeIndexArray** (numpy.ndarray) – The array of edge indices with each being a pair of indices.
- **edgeValues** (list) – The list of edge values
adjacencyList (useWeights=True)
   Returns an adjacency list representation L of the graph, in which L[i] is the list of all neighbours of vertex i. Furthermore, the method returns W in which W[i] which is the corresponding set of weights.
   Parameters useWeights (bool) – If true uses weights of edges as opposed to just adjacencies.
   Returns L A list whose ith element is a list of neighbours for vertex i.
   Returns W A list whose ith element is a list of neighbour weights/adjacencies for vertex i.

adjacencyMatrix()
   Return the adjacency matrix in numpy.ndarray format. Warning: should not be used unless sufficient memory is available to store the dense matrix.
   Returns The adjacency matrix in dense format

betweenness()
   Return the betweenness of each vertex in the graph. The betweenness is defined as the number of shortest paths passing through each vertex.
   Returns A vector of betweenness values of the same length as the number of vertices in the graph.

breadthFirstSearch (root)
   Breadth first search starting from a particular vertex. Returns a list of connected vertices in the order they were found.
   Parameters root (int) – The index of the root vertex.
   Returns A list of vertices connected to the input one via a path in the graph.

clusteringCoefficient()
   Find the global clustering coefficient of this graph as defined here http://en.wikipedia.org/wiki/Clustering_coefficient
   Returns The clustering coefficient of this graph.

complement()
   Returns a graph with identical vertices (same reference) to the current one, but with the complement of the set of edges. Edges that do not exist have weight 1. This makes a sparse graph dense.
   Returns A new graph with edges complementing the current one.

copy()
   Returns a copy of this object, which also has a copy of the VertexList.

degreeDistribution()
   Return a vector of (out)degree distributions. The ith element of the vector corresponds to the frequency of degree i.
   Returns A vector of (out)degree distributions.

degreeSequence()
   Returns a vector of the degrees (including self edges) for each vertex for an undirected graph.

density()
   The density of the graph is the number of edges/number of possible edges, which does not include self loops. The density of a graph with no vertices is zero.
   Returns The density of the graph.

depthFirstSearch (root)
   Depth first search starting from a particular vertex. Returns a list of connected vertices in the order they were found.
Parameters `root` (int) – The index of the root vertex.

Returns A list of vertices connected to the input one via a path in the graph.

diameter(`useWeights=False, P=None`)  
Finds the diameter of a graph i.e. the longest shortest path. If `useWeights` is True then the weights in the adjacency matrix are used if P is not provided.

Parameters

- `useWeights` (bool) – Whether to use edge weights to compute a diameter.
- `P` (ndarray) – An optional nxn matrix whose ijth entry is the shortest path from i to j.

Returns The diameter of this graph.

diameter2()  
Use Dijkstra's Algorithm to compute the diameter of the graph.

Returns The diameter of the graph.

dijkstrasAlgorithm(`vertexIndex`, `neighbourLists=None`)  
Run Dijkstra's Algorithm on the graph for a given source vertex. The parameter `neighbourLists` is a tuple containing two lists. The first of this lists contains at the ith position all the neighbours of vertex i. The second list contains the corresponding weight on the edge. If `neighbourLists=None`, then it is computed automatically and all edge weights are set to 1. Returns an array with the distance to all vertices (including itself).

Parameters

- `vertexIndex` (int) – the index of the source vertex.
- `neighbourLists` (list) – A tuple of two lists containing vertex adjacencies and edge weights respectively.

Returns An array whose ith element is the distance to vertex i.

effectiveDiameter(q, P=None)  
The effective diameter is the minimum d such that for a fraction q of reachable node pairs, the path length is at most d. This is more robust than the standard diameter method. One can optionally pass in a matrix P whose ijth entry is the shortest path from i to j.

Parameters

- `q` (float) – The fraction of node pairs to consider.
- `P` (ndarray) – An optional nxn matrix whose ijth entry is the shortest path from i to j.

Returns The effective diameter of this graph.

egoGraph(`vertexIndex`)  
Returns the subgraph composed of the given vertex and its immediate neighbours. In the new graph, the ego is index 0 and neighbours are indexed in order after 0.

Parameters `vertexIndex` (int) – the index of the source vertex.

Returns A subgraph of the current one consisting of only immediate neighbours.

findAllDistances(`useWeights=True`)  
Use the repeated calls to Dijkstra's algorithm to find the shortest path between all pairs of vertices. If `useWeights` is true, then the weights are used to compute the path, otherwise adjacencies are used. Note that the shortest path of a vertex to itself is always zero. Returns a matrix whose ijth entry is the shortest path between vertices i and j.

Parameters `useWeights` (bool) – Whether to use the edge weight to compute path cost.
Returns A matrix of shortest paths between all vertices.

**findConnectedComponents()**

Finds a list of all connected components of the graph, in order of size with the smallest first.

Returns A list of lists of component indices.

**findTrees()**

Returns a list of trees for a directed graph. The reason for only allowing directed graphs is that the root of a tree in an undirected graph is ambiguous. Each tree is represented by a list of indices of vertices in the graph.

Returns A list of trees (vertex indices) in the current graph sorted in descending order by size.

**fitPowerLaw()**

Fits the out-degree probabilities of this graph using the power law \( p_d \sim d^{-\gamma} \). The value of \( x_{\text{min}} \) is the point to start taking examples.

Returns \( \alpha \) The power law exponent.

Returns \( ks \) A fit of the power law curve to the data using KS.

Returns \( x_{\text{min}} \) The minimum value of x.

**floydWarshall(useWeights=True)**

Use the Floyd-Warshall algorithm to find the shortest path between all pairs of vertices. If useWeights is true, then the weights are used to compute the path, otherwise adjacencies are used. Note that the shortest path of a vertex to itself is always zero. Returns a matrix whose \( ij \)th entry is the shortest path between vertices \( i \) and \( j \). This algorithm scales as \( O(n^3) \) with the number of vertices \( n \), and is not recommended for very large graphs.

Parameters useWeights (bool) – Whether to use the edge weight to compute path cost.

Returns A matrix of shortest paths between all vertices.

**classmethod fromNetworkXGraph(networkXGraph)**

Take a networkx Graph or DiGraph object, and return a subclass of AbstractMatrixGraph. Notice that networkx must be installed to use this function. The networkXGraph graph dict must have an attribute VListType which is the type of the VertexList used to construct the SparseGraph. Furthermore, only node attributes index by “label” are stored in the VertexList, and edge values are currently ignored.

Returns A networkx Graph or DiGraph object.

**geodesicDistance(P=None, vertexInds=None)**

Compute the mean geodesic distance for a graph. This is denoted for an undirected graph by \( 1/(1/2 n(n+1)) \sum_{(i \leq j)} d_{ij} \) where \( d_{ij} \) is the shortest path length between \( i \) and \( j \). Note that if \( i \) and \( j \) are not connected we assume a path length of 0. If the graph is directed then the geodesic distance is \( 1/(n^2) \sum_{(i, j)} d_{ij} \).

Parameters

- \( P \) (ndarray) – An optional nxn matrix whose \( ij \)th entry is the shortest path from \( i \) to \( j \).
- vertexInds (list) – An optional list of vertices used to compute the mean geodesic distance. If this list is none, then all vertices are used.

Returns The mean geodesic distance of this graph.

**getAllDirEdges()**

Returns the set of directed edges of the current graph as a matrix in which each row corresponds to an edge. For an undirected graph, there is an edge from \( v1 \) to \( v2 \) and from \( v2 \) to \( v1 \) if \( v2! = v1 \).

Returns A matrix with 2 columns, and each row corresponding to an edge.
**getAllEdges()**
Returns the set of edges of the current graph as a matrix in which each row corresponds to an edge. For an undirected graph, \( v1 \geq v2 \).

Returns: A matrix with 2 columns, and each row corresponding to an edge.

**getAllVertexIds()**
Returns a list of all the vertex IDs of this graph.

**getEdge(vertexIndex1, vertexIndex2)**
Get the value of an edge, or None if no edge exists.

Parameters
- `vertexIndex1` (int) – The index of the first vertex.
- `vertexIndex2` (int) – The index of the second vertex.

Returns: The value of the edge between the given vertex indices.

**getEdgeValues(edgeArray)**
Take an array of \( n \times 2 \) of vertex indices and return the corresponding edge values.

Parameters: `edgeArray` (numpy.ndarray) – An array with an edge on each row

Returns: A vector of \( n \) values corresponding to the edge weights of `edgeArray`

**getNumDirEdges()**
Returns the number of edges, taking this graph as a directed graph.

**getNumEdges()**
Returns the total number of edges in this graph.

**getNumVertices()**

Returns: the number of vertices in this graph.

**getVertex(vertexIndex)**
Returns the vertex associated with the given vertex index.

Parameters: `vertexIndex` (int) – the index of the vertex.

Returns: The value of the vertex at the given index.

**getVertexList()**
Returns the AbstractVertexList object of this graph.

**getVertices(vertexIndices)**
Takes a list of vertex indices and returns the corresponding vertex values.

Parameters: `vertexIndices` (list) – A list of vertex indices

Returns: A list of vertices corresponding to the indices.

**getWeightMatrix()**
Return the weight matrix in dense format. Warning: should not be used unless sufficient memory is available to store the dense matrix.

Returns: A numpy.ndarray weight matrix.

**harmonicGeodesicDistance(P=None, vertexInds=None)**
Compute the “harmonic mean” geodesic distance for a graph. This is denoted by the inverse of \( 1/(1/2 \cdot n \cdot (n+1)) \cdot \sum_{i=0}^{n-1} d_{ij}^{-1} \) where \( d_{ij} \) is the shortest path length between \( i \) and \( j \) for an undirected graph. The distance from a node to itself is infinite. For a directed graph, the inverse distance is \( 1/n^2 \cdot \sum_{i,j} d_{ij}^{-1} \).
Parameters

- **P** (ndarray) – An optional nxn matrix whose ijth entry is the shortest path from i to j.
- **vertexInds** (list) – An optional list of vertices used to compute the mean geodesic distance. If this list is none, then all vertices are used.

**Returns** The mean harmonic geodesic distance of this graph.

**hopCount** (*P=None*)

Returns an array such that the ith element is the number of pairs of vertices reachable within i hops. This includes self pairs, and all other pairs are counted twice in the undirected case otherwise once.

**Parameters** **P** (ndarray) – An optional nxn matrix whose ijth entry is the shortest unweighted path from i to j.

**Returns** An array of hop counts.

**inDegreeDistribution**()

Returns a vector of in-degree distributions. The ith element of the vector corresponds to the frequency of degree i.

**Returns** A vector of (in)degree distributions.

**inDegreeSequence**()

Return a vector of the (in)degree sequence for each vertex.

**incidenceMatrix**()

Return the incidence matrix of this graph as a scipy sparse matrix. The incidence matrix X is of size numVertices x numEdges, and has a 1 in element Xij = -1 of edge j leaves vertex i, and Xij = 1 if edge j enters vertex i. Notice that for an undirected graph XX^T is the laplacian matrix.

**intersect** (**graph**)

Take the intersection of the edges of this graph and the input graph. Resulting edge weights are ignored and only adjacencies are stored.

**Parameters** **graph** (apgl.graph.SparseGraph) – the input graph.

**Returns** A new graph with the intersection of edges of the current plus graph.

**isTree**()

Returns true if this graph is a tree. Every vertex must have an in-degree of 1 (i.e. one parent), except the root which has an in-degree of zero and non-zero out-degree.

**Returns** A boolean indicating whether the current graph is a tree.

**isUndirected**()

Returns true if this graph is undirected otherwise false.

**laplacianMatrix** (**outDegree=True**)

Return the Laplacian matrix of this graph, which is defined as L_{ii} = deg(i) L_{ij} = -1 if an edge between i and j, otherwise L_{ij} = 0 . For a directed graph one can specify whether to use the out-degree or in-degree.

**Parameters** **outDegree** (bool) – whether to use the out-degree for the computation of the degree matrix

**Returns** A laplacian adjacency matrix as numpy array.

**laplacianWeightMatrix** (**outDegree=True**)

Return the Laplacian matrix of this graph, L = D - W, where D is the degree matrix and W is the weight matrix. For a directed graph one can specify whether to use the out-degree or in-degree.
Parameters `outDegree` (bool) – whether to use the out-degree for the computation of the degree matrix

Returns A laplacian weight matrix.

classmethod `load` (*filename*)

Load the graph object from the corresponding file. Data is loaded in a zip format as created using save().

Parameters `filename` (str) – The name of the file to load.

Returns A graph corresponding to the one saved in filename.

static `loadMatrix` (*filename*)

maxEigenvector()

Returns the eigenvector of maximum eigenvalue of the adjacency matrix. The eigenvector is of unit length, and measures the centrality of the corresponding vertex. It is based on the principle that connections to high-scoring nodes contribute more to the score of the node in question than equal connections to low-scoring nodes.

Returns The maximum eigenvector of the adjacency matrix.

maxProductPaths()

Find the maximum product paths between all pairs of vertices using a modified version of the Floyd-Warshall algorithm.

Returns A matrix P whose ijth entry corresponds to the maximal product of edge weights between them.

maybeIsomorphicWith (*graph*)

Returns false if graph is definitely not isomorphic with the current graph, however a True may mean the graphs are not isomorphic. Makes a comparison with the eigenvalues of the Laplacian matrices.

Returns True if the current graph is maybe isomorphic with the input one.

multiply (*graph*)

Multiply the edge weights of the input graph to the current one. Results in an intersection of the edges.

Parameters `graph` (apgl.graph.PySparseGraph) – the input graph.

Returns A new graph with edge weights which are multiples of the current and graph

nativeAdjacencyMatrix()

Return the adjacency matrix in sparse format.

neighbourOf (*vertexIndex*)

Return an array of the indices of vertices than have an edge going to the input vertex.

Parameters `vertexIndex` (int) – the index of a vertex.

Returns An array of the indices of all vertices with an edge towards the input vertex.

neighbours (*vertexIndex*)

Return an array of the indices of neighbours. In the case of a directed graph it is an array of those vertices connected by an edge from the current one.

Parameters `vertexIndex` (int) – the index of a vertex.

Returns An array of the indices of all neighbours of the input vertex.

normalisedLaplacianRw (*outDegree=False*)

Compute the normalised random walk laplacian matrix with \( L = I - D^{-1} W \) in which \( W \) is the weight matrix and \( D_{ii} \) is the sum of the ith vertices weights.
Parameters outDegree (bool) – whether to use the out-degree for the computation of the degree matrix

Returns A normalised random-walk laplacian matrix as a numpy array.

normalisedLaplacianSym (outDegree=True)
Compute the normalised symmetric laplacian matrix using \( L = I - D^{-1/2} W D^{-1/2} \), in which \( W \) is the weight matrix and \( D_{ii} \) is the sum of the \( i \)th vertices weights.

Parameters outDegree (bool) – whether to use the out-degree for the computation of the degree matrix

Returns A normalised symmetric laplacian matrix as a numpy array.

outDegreeSequence()
Return a vector of the (out)degree for each vertex.

removeAllEdges()
Removes all edges from this graph.

removeEdge (vertexIndex1, vertexIndex2)
Remove an edge between two vertices.

Parameters
• vertexIndex1 (int) – The index of the first vertex.
• vertexIndex2 (int) – The index of the second vertex.

save (filename)
Save the graph object to the corresponding filename under the .zip extension. The adjacency matrix is stored in matrix market format and the AbstractVertexList decides how to store the vertex labels.

Parameters filename (str) – The name of the file to save.

Returns The name of the saved zip file.

saveMatrix (W, filename)

setDiff (graph)
Find the edges in the current graph which are not present in the input graph.

Parameters graph (apgl.graph.PySparseGraph) – the input graph.

Returns A new graph with edges from the current graph and not in the input graph.

setVertex (vertexIndex, vertex)
Set the vertex with given index to a particular value.

Parameters
• vertexIndex (int) – the index of the vertex.
• vertex – the value of the vertex.

setVertexList (vList)
Assign a new VertexList object to this graph. The number of vertices in the VertexList must be the same as in the graph.

Parameters vList (apgl.graph.AbstractVertexList) – A new subclass of AbstractVertexList to assign to this graph.

setVertices (vertexIndices, vertices)
Assign new values to the vertices corresponding to vertexIndices.

Parameters
• **vertexIndices** (*list*) – A list of vertex indices
• **vertices** (*list*) – A list of vertex values

**setWeightMatrix** (*W*)
Set the weight matrix of this graph. Requires as input an ndarray with the same dimensions as the current weight matrix. Edges are represented by non-zero edges.

**Parameters** W (*ndarray*) – The name of the file to load.

**setWeightMatrixSparse** (*W*)
Set the weight matrix of this graph. Requires as input a scipy sparse matrix with the same dimensions as the current weight matrix. Edges are represented by non-zero edges.

**Parameters** W – The weight matrix to use.

**size**
The number of vertices in the graph

**subgraph** (*vertexIndices*)
Pass in a list or set of vertexIndices and returns the subgraph containing those vertices only, and edges between them.

**Parameters** vertexIndices (*list*) – the indices of the subgraph vertices.

**Returns** A new PySparseGraph containing only vertices and edges from vertexIndices

**toDictGraph** ()
Convert to a DictGraph object. Currently ignores vertex labels.

**Return** graph A DictGraph object.

**toIGraph** ()
Convert this graph into a igraph Graph object, which requires igraph to be installed. Edge values are stored under the “value” index. Vertices are stored as indices with a “label” value being the corresponding vertex value.

**Returns** An igraph Graph object.

**toNetworkXGraph** ()
Convert this graph into a networkx Graph or DiGraph object, which requires networkx to be installed. Edge values are stored under the “value” index. Vertices are stored as indices with a “label” value being the corresponding vertex value. The type of vertex list is stored as a graph attribute under the index “VListType”

**Returns** A networkx Graph or DiGraph object.

**triangleSequence** ()
Computes the number of triangles each vertex participates in using the diagonal of the adjacency matrix. In an undirected graph, a each triangle is counted twice (once for each direction). Note that self loops are not used to form triangles.

**Returns** An array of triangle counts for each vertex.

**undirected** = None

**union** (*graph*)
Take the union of the edges of this graph and the input graph. Resulting edge weights are ignored and only adjacencies are stored.

**Parameters** graph (apgl.graph.SparseGraph) – the input graph.

**Returns** A new graph with the union of edges of the current one.

vList = None
vlist
The vertex list

weightMatrixType()
Returns the type of the sparse matrix used to store edge weights.

**SparseGraph**

The SparseGraph object represents a graph with an underlying sparse weight matrix representation from scipy.sparse. This has the advantage of being efficient in memory usage for graphs with few edges. Graphs of a 1,000,000 vertices or more can be created with minimal memory overheads. The following is a very simple example of how to use SparseGraph:

```python
from apgl.graph import SparseGraph
import numpy

numVertices = 10

graph = SparseGraph(numVertices)
graph.addEdge(0, 1)
#Note can also use the notation e.g. graph[0,1] = 1 to create an edge
graph[0, 2] = 1
graph[0, 3] = 1
graph[2, 1] = 1
graph[2, 5] = 1
graph[2, 6] = 1
graph[6, 9] = 1

subgraph = graph.subgraph([0,1,2,3])

graph.vlist[0] = "abc"
graph.vlist[1] = 123

print(graph.size)
```

The code creates a new SparseGraph with 10 vertices, after which edges are added and a subgraph is extracted using vertices 0, 1, 2, and 3. Notice that numpy.array vertices can be added to a SparseGraph using the VertexList class in the constructor. Finally, the first and second vertices are initialised with “abc” and 123 respectively.

In order to speed up certain operations on the graph, SparseGraph can be initialised with an empty sparse matrix of several types. For example, the csr_matrix allows fast out-degree computations whereas csc_matrix is faster for finding in-degrees of directed graphs.

```python
from apgl.graph import SparseGraph
import numpy
import scipy.sparse

numVertices = 10

weightMatrix = scipy.sparse.lil_matrix((numVertices, numVertices))
graph = SparseGraph(numVertices, W=weightMatrix)
graph[0, 1] = 1
graph[0, 2] = 1

#Output the number of vertices
print(graph.size)
```

Here, the SparseGraph is initialised with 10 vertices and the sparse matrix weightMatrix passed to the constructor is used to store edge weights.
Methods

class `apgl.graph.SparseGraph.SparseGraph`(vertices, undirected=True, W=None, dtype='float', frmt='csr')

Bases: `apgl.graph.AbstractMatrixGraph.AbstractMatrixGraph`

Create a SparseGraph with a given AbstractVertexList or number of vertices, and specify whether it is directed. One can optionally pass in a sparse matrix W which is used as the weight matrix of the graph. Different kinds of sparse matrix can impact the speed of various operations. The currently supported sparse matrix types are: lil_matrix, csr_matrix, csc_matrix and dok_matrix. The default sparse matrix is csr_matrix.

Parameters

- vertices – the initial set of vertices as a AbstractVertexList object, or an int to specify the number of vertices in which case vertices are stored in a GeneralVertexList.
- undirected (boolean) – a boolean variable to indicate if the graph is undirected.
- W – a square sparse matrix of the same size as the number of vertices, or None to create the default one.
- dtype – the data type of the sparse matrix if W is not specified.
- frmt – the format of the sparse matrix: lil, csr or csc if W is not specified

`W = None`

add (graph)

Add the edge weights of the input graph to the current one. Results in a union of the edges.

Parameters graph (apgl.graph.SparseGraph) – the input graph.

Returns A new graph with same vertex list and addition of edge weights

addEdge (vertexIndex1, vertexIndex2, edge=1)

Add a non-zero edge between two vertices.

Parameters

- vertexIndex1 (int) – The index of the first vertex.
- vertexIndex2 (int) – The index of the second vertex.
- edge (float) – The value of the edge.

addEdges (edgeIndexArray, edgeValues=[ ])

Takes a numpy array of edge index pairs, and edge values and adds them to this graph. The array is 2 dimensional such that each row is a pair of edge indices.

Parameters

- edgeIndexArray (numpy.ndarray) – The array of edge indices with each being a pair of indices.
- edgeValues (list) – The list of edge values

adjacencyList (useWeights=True)

Returns an adjacency list representation L of the graph, in which L[i] is the list of all neighbours of vertex i. Furthermore, the method returns W in which W[i] which is the corresponding set of weights.

Parameters useWeights (bool) – If true uses weights of edges as opposed to just adjacencies.

Returns L A list whose ith element is a list of neighbours for vertex i.

Returns W A list whose ith element is a list of neighbour weights/adjacencies for vertex i.
adjacencyMatrix()  
Return the adjacency matrix in numpy.ndarray format. Warning: should not be used unless sufficient  
memory is available to store the dense matrix.  

Returns  The adjacency matrix in dense format

betweenness()  
Return the betweenness of each vertex in the graph. The betweenness is defined as the number of shortest  
paths passing through each vertex.  

Returns  A vector of betweenness values of the same length as the number of vertices in the  
graph.

breadthFirstSearch(root)  
Breadth first search starting from a particular vertex. Returns a list of connected vertices in the order they  
were found.  

Parameters  root (int) – The index of the root vertex.  

Returns  A list of vertices connected to the input one via a path in the graph.

clusteringCoefficient()  
Find the global clustering coefficient of this graph as defined here  
http://en.wikipedia.org/wiki/Clustering_coefficient  

Returns  The clustering coefficient of this graph.

complement()  
Returns a graph with identical vertices (same reference) to the current one, but with the complement of the  
set of edges. Edges that do not exist have weight 1. This makes a sparse graph dense.  

Returns  A new graph with edges complimenting the current one.

concat(graph)  
Take a new graph and concatenate it to the current one. Returns a new graph of the concatenated graphs  
with this graphs vertices first in the new list of vertices.  

Parameters  graph (apgl.graph.SparseGraph) – the input graph.

copy()  
Returns a copy of this object, which also has a copy of the AbstractVertexList.

degreeDistribution()  
Return a vector of (out)degree distributions. The ith element of the vector corresponds to the frequency of  
degree i.  

Returns  A vector of (out)degree distributions.

degreeSequence()  
Returns  a vector of the degrees (including self edges) for each vertex for an undirected graph.

density()  
The density of the graph is the number of edges/number of possible edges, which does not include self  
loops. The density of a graph with no vertices is zero.  

Returns  The density of the graph.

depthFirstSearch(root)  
Depth first search starting from a particular vertex. Returns a list of connected vertices in the order they  
were found.  

Parameters  root (int) – The index of the root vertex.  

Returns  A list of vertices connected to the input one via a path in the graph.
diameter (useWeights=False, P=None)
Finds the diameter of a graph i.e. the longest shortest path. If useWeights is True then the weights in the adjacency matrix are used if P is not provided.

Parameters
- useWeights (bool) – Whether to use edge weights to compute a diameter.
- P (ndarray) – An optional nxn matrix whose ijth entry is the shortest path from i to j.

Returns The diameter of this graph.

diameter2 ()
Use Dijkstra’s Algorithm to compute the diameter of the graph.

Returns The diameter of the graph.

dijkstraAlgorithm (vertexIndex, neighbourLists=None)
Run Dijkstra’s Algorithm on the graph for a given source vertex. The parameter neighbourLists is a tuple containing two lists. The first of this lists contains at the ith position all the neighbours of vertex i. The second list contains the corresponding weight on the edge. If neighbourLists=None, then it is computed automatically and all edge weights are set to 1. Returns an array with the distance to all vertices (including itself).

Parameters
- vertexIndex (int) – the index of the source vertex.
- neighbourLists (list) – A tuple of two lists containing vertex adjacencies and edge weights respectively.

Returns An array whose ith element is the distance to vertex i.

effectiveDiameter (q, P=None)
The effective diameter is the minimum d such that for a fraction q of reachable node pairs, the path length is at most d. This is more robust than the standard diameter method. One can optionally pass in a matrix P whose ijth entry is the shortest path from i to j.

Parameters
- q (float) – The fraction of node pairs to consider.
- P (ndarray) – An optional nxn matrix whose ijth entry is the shortest path from i to j.

Returns The effective diameter of this graph.

egoGraph (vertexIndex)
Returns the subgraph composed of the given vertex and its immediate neighbours. In the new graph, the ego is index 0 and neighbours are indexed in order after 0.

Parameters vertexIndex (int) – the index of the source vertex.

Returns A subgraph of the current one consisting of only immediate neighbours.

findAllDistances (useWeights=True)
Use the repeated calls to Dijkstra’ algorithm to find the shortest path between all pairs of vertices. If useWeights is true, then the weights are used to compute the path, otherwise adjacencies are used. Note that the shortest path of a vertex to itself is always zero. Returns a matrix whose ijth entry is the shortest path between vertices i and j.

Parameters useWeights (bool) – Whether to use the edge weight to compute path cost.

Returns A matrix of shortest paths between all vertices.
findConnectedComponents ()
Finds a list of all connected components of the graph, in order of size with the smallest first.

Returns A list of lists of component indices.

findTrees ()
Returns a list of trees for a directed graph. The reason for only allowing directed graphs is that the root of a tree in an undirected graph is ambiguous. Each tree is represented by an list of indices of vertices in the graph.

Returns A list of trees (vertex indices) in the current graph sorted in descending order by size.

fitPowerLaw ()
Fits the out-degree probabilities of this graph using the power law \( p_d \sim d^{-\gamma} \). The value of xmin is the point to start taking examples.

Returns alpha The power law exponent.
Returns ks A fit of the power law curve to the data using KS.
Returns xmin The minimum value of x.

floydWarshall (useWeights=True)
Use the Floyd-Warshall algorithm to find the shortest path between all pairs of vertices. If useWeights is true, then the weights are used to compute the path, otherwise adjacencies are used. Note that the shortest path of a vertex to itself is always zero. Returns a matrix whose ij th entry is the shortest path between vertices i and j. This algorithm scales as \( O(n^3) \) with the number of vertices n, and is not recommended for very large graphs.

Parameters useWeights (bool) – Whether to use the edge weight to compute path cost.

Returns A matrix of shortest paths between all vertices.

classmethod fromNetworkXGraph (networkXGraph)
Take a networkx Graph or DiGraph object, and return a subclass of AbstractMatrixGraph. Notice that networkx must be installed to use this function. The networkXGraph graph dict must have an attribute VListType which is the type of the VertexList used to construct the SparseGraph. Furthermore, only node attributes index by “label” are stored in the VertexList, and edge values are currently ignored.

Returns A networkx Graph or DiGraph object.

geodesicDistance (P=None, vertexInds=None)
Compute the mean geodesic distance for a graph. This is denoted for an undirected graph by \( 1/(1/2 \, n(n+1)) \) \sum_{i<=j} d_{ij} \) where \( d_{ij} \) is the shortest path length between i and j. Note that if i and j are not connected we assume a path length of 0. If the graph is directed then the geodesic distance is \( 1/(n^2) \sum_{i, j} d_{ij} \).

Parameters
- P (ndarray) – An optional nxn matrix whose ijth entry is the shortest path from i to j.
- vertexInds (list) – An optional list of vertices used to compute the mean geodesic distance. If this list is none, then all vertices are used.

Returns The mean geodesic distance of this graph.

g getAllDirEdges ()
Returns the set of directed edges of the current graph as a matrix in which each row corresponds to an edge. For an undirected graph, there is an edge from v1 to v2 and from v2 to v1 if v2!=v1.

Returns A matrix with 2 columns, and each row corresponding to an edge.

g getAllEdges ()
Returns the set of edges of the current graph as a matrix in which each row corresponds to an edge. For an undirected graph, v1>=v2.
Returns A matrix with 2 columns, and each row corresponding to an edge.

`getAllVertexIds()`
Returns a list of all the vertex IDs of this graph.

`getEdge(vertexIndex1, vertexIndex2)`
Get the value of an edge, or None if no edge exists.

**Parameters**

- `vertexIndex1` (int) – The index of the first vertex.
- `vertexIndex2` (int) – The index of the second vertex.

**Returns** The value of the edge between the given vertex indices.

`getEdgeValues(edgeArray)`
Take an array of n x 2 of vertex indices and return the corresponding edge values.

**Parameters** `edgeArray` (numpy.ndarray) – An array with an edge on each row

**Returns** A vector of n values corresponding to the edge weights of edgeArray

`getNumDirEdges()`
Returns the number of edges, taking this graph as a directed graph.

`getNumEdges()`
Returns the total number of edges in this graph.

`getNumVertices()`
Returns the number of vertices in this graph.

`getSparseWeightMatrix()`
Returns the original sparse weight matrix.

**Returns** A scipy.sparse weight matrix.

`getVertex(vertexIndex)`
Returns the vertex associated with the given vertex index.

**Parameters** `vertexIndex` (int) – the index of the vertex.

**Returns** The value of the vertex at the given index.

`getVertexList()`
Returns the AbstractVertexList object of this graph.

`getVertices(vertexIndices)`
Takes a list of vertex indices and returns the corresponding vertex values.

**Parameters** `vertexIndices` (list) – A list of vertex indices

**Returns** A list of vertices corresponding to the indices

`getWeightMatrix()`
Return the weight matrix in dense format. Warning: should not be used unless sufficient memory is available to store the dense matrix.

**Returns** A numpy.ndarray weight matrix.

`harmonicGeodesicDistance(P=None, vertexInds=None)`
Compute the “harmonic mean” geodesic distance for a graph. This is denoted by the inverse of \(1/(1/2 n(n+1)) \sum_{i<j} d_{ij}^{-1}\) where \(d_{ij}\) is the shortest path length between \(i\) and \(j\) for an undirected graph.
The distance from a node to itself is infinite. For a directed graph, the inverse distance is \(1/n^2 \sum_{i,j} d_{ij}^{-1}\).

**Parameters**

- **P** (*ndarray*) – An optional nxn matrix whose \(ij\)th entry is the shortest path from \(i\) to \(j\).

- **vertexInds** (*list*) – An optional list of vertices used to compute the mean geodesic distance. If this list is none, then all vertices are used.

**Returns** The mean harmonic geodesic distance of this graph.

**hopCount** (*P=None*)

Returns an array such that the \(i\)th element is the number of pairs of vertices reachable within \(i\) hops. This includes self pairs, and all other pairs are counted twice in the undirected case otherwise once.

**Parameters**

- **P** (*ndarray*) – An optional nxn matrix whose \(ij\)th entry is the shortest unweighted path from \(i\) to \(j\).

**Returns** An array of hop counts.

**inDegreeDistribution**()

Returns a vector of in-degree distributions. The \(i\)th element of the vector corresponds to the frequency of degree \(i\).

**Returns** A vector of (in)degree distributions.

**inDegreeSequence**()

**Returns** a vector of the (in)degree sequence for each vertex.

**incidenceMatrix**()

Return the incidence matrix of this graph as a scipy sparse matrix. The incidence matrix \(X\) is of size \(\text{numVertices} \times \text{numEdges}\), and has a 1 in element \(X_{ij} = -1\) if an edge between \(i\) and \(j\), otherwise \(X_{ij} = 1\) if edge \(j\) enters vertex \(i\). Notice that for an undirected graph \(XX^T\) is the laplacian matrix.

**intersect** (*graph*)

Take the intersection of the edges of this graph and the input graph. Resulting edge weights are ignored and only adjacencies are stored.

**Parameters**

- **graph** (*apgl.graph.SparseGraph*) – the input graph.

**Returns** A new graph with the intersection of edges of the current plus graph.

**isTree**()

Returns true if this graph is a tree. Every vertex must have an in-degree of 1 (i.e. one parent), except the root which has an in-degree of zero and non-zero out-degree.

**Returns** A boolean indicating whether the current graph is a tree.

**isUndirected**()

Returns true if this graph is undirected otherwise false.

**laplacianMatrix** (*outDegree=True, sparse=False*)

Return the Laplacian matrix of this graph, which is defined as \(\mathcal{L}_{ii} = \text{deg}(i)\) \(\mathcal{L}_{ij} = -1\) if an edge between \(i\) and \(j\), otherwise \(\mathcal{L}_{ij} = 0\). For a directed graph one can specify whether to use the out-degree or in-degree.

**Parameters**

- **outDegree** (*bool*) – whether to use the out-degree for the computation of the degree matrix

- **sparse** (*bool*) – whether to return a sparse matrix or numpy array
Returns A laplacian adjacency matrix.

\( \text{laplacianWeightMatrix}(\text{outDegree=\text{True}}) \)

Return the Laplacian matrix of this graph, \( L = D - W \), where \( D \) is the degree matrix and \( W \) is the weight matrix. For a directed graph one can specify whether to use the out-degree or in-degree.

**Parameters**
- **outDegree** (bool) – whether to use the out-degree for the computation of the degree matrix

**Returns** A laplacian weight matrix.

classmethod \( \text{load}(\text{filename}) \)

Load the graph object from the corresponding file. Data is loaded in a zip format as created using save().

**Parameters**
- **filename** (str) – The name of the file to load.

**Returns** A graph corresponding to the one saved in filename.

\( \text{static loadMatrix}(\text{filename}) \)

**maxEigenvector()**

Returns the eigenvector of maximum eigenvalue of the adjacency matrix. The eigenvector is of unit length, and measures the centrality of the corresponding vertex. It is based on the principle that connections to high-scoring nodes contribute more to the score of the node in question than equal connections to low-scoring nodes.

**Returns** The maximum eigenvector of the adjacency matrix.

\( \text{maxProductPaths()} \)

Find the maximum product paths between all pairs of vertices using a modified version of the Floyd-Warshall algorithm.

**Returns** A matrix \( P \) whose \( i,j \)th entry corresponds to the maximal product of edge weights between them.

\( \text{maybeIsomorphicWith}(\text{graph}) \)

Returns false if graph is definitely not isomorphic with the current graph, however a True may mean the graphs are not isomorphic. Makes a comparison with the eigenvalues of the Laplacian matrices.

**Returns** True if the current graph is maybe isomorphic with the input one.

\( \text{multiply}(\text{graph}) \)

Multiply the edge weights of the input graph to the current one. Results in an intersection of the edges.

**Parameters**
- **graph** (apgl.graph.SparseGraph) – the input graph.

**Returns** A new graph with edge weights which are multiples of the current and graph

\( \text{nativeAdjacencyMatrix()} \)

**Returns** the adjacency matrix in the native sparse format.

\( \text{neighbourOf}(\text{vertexIndex}) \)

Return an array of the indices of vertices than have an edge going to the input vertex.

**Parameters**
- **vertexIndex** (int) – the index of a vertex.

**Returns** An array of the indices of all vertices with an edge towards the input vertex.

\( \text{neighbours}(\text{vertexIndex}) \)

Return an array of the indices of neighbours. In the case of a directed graph it is an array of those vertices connected by an edge from the current one.

**Parameters**
- **vertexIndex** (int) – the index of a vertex.

**Returns** An array of the indices of all neighbours of the input vertex.
normalisedLaplacianRw (outDegree=True)

- **Parameters**: outDegree (bool) – whether to use the out-degree for the computation of the degree matrix
- **Returns**: A normalised random walk laplacian matrix as a numpy array.

normalisedLaplacianSym (outDegree=True, sparse=False)

- **Parameters**
  - outDegree (bool) – whether to use the out-degree for the computation of the degree matrix
  - sparse (bool) – whether to return a sparse matrix or numpy array
- **Returns**: A normalised symmetric laplacian matrix

outDegreeSequence()

- **Returns**: a vector of the (out)degree sequence for each vertex.

removeAllEdges()

- **Removes all edges from this graph.**

removeEdge (vertexIndex1, vertexIndex2)

- **Remove an edge between two vertices.**
  - **Parameters**
    - vertexIndex1 (int) – The index of the first vertex.
    - vertexIndex2 (int) – The index of the second vertex.

save (filename)

- **Save the graph object to the corresponding filename under the .zip extension.** The adjacency matrix is stored in matrix market format and the AbstractVertexList decides how to store the vertex labels.
  - **Parameters**: filename (str) – The name of the file to save.
  - **Returns**: The name of the saved zip file.

saveMatrix (W, filename)

setDiff (graph)

- **Find the edges in the current graph which are not present in the input graph.**
  - **Parameters**: graph (apgl.graph.SparseGraph) – the input graph.
  - **Returns**: A new graph with edges from the current graph and not in the input graph.

setVertex (vertexIndex, vertex)

- **Set the vertex with given index to a particular value.**
  - **Parameters**
    - vertexIndex (int) – the index of the vertex.
    - vertex – the value of the vertex.

setVertexList (vList)

- **Assign a new VertexList object to this graph.** The number of vertices in the VertexList must be the same as in the graph.
Another Python Graph Library Documentation, Release 0.8.1

Parameters vList (apgl.graph.AbstractVertexList) – A new subclass of AbstractVertexList to assign to this graph.

**setVertices** (vertexIndices, vertices)
Assign new values to the vertices corresponding to vertexIndices

Parameters

- vertexIndices (list) – A list of vertex indices
- vertices (list) – A list of vertex values

**setWeightMatrix** (W)
Set the weight matrix of this graph. Requires as input an ndarray or a scipy sparse matrix with the same dimensions as the current weight matrix. Edges are represented by non-zero edges.

Parameters

- W (ndarray or scipy.sparse matrix) – The weight matrix to use.

**setWeightMatrixSparse** (W)
Set the weight matrix of this graph. Requires as input a scipy sparse matrix with the same dimensions as the current weight matrix. Edges are represented by non-zero edges.

Parameters

- W – The weight matrix to use.

**size**
The number of vertices in the graph

**subgraph** (vertexIndices)
Pass in a list or set of vertexIndices and returns the subgraph containing those vertices only, and edges between them. The subgraph indices correspond to the sorted input indices.

Parameters

- vertexIndices (list) – the indices of the subgraph vertices.

Returns

A new SparseGraph containing only vertices and edges from vertexIndices

**toCsc**()
Convert the internal matrix representation to csc format (compressed sparse column) in order to improve the efficiency of certain operations.

**toCsr**()
Convert the internal matrix representation to csr format (compressed sparse row) in order to improve the efficiency of certain operations.

**toDictGraph**()
Convert to a DictGraph object. Currently ignores vertex labels.

Returns

A DictGraph object.

**toIGraph**()
Convert this graph into a igraph Graph object, which requires igraph to be installed. Edge values are stored under the “value” index. Vertices are stored as indices with a “label” value being the corresponding vertex value.

Returns

An igraph Graph object.

**toNetworkXGraph**()
Convert this graph into a networkx Graph or DiGraph object, which requires networkx to be installed. Edge values are stored under the “value” index. Vertices are stored as indices with a “label” value being the corresponding vertex value. The type of vertex list is stored as a graph attribute under the index “VListType”

Returns

A networkx Graph or DiGraph object.
triangleSequence()
Comes the number of triangles each vertex participates in using the diagonal of the adjacency matrix. In an undirected graph, a each triangle is counted twice (once for each direction). Note that self loops are not used to form triangles.

Returns An array of triangle counts for each vertex.

undirected = None

union (graph)
Take the union of the edges of this graph and the input graph. Resulting edge weights are ignored and only adjacencies are stored.

Parameters graph (apgl.graph.SparseGraph) – the input graph.

Returns A new graph with the union of edges of the current one.

vList = None

vlist
The vertex list

weightMatrixType()

Returns the type of the sparse matrix used to store edge weights.

VertexList

The VertexList object is a indexed set of vectorial vertices, and is often used in conjunction with either SparseGraph, SparseMultiGraph or DenseGraph. The following example demonstrates its usage:

from apgl.graph import *
import numpy

numVertices = 10
numFeatures = 3

vList = VertexList(numVertices, numFeatures)
value = numpy.array([0.1, 0.5, 0.1])
vList.setVertex(0, value)

This code creates a VertexList object, and assigns a value to the first vertex.

Methods

class apgl.graph.VertexList.VertexList (numVertices, numFeatures=0, dtype=<type ‘float’>)

Create an empty (zeroed) VertexList with the specified number of features for each vertex and number of vertices.

Parameters

• numVertices (int) – The number of vertices.
• numFeatures (int) – The number of features for each vertex.
• dtype – the data type for the vertex matrix, e.g numpy.int8.

V = None
addVertices \((n)\)
Adds \(n\) vertices to this object.

clearVertex \((\text{index})\)
Sets a vertex to the all-zeros array.

Parameters index \((\text{int})\) – the index of the vertex to assign a value.

copy ()
Returns a copy of this object.

ext = ‘.nvl’

getFeatureDistribution \((f\text{Index}, v\text{Indices}={\text{None}})\)
Returns a tuple \((\text{frequencies, items})\) about a particular feature given by \(f\text{Index}\). This method is deprecated.

getNumFeatures ()
Returns the number of features of the vertices of this object.

getNumVertices ()
Returns the number of vertices contained in this object.

getVertex \((\text{index})\)
Returns the value of a vertex.

Parameters index \((\text{int})\) – the index of the vertex.

Returns the value of the vertex.

getVertices \((\text{vertexIndices}={\text{None}})\)
Returns a set of vertices specified by vertexIndices. If vertexIndices is None then all vertices are returned.

Parameters vertexIndices \((\text{list})\) – a list of vertex indices.

Returns A set of vertices corresponding to the input indices.

static load \((\text{filename})\)
Load this object from \text{filename.nvl}.

Parameters filename \((\text{str})\) – The name of the file to load.

replaceVertices \((\text{vertices})\)
Replace all the vertices within this class with a new set. Must have the same number vertices, but can alter the number of features.

Parameters vertices \((\text{numpy.ndarray})\) – a set of vertices of the same number of rows as this object.

save \((\text{filename})\)
Save this object to \text{filename.nvl}.

Parameters filename \((\text{str})\) – The name of the file to save.

Returns The name of the saved file including extension.

setVertex \((\text{index, value})\)
Set a vertex to the corresponding value.

Parameters

• index \((\text{int})\) – the index of the vertex to assign a value.

• value \((\text{numpy.ndarray})\) – the value to assign to the vertex.

setVertices \((\text{vertices})\)
Set the vertices to the given numpy array.
**Parameters**

vertices (numpy.ndarray) – a set of vertices of the same shape as this object.

**subList** (indices)

Returns a subset of this object, indicated by the given indices.

### 4.1.3 Input/Output

This is a set of classes to read and write graphs to disk.

**SimpleGraphReader**

Methods

```python
class apgl.io.SimpleGraphReader
    A class to read SimpleGraph files.

readFromFile (fileName)
    Read vertices and edges of the graph from the given file name. The file must have as its first line “Vertices” followed by a list of vertex indices (one per line). Then the lines following “Arcs” or “Edges” have a list of pairs of vertex indices represented directed or undirected edges.
```

**SimpleGraphWriter**

Methods

```python
class apgl.io.SimpleGraphWriter
    A class to output all edges of a graph in a simple text format

writeToFile (fileName, graph)
    Write vertices and edges of the graph in a text format to the given file name. The file has a first line “Vertices” followed by a list of vertex indices (one per line). Then the lines following “Arcs” or “Edges” have a list of pairs of vertex indices represented directed or undirected edges.
```

### 4.1.4 Utility Classes

This package has a number of utility classes which are used to aid the main code.

**Parameter**

The Parameter class contains a set of static methods which are useful for type checking. For example

```python
from apgl.util.Parameter import Parameter

i = 5
j = 12
min = 0
max = 10

#Parameter i checked as int and found to be within min and max
Parameter.checkInt(i, min, max)
```
Another Python Graph Library Documentation, Release 0.8.1

A ValueError is raised as j is greater than max
Parameter.checkInt(j, min, max)

Methods

class apgl.util.Parameter
   A class of static methods which are useful for type checking.

   a = 1
   b = 2

   static checkArray(array, softCheck=False, arrayInfo='')
   Check that an array contains no nan or inf values

   static checkBoolean(val, softCheck=False)
   Check if val is a boolean and raise a ValueError if it is not.

       Parameters val (bool) – The value to test.

   static checkClass(obj, objectType, softCheck=False)
   Check if an object belongs to a particular class and raise a ValueError if it does not.

       Parameters objectType – The object to test.

   static checkFloat(i, min, max, softCheck=False)
   Check if i is an floating point value between min and max inclusive and raise a ValueError if it is not. If one requires an open ended range, then min or max can be float('inf') for example.

       Parameters

           • i (float) – The value to test, such that min <= i <= max.

           • min (float) – The minimum value of i.

           • max (float) – The maximum value of i.

   static checkIndex(i, min, max, softCheck=False)
   Check if i is an integer value between min inclusive and max exclusive and raise a ValueError if it is not. If one requires an open ended range, then min or max can be float('inf') for example.

       Parameters

           • i (int) – The value to test, such that min <= i < max.

           • min (int) – The minimum value of i.

           • max (int) – The maximum+1 value of i.

   static checkInt(i, min, max, softCheck=False)
   Check if i is an integer value between min and max inclusive and raise a ValueError if it is not. If one requires an open ended range, then min or max can be float('inf') for example.

       Parameters

           • i (int) – The value to test, such that min <= i <= max.

           • min (int) – The minimum value of i.

           • max (int) – The maximum value of i.

   static checkList(lst, func, params, softCheck=False)
   Check if a list/ndarray obeys the constaints given by func. For example, in order to check if a list/ndarray lst contains integers between 0 and 10, one can use Parameter.checkList(lst, Parameter.checkInt, [0, 10]).
The first argument of checkInt is taken from the input list and the remaining ones correspond to the 3rd parameter of checkList. If an array is used as input, then it must be 1D.

**Parameters**

- **lst (list)** – The list to test.
- **func** – A function which raises an exception when an invalid list entry is encountered, otherwise does nothing.
- **params** – A list of parameter to the checking function.

**static checkOrthogonal (A, tol=1e-06, softCheck=False, investigate=False, arrayInfo=’?’)**

Takes as input a matrix A and checks if it is orthogonal by verifying whether \( \|A*A.T - Id\|_F < tol \).

**static checkString (s, strList, softCheck=False)**

Check if s is an string in strList and raise a ValueError if it is not.

**Parameters**

- **s (str)** – The string to test.
- **strList (list)** – A list of valid strings.

**static checkSymmetric (A, tol=1e-06, softCheck=False, arrayInfo=’?’)**

Takes as input a matrix A and checks if it is symmetric by verifying whether \( \|A - A.T\|_F < tol \).

**floatTypes = [<type ‘float’>, <type ‘numpy.float32’>, <type ‘numpy.float64’>]**

**intTypes = [<type ‘int’>, <type ‘numpy.int8’>, <type ‘numpy.int16’>, <type ‘numpy.int32’>, <type ‘numpy.int64’>]**

**static whatToDo (msg, softCheck=False)**

There is also a PDF version.
CHAPTER

FIVE

SUPPORT

For any questions or comments please email me at <my first name> at gmail dot com, or post on the mailing list.
INDICES AND TABLES

- genindex
- modindex
- search
INDEX

A
a (apgl.util.Parameter attribute), 66
add() (apgl.graph.CsArrayGraph.CsArrayGraph method), 15
add() (apgl.graph.DenseGraph.DenseGraph method), 24
add() (apgl.graph.PySparseGraph.PySparseGraph method), 44
add() (apgl.graph.SparseGraph.SparseGraph method), 54
addChild() (apgl.graph.DictTree.DictTree method), 37
addEdge() (apgl.graph.CsArrayGraph.CsArrayGraph method), 15
addEdge() (apgl.graph.DenseGraph.DenseGraph method), 25
addEdge() (apgl.graph.DictGraph.DictGraph method), 33
addEdge() (apgl.graph.DictTree.DictTree method), 37
addEdge() (apgl.graph.PySparseGraph.PySparseGraph method), 44
addEdge() (apgl.graph.SparseGraph.SparseGraph method), 54
addEdges() (apgl.graph.CsArrayGraph.CsArrayGraph method), 16
addEdges() (apgl.graph.DenseGraph.DenseGraph method), 25
addEdges() (apgl.graph.DictGraph.DictGraph method), 34
addEdges() (apgl.graph.DictTree.DictTree method), 37
addEdges() (apgl.graph.PySparseGraph.PySparseGraph method), 44
addEdges() (apgl.graph.SparseGraph.SparseGraph method), 54
addVertices() (apgl.graph.CsArrayGraph.CsArrayGraph method), 16
addVertices() (apgl.graph.DictGraph.DictGraph method), 34
addVertices() (apgl.graph.DictTree.DictTree method), 38
addVertices() (apgl.graph.PySparseGraph.PySparseGraph method), 45
addVertices() (apgl.graph.SparseGraph.SparseGraph method), 54
adjacencyList() (apgl.graph.DenseGraph.DenseGraph method), 25
adjacencyList() (apgl.graph.DictGraph.DictGraph method), 34
adjacencyList() (apgl.graph.DictTree.DictTree method), 38
adjacencyList() (apgl.graph.PySparseGraph.PySparseGraph method), 44
adjacencyList() (apgl.graph.SparseGraph.SparseGraph method), 54
adjacencyMatrix() (apgl.graph.CsArrayGraph.CsArrayGraph method), 16
adjacencyMatrix() (apgl.graph.DenseGraph.DenseGraph method), 25
adjacencyMatrix() (apgl.graph.DictGraph.DictGraph method), 35
adjacencyMatrix() (apgl.graph.DictTree.DictTree method), 38
adjacencyMatrix() (apgl.graph.PySparseGraph.PySparseGraph method), 45
adjacencyMatrix() (apgl.graph.SparseGraph.SparseGraph method), 54

B
b (apgl.util.Parameter attribute), 66
BarabasiAlbertGenerator (class in apgl.generator.BarabasiAlbertGenerator), 10
betweenness() (apgl.graph.CsArrayGraph.CsArrayGraph method), 16
betweenness() (apgl.graph.DenseGraph.DenseGraph method), 25
betweenness() (apgl.graph.PySparseGraph.PySparseGraph method), 45
betweenness() (apgl.graph.SparseGraph.SparseGraph method), 55
breadthFirstSearch() (apgl.graph.CsArrayGraph.CsArrayGraph method), 16
breadthFirstSearch() (apgl.graph.DenseGraph.DenseGraph method), 25
breadthFirstSearch() (apgl.graph.DictGraph.DictGraph method), 34
breadthFirstSearch() (apgl.graph.DictTree.DictTree method), 38
breadthFirstSearch() (apgl.graph.PySparseGraph.PySparseGraph method), 45
breadthFirstSearch() (apgl.graph.SparseGraph.SparseGraph method), 54
breadthFirstSearch() (apgl.graph.SparseGraph.SparseGraph CsArrayGraph (class in apgl.graph.CsArrayGraph), 15
cut() (apgl.graph.DictTree.DictTree method), 38

C
checkArray() (apgl.util.Parameter static method), 66
checkBoolean() (apgl.util.Parameter static method), 66
checkClass() (apgl.util.Parameter static method), 66
checkFloat() (apgl.util.Parameter static method), 66
checkIndex() (apgl.util.Parameter static method), 66
checkList() (apgl.util.Parameter static method), 66
checkOrthogonal() (apgl.util.Parameter static method), 67
checkString() (apgl.util.Parameter static method), 67
checkSymmetric() (apgl.util.Parameter static method), 67
children() (apgl.graph.DictTree.DictTree method), 38
clearVertex() (apgl.graph.GeneralVertexList GeneralVertexList method), 42
clearVertex() (apgl.graph.VertexList.VertexList method), 64
clusteringCoefficient() (apgl.generator.ErdosRenyiGenerator.ErdosRenyiGenerator method), 12
clusteringCoefficient() (apgl.generator.SmallWorldGenerator SmallWorldGenerator method), 14
clusteringCoefficient() (apgl.graph.CsArrayGraph CsArrayGraph method), 16
clusteringCoefficient() (apgl.graph.DenseGraph.DenseGraph method), 25
clusteringCoefficient() (apgl.graph.PySparseGraph.PySparseGraph method), 45
clusteringCoefficient() (apgl.graph.SparseGraph.SparseGraph method), 55
complement() (apgl.graph.CsArrayGraph.CsArrayGraph method), 16
complement() (apgl.graph.DenseGraph.DenseGraph method), 25
complement() (apgl.graph.DictGraph.DictGraph method), 34
complement() (apgl.graph.DictTree.DictTree method), 38
complement() (apgl.graph.PySparseGraph.PySparseGraph method), 45
complement() (apgl.graph.SparseGraph.SparseGraph method), 55
complement() (apgl.graph.DictGraph.DictGraph method), 34
copy() (apgl.graph.CsArrayGraph.CsArrayGraph method), 16
copy() (apgl.graph.DenseGraph.DenseGraph method), 24
copy() (apgl.graph.DictTree.DictTree method), 38
copy() (apgl.graph.GeneralVertexList GeneralVertexList method), 42
copy() (apgl.graph.DictGraph.DictGraph method), 34
copy() (apgl.graph.PySparseGraph.PySparseGraph method), 45
copy() (apgl.graph.SparseGraph.SparseGraph method), 55
copy() (apgl.graph.DictTree.DictTree method), 38

duplicate() (apgl.graph.DictTree.DictTree method), 38
degreeDistribution() (apgl.graph.CsArrayGraph.CsArrayGraph method), 16
degreeDistribution() (apgl.graph.DenseGraph.DenseGraph method), 26
degreeDistribution() (apgl.graph.PySparseGraph.PySparseGraph method), 45
degreeDistribution() (apgl.graph.SparseGraph.SparseGraph method), 55
degreeSequence() (apgl.graph.CsArrayGraph.CsArrayGraph method), 17
degreeSequence() (apgl.graph.DenseGraph.DenseGraph method), 26
degreeSequence() (apgl.graph.DictGraph.DictGraph method), 34
degreeSequence() (apgl.graph.PySparseGraph.PySparseGraph method), 45
degreeSequence() (apgl.graph.SparseGraph.SparseGraph method), 55
density() (apgl.graph.CsArrayGraph.CsArrayGraph method), 17
density() (apgl.graph.DenseGraph.DenseGraph method), 26
density() (apgl.graph.DictGraph.DictGraph method), 34
density() (apgl.graph.DictTree.DictTree method), 38
density() (apgl.graph.PySparseGraph.PySparseGraph method), 45
density() (apgl.graph.SparseGraph.SparseGraph method), 55
depth() (apgl.graph.DictTree.DictTree method), 38
depthFirstSearch() (apgl.graph.CsArrayGraph.CsArrayGraph method), 17
depthFirstSearch() (apgl.graph.DenseGraph.DenseGraph method), 26
depthFirstSearch() (apgl.graph.DictGraph.DictGraph method), 34
depthFirstSearch() (apgl.graph.DictTree.DictTree method), 38
depthFirstSearch() (apgl.graph.PySparseGraph.PySparseGraph method), 45
depthFirstSearch() (apgl.graph.SparseGraph.SparseGraph method), 55
diameter() (apgl.graph.CsArrayGraph.CsArrayGraph method), 17
diameter() (apgl.graph.DenseGraph.DenseGraph method), 26

74 Index
Another Python Graph Library Documentation, Release 0.8.1

floydWarshall() (apgl.graph.SparseGraph.SparseGraph
method), 57
fromNetworkXGraph() (apgl.graph.CsArrayGraph.CsArray
graph class method), 18
fromNetworkXGraph() (apgl.graph.DenseGraph.DenseGraph
getAllVertexIds() (apgl.graph.CsArrayGraph.CsArray
graph method), 28
fromNetworkXGraph() (apgl.graph.PySparseGraph.PySparse
graph class method), 47
fromNetworkXGraph() (apgl.graph.SparseGraph.SparseGraph
graph method), 57
getAllVertexIds() (apgl.graph.DictGraph.DictGraph
method), 35
getEdge() (apgl.graph.CsArrayGraph.CsArray
method), 19
getEdge() (apgl.graph.DenseGraph.DenseGraph
method), 28
getEdge() (apgl.graph.PySparseGraph.PySparse
graph method), 48
getEdge() (apgl.graph.SparseGraph.SparseGraph
method), 58
generate() (apgl.generator.BarabasiAlbertGenerator.BarabasiAlbert
method), 10
generate() (apgl.generator.ConfigModelGenerator.ConfigModelGenerator
method), 11
generate() (apgl.generator.ErdosRenyiGenerator.ErdosRenyi
method), 12
generate() (apgl.generator.KroneckerGenerator.Kronecker
method), 13
generate() (apgl.generator.SmallWorldGenerator.SmallWorld
method), 14
generate2() (apgl.generator.ErdosRenyiGenerator.ErdosRenyi
method), 12
geodesicDistance() (apgl.graph.CsArrayGraph.CsArray
method), 18
geodesicDistance() (apgl.graph.DenseGraph.DenseGraph
method), 28
geodesicDistance() (apgl.graph.PySparseGraph.PySparse
graph method), 47
geodesicDistance() (apgl.graph.SparseGraph.SparseGraph
method), 57
getAllDirEdges() (apgl.graph.CsArrayGraph.CsArray
graph method), 19
getAllDirEdges() (apgl.graph.DenseGraph.DenseGraph
method), 28
getAllDirEdges() (apgl.graph.DictTree.DictTree
method), 39
getAllDirEdges() (apgl.graph.DictGraph.DictGraph
method), 35
getAllDirEdges() (apgl.graph.DictTree.DictTree
method), 39
getAllDirEdges() (apgl.graph.PySparseGraph.PySparse
graph method), 48
getAllDirEdges() (apgl.graph.SparseGraph.SparseGraph
method), 58
getAllEdges() (apgl.graph.CsArrayGraph.CsArray
method), 19
getAllEdges() (apgl.graph.DictGraph.DictGraph
method), 35
getAllEdges() (apgl.graph.DictTree.DictTree
method), 39
getAllEdges() (apgl.graph.DenseGraph.DenseGraph
method), 28
getAllEdges() (apgl.graph.PySparseGraph.PySparse
graph method), 47
getAllEdges() (apgl.graph.SparseGraph.SparseGraph
method), 57
getAllEdgeIndices() (apgl.graph.DictGraph.DictGraph
method), 35
getAllEdgeIndices() (apgl.graph.DictTree.DictTree
method), 39
getAllEdgeIndices() (apgl.graph.PySparseGraph.PySparse
graph method), 48
getInDegSequence() (apgl.generator.ConfigModelGenerator.ConfigModelGenerator
method), 11
getNumDirEdges() (apgl.graph.CsArrayGraph.CsArray
graph method), 19
getNumDirEdges() (apgl.graph.DenseGraph.DenseGraph
method), 28
getNumDirEdges() (apgl.graph.DictGraph.DictGraph
method), 35
getNumDirEdges() (apgl.graph.DictTree.DictTree
method), 39
getNumDirEdges() (apgl.graph.PySparseGraph.PySparse
graph method), 48
getNumDirEdges() (apgl.graph.SparseGraph.SparseGraph
method), 58
getNumEdges() (apgl.graph.CsArrayGraph.CsArray
graph method), 19
getNumEdges() (apgl.graph.DenseGraph.DenseGraph
method), 28
getNumEdges() (apgl.graph.DictGraph.DictGraph
method), 35
getNumEdges() (apgl.graph.DictTree.DictTree
method), 39
getNumEdges() (apgl.graph.PySparseGraph.PySparse
graph method), 48
getNumEdges() (apgl.graph.SparseGraph.SparseGraph
method), 58
G
GeneralVertexList (class in
apgl.graph.GeneralVertexList), 42
generate() (apgl.generator.BarabasiAlbertGenerator.BarabasiAlbert
method), 10
generate() (apgl.generator.ConfigModelGenerator.ConfigModelGenerator
method), 11
generate() (apgl.generator.ErdosRenyiGenerator.ErdosRenyi
method), 12
generate() (apgl.generator.KroneckerGenerator.Kronecker
method), 13
generate() (apgl.generator.SmallWorldGenerator.SmallWorld
method), 14
generate2() (apgl.generator.ErdosRenyiGenerator.ErdosRenyi
method), 12
geodesicDistance() (apgl.graph.CsArrayGraph.CsArray
method), 18
geodesicDistance() (apgl.graph.DenseGraph.DenseGraph
method), 28
geodesicDistance() (apgl.graph.PySparseGraph.PySparse
graph method), 47
geodesicDistance() (apgl.graph.SparseGraph.SparseGraph
method), 57
getEdgeValues() (apgl.graph.CsArrayGraph.CsArray
method), 19
getEdgeValues() (apgl.graph.DictGraph.DictGraph
method), 35
getEdgeValues() (apgl.graph.DictTree.DictTree
method), 39
getEdgeValues() (apgl.graph.DenseGraph.DenseGraph
method), 28
getEdgeValues() (apgl.graph.PySparseGraph.PySparse
graph method), 48
getEdgeValues() (apgl.graph.SparseGraph.SparseGraph
method), 58
getFeatureDistribution() (apgl.graph.VertexList.VertexList
method), 64
getInDegSequence() (apgl.generator.ConfigModelGenerator.ConfigModelGenerator
method), 11
getNumDirEdges() (apgl.graph.CsArrayGraph.CsArray
graph method), 19
getNumDirEdges() (apgl.graph.DenseGraph.DenseGraph
method), 28
getNumDirEdges() (apgl.graph.DictGraph.DictGraph
method), 35
getNumDirEdges() (apgl.graph.DictTree.DictTree
method), 39
getNumDirEdges() (apgl.graph.PySparseGraph.PySparse
graph method), 48
getNumDirEdges() (apgl.graph.SparseGraph.SparseGraph
method), 58
getNumEdges() (apgl.graph.CsArrayGraph.CsArray
graph method), 19
getNumEdges() (apgl.graph.DenseGraph.DenseGraph
method), 28
getNumEdges() (apgl.graph.DictGraph.DictGraph
method), 35
getNumEdges() (apgl.graph.DictTree.DictTree
method), 39
getNumEdges() (apgl.graph.PySparseGraph.PySparse
graph method), 48
getNumEdges() (apgl.graph.SparseGraph.SparseGraph
method), 58
Index
getNumEdges() (apgl.graph.DictGraph.DictGraph method), 35
getNumEdges() (apgl.graph.DictTree.DictTree method), 39
getNumEdges() (apgl.graph.PySparseGraph.PySparseGraph method), 48
getNumEdges() (apgl.graph.SparseGraph.SparseGraph method), 58
getNumFeatures() (apgl.graph.VertexList.VertexList method), 64
getNumVertices() (apgl.graph.CsArrayGraph.CsArrayGraph method), 19
getNumVertices() (apgl.graph.DenseGraph.DenseGraph method), 28
getNumVertices() (apgl.graph.DictGraph.DictGraph method), 35
getNumVertices() (apgl.graph.DictTree.DictTree method), 39
getNumVertices() (apgl.graph.GeneralVertexList.GeneralVertexList method), 42
getNumVertices() (apgl.graph.PySparseGraph.PySparseGraph method), 48
getNumVertices() (apgl.graph.SparseGraph.SparseGraph method), 58
getNumVertices() (apgl.graph.VertexList.VertexList method), 64

getVertexList() (apgl.graph.CsArrayGraph.CsArrayGraph method), 19
getVertexList() (apgl.graph.DenseGraph.DenseGraph method), 29
getVertexList() (apgl.graph.PySparseGraph.PySparseGraph method), 48
getVertexList() (apgl.graph.SparseGraph.SparseGraph method), 58

getSparseWeightMatrix() (apgl.graph.DictGraph.DictGraph method), 35
getSparseWeightMatrix() (apgl.graph.DictTree.DictTree method), 39
getSparseWeightMatrix() (apgl.graph.PySparseGraph.PySparseGraph method), 48
getSparseWeightMatrix() (apgl.graph.SparseGraph.SparseGraph method), 58

getWeights() (apgl.graph.CsArrayGraph.CsArrayGraph method), 20
getWeights() (apgl.graph.DenseGraph.DenseGraph method), 29
getWeights() (apgl.graph.DictGraph.DictGraph method), 35
getWeights() (apgl.graph.DictTree.DictTree method), 40
getWeights() (apgl.graph.GeneralVertexList.GeneralVertexList method), 42
getWeights() (apgl.graph.PySparseGraph.PySparseGraph method), 48
getWeights() (apgl.graph.SparseGraph.SparseGraph method), 58
getWeights() (apgl.graph.VertexList.VertexList method), 64

harmonicGeodesicDistance() (apgl.graph.CsArrayGraph.CsArrayGraph method), 20
harmonicGeodesicDistance() (apgl.graph.DenseGraph.DenseGraph method), 29
harmonicGeodesicDistance() (apgl.graph.DictGraph.DictGraph method), 35
harmonicGeodesicDistance() (apgl.graph.DictTree.DictTree method), 40
harmonicGeodesicDistance() (apgl.graph.GeneralVertexList.GeneralVertexList method), 42
harmonicGeodesicDistance() (apgl.graph.PySparseGraph.PySparseGraph method), 48
harmonicGeodesicDistance() (apgl.graph.SparseGraph.SparseGraph method), 58
hopCount() (apgl.graph.CsArrayGraph.CsArrayGraph method), 20
hopCount() (apgl.graph.DenseGraph.DenseGraph method), 29
hopCount() (apgl.graph.PySparseGraph.PySparseGraph method), 49
hopCount() (apgl.graph.SparseGraph.SparseGraph method), 59

incidenceMatrix() (apgl.graph.CsArrayGraph.CsArrayGraph method), 20
incidenceMatrix() (apgl.graph.DenseGraph.DenseGraph method), 29
incidenceMatrix() (apgl.graph.PySparseGraph.PySparseGraph method), 49
incidenceMatrix() (apgl.graph.SparseGraph.SparseGraph method), 59

inDegreeDistribution() (apgl.graph.CsArrayGraph.CsArrayGraph method), 20
inDegreeDistribution() (apgl.graph.DenseGraph.DenseGraph method), 29
inDegreeDistribution() (apgl.graph.DictGraph.DictGraph method), 36
inDegreeDistribution() (apgl.graph.DictTree.DictTree method), 40
inDegreeDistribution() (apgl.graph.PySparseGraph.PySparseGraph method), 49
inDegreeDistribution() (apgl.graph.SparseGraph.SparseGraph method), 59

inDegreeSequence() (apgl.graph.CsArrayGraph.CsArrayGraph method), 20
inDegreeSequence() (apgl.graph.DenseGraph.DenseGraph method), 29
inDegreeSequence() (apgl.graph.DictGraph.DictGraph method), 36
inDegreeSequence() (apgl.graph.DictTree.DictTree method), 40
inDegreeSequence() (apgl.graph.PySparseGraph.PySparseGraph method), 49
inDegreeSequence() (apgl.graph.SparseGraph.SparseGraph method), 59

intersect() (apgl.graph.CsArrayGraph.CsArrayGraph method), 20
intersect() (apgl.graph.DenseGraph.DenseGraph method), 29
intersect() (apgl.graph.DictTree.DictTree method), 40
intersect() (apgl.graph.PySparseGraph.PySparseGraph method), 49
intersect() (apgl.graph.SparseGraph.SparseGraph method), 59

intTypes (apgl.util.Parameter attribute), 67
isLeaf() (apgl.graph.DictTree.DictTree method), 40
isNonLeaf() (apgl.graph.DictTree.DictTree method), 40
isSubtree() (apgl.graph.DictTree.DictTree method), 40
isTree() (apgl.graph.CsArrayGraph.CsArrayGraph method), 20
isTree() (apgl.graph.DenseGraph.DenseGraph method), 30
isTree() (apgl.graph.PySparseGraph.PySparseGraph method), 49
isTree() (apgl.graph.SparseGraph.SparseGraph method), 59

isUndirected() (apgl.graph.CsArrayGraph.CsArrayGraph method), 20
isUndirected() (apgl.graph.DenseGraph.DenseGraph method), 30
isUndirected() (apgl.graph.DictGraph.DictGraph method), 36
isUndirected() (apgl.graph.DictTree.DictTree method), 40
isUndirected() (apgl.graph.PySparseGraph.PySparseGraph method), 49
isUndirected() (apgl.graph.SparseGraph.SparseGraph method), 59

KroneckerGenerator (class in apgl.generator.KroneckerGenerator), 13

laplacianMatrix() (apgl.graph.CsArrayGraph.CsArrayGraph method), 20
laplacianMatrix() (apgl.graph.DenseGraph.DenseGraph method), 30
laplacianMatrix() (apgl.graph.PySparseGraph.PySparseGraph method), 49
laplacianMatrix() (apgl.graph.SparseGraph.SparseGraph method), 59

laplacianWeightMatrix() (apgl.graph.CsArrayGraph.CsArrayGraph method), 21
laplacianWeightMatrix() (apgl.graph.DenseGraph.DenseGraph method), 30
laplacianWeightMatrix() (apgl.graph.PySparseGraph.PySparseGraph method), 49
laplacianWeightMatrix() (apgl.graph.SparseGraph.SparseGraph method), 59

leaves() (apgl.graph.DictTree.DictTree method), 40
load() (apgl.graph.CsArrayGraph.CsArrayGraph method), 21
load() (apgl.graph.DenseGraph.DenseGraph method), 30
load() (apgl.graph.GeneralVertexList.GeneralVertexList static method), 43
load() (apgl.graph.PySparseGraph.PySparseGraph method), 50
load() (apgl.graph.SparseGraph.SparseGraph method), 60

loadMatrix() (apgl.graph.CsArrayGraph.CsArrayGraph static method), 21
loadMatrix() (apgl.graph.DenseGraph.DenseGraph static method), 30
loadMatrix() (apgl.graph.PySparseGraph.PySparseGraph static method), 49
loadMatrix() (apgl.graph.SparseGraph.SparseGraph static method), 60
<table>
<thead>
<tr>
<th>Function</th>
<th>Class</th>
<th>Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadMatrix()</td>
<td>aplg.graph.SparseGraph.SparseGraph</td>
<td>static method</td>
<td>60</td>
</tr>
<tr>
<td>maxEigenvector()</td>
<td>aplg.graph.CsArrayGraph.CsArrayGraph</td>
<td>method</td>
<td>21</td>
</tr>
<tr>
<td>maxEigenvector()</td>
<td>aplg.graph.DenseGraph.DenseGraph</td>
<td>method</td>
<td>30</td>
</tr>
<tr>
<td>maxEigenvector()</td>
<td>aplg.graph.PySparseGraph.PySparseGraph</td>
<td>method</td>
<td>50</td>
</tr>
<tr>
<td>maxEigenvector()</td>
<td>aplg.graph.SparseGraph.SparseGraph</td>
<td>method</td>
<td>60</td>
</tr>
<tr>
<td>maxProductPaths()</td>
<td>aplg.graph.CsArrayGraph.CsArrayGraph</td>
<td>method</td>
<td>21</td>
</tr>
<tr>
<td>maxProductPaths()</td>
<td>aplg.graph.DenseGraph.DenseGraph</td>
<td>method</td>
<td>30</td>
</tr>
<tr>
<td>maxProductPaths()</td>
<td>aplg.graph.PySparseGraph.PySparseGraph</td>
<td>method</td>
<td>50</td>
</tr>
<tr>
<td>maxProductPaths()</td>
<td>aplg.graph.SparseGraph.SparseGraph</td>
<td>method</td>
<td>60</td>
</tr>
<tr>
<td>maybeIsomorphicWith()</td>
<td>aplg.graph.CsArrayGraph.CsArrayGraph</td>
<td>method</td>
<td>21</td>
</tr>
<tr>
<td>maybeIsomorphicWith()</td>
<td>aplg.graph.DenseGraph.DenseGraph</td>
<td>method</td>
<td>30</td>
</tr>
<tr>
<td>maybeIsomorphicWith()</td>
<td>aplg.graph.PySparseGraph.PySparseGraph</td>
<td>method</td>
<td>50</td>
</tr>
<tr>
<td>maybeIsomorphicWith()</td>
<td>aplg.graph.SparseGraph.SparseGraph</td>
<td>method</td>
<td>60</td>
</tr>
<tr>
<td>multiply()</td>
<td>aplg.graph.CsArrayGraph.CsArrayGraph</td>
<td>method</td>
<td>21</td>
</tr>
<tr>
<td>multiply()</td>
<td>aplg.graph.DenseGraph.DenseGraph</td>
<td>method</td>
<td>30</td>
</tr>
<tr>
<td>multiply()</td>
<td>aplg.graph.PySparseGraph.PySparseGraph</td>
<td>method</td>
<td>50</td>
</tr>
<tr>
<td>multiply()</td>
<td>aplg.graph.SparseGraph.SparseGraph</td>
<td>method</td>
<td>60</td>
</tr>
<tr>
<td>nativeAdjacencyMatrix()</td>
<td>aplg.graph.PySparseGraph.PySparseGraph</td>
<td>method</td>
<td>50</td>
</tr>
<tr>
<td>nativeAdjacencyMatrix()</td>
<td>aplg.graph.SparseGraph.SparseGraph</td>
<td>method</td>
<td>60</td>
</tr>
<tr>
<td>neighbourOf()</td>
<td>aplg.graph.CsArrayGraph.CsArrayGraph</td>
<td>method</td>
<td>21</td>
</tr>
<tr>
<td>neighbourOf()</td>
<td>aplg.graph.DenseGraph.DenseGraph</td>
<td>method</td>
<td>31</td>
</tr>
<tr>
<td>neighbourOf()</td>
<td>aplg.graph.DictGraph.DictGraph</td>
<td>method</td>
<td>36</td>
</tr>
<tr>
<td>neighbourOf()</td>
<td>aplg.graph.DictTree.DictTree</td>
<td>method</td>
<td>40</td>
</tr>
<tr>
<td>neighbourOf()</td>
<td>aplg.graph.PySparseGraph.PySparseGraph</td>
<td>method</td>
<td>50</td>
</tr>
<tr>
<td>neighbourOf()</td>
<td>aplg.graph.SparseGraph.SparseGraph</td>
<td>method</td>
<td>60</td>
</tr>
<tr>
<td>neighbour()</td>
<td>aplg.graph.CsArrayGraph.CsArrayGraph</td>
<td>method</td>
<td>21</td>
</tr>
<tr>
<td>neighbour()</td>
<td>aplg.graph.DenseGraph.DenseGraph</td>
<td>method</td>
<td>31</td>
</tr>
<tr>
<td>neighbour()</td>
<td>aplg.graph.DictGraph.DictGraph</td>
<td>method</td>
<td>36</td>
</tr>
<tr>
<td>neighbour()</td>
<td>aplg.graph.DictTree.DictTree</td>
<td>method</td>
<td>40</td>
</tr>
<tr>
<td>neighbour()</td>
<td>aplg.graph.PySparseGraph.PySparseGraph</td>
<td>method</td>
<td>50</td>
</tr>
<tr>
<td>neighbour()</td>
<td>aplg.graph.SparseGraph.SparseGraph</td>
<td>method</td>
<td>60</td>
</tr>
</tbody>
</table>

**Index 79**
Another Python Graph Library Documentation, Release 0.8.1

P
Parameter (class in apgl.util), 66
pruneVertex() (apgl.graph.DictTree.DictTree method), 40
PySparseGraph (class in apgl.graph.PySparseGraph), 44

R
readFromFile() (apgl.io.SimpleGraphReader method), 65
removeAllEdges() (apgl.graph.CsArrayGraph.CsArrayGraph method), 22
removeAllEdges() (apgl.graph.DenseGraph.DenseGraph method), 31
removeAllEdges() (apgl.graph.PySparseGraph.PySparseGraph method), 51
removeAllEdges() (apgl.graph.SparseGraph.SparseGraph method), 61
removeEdge() (apgl.graph.CsArrayGraph.CsArrayGraph method), 22
removeEdge() (apgl.graph.DenseGraph.DenseGraph method), 31
removeEdge() (apgl.graph.DictGraph.DictGraph method), 36
removeEdge() (apgl.graph.DictTree.DictTree method), 40
removeEdge() (apgl.graph.PySparseGraph.PySparseGraph method), 51
removeEdge() (apgl.graph.SparseGraph.SparseGraph method), 61
removeEdges() (apgl.graph.DictTree.DictTree method), 41
removeVertex() (apgl.graph.DictGraph.DictGraph method), 36
removeVertex() (apgl.graph.DictTree.DictTree method), 41
replaceVertices() (apgl.graph.VertexList.VertexList method), 64

S
save() (apgl.graph.CsArrayGraph.CsArrayGraph method), 22
save() (apgl.graph.DenseGraph.DenseGraph method), 31
save() (apgl.graph.GeneralVertexList.GeneralVertexList method), 43
save() (apgl.graph.PySparseGraph.PySparseGraph method), 51
save() (apgl.graph.SparseGraph.SparseGraph method), 61
save() (apgl.graph.VertexList.VertexList method), 64
saveMatrix() (apgl.graph.CsArrayGraph.CsArrayGraph method), 22
saveMatrix() (apgl.graph.DenseGraph.DenseGraph method), 31
saveMatrix() (apgl.graph.PySparseGraph.PySparseGraph method), 51
saveMatrix() (apgl.graph.SparseGraph.SparseGraph method), 61
setDiff() (apgl.graph.CsArrayGraph.CsArrayGraph method), 22
setDiff() (apgl.graph.DenseGraph.DenseGraph method), 31
setDiff() (apgl.graph.PySparseGraph.PySparseGraph method), 51
setDiff() (apgl.graph.SparseGraph.SparseGraph method), 61
setEll() (apgl.generator.BarabasiAlbertGenerator.BarabasiAlbertGenerator method), 10
setInDegSequence() (apgl.generator.ConfigModelGenerator.ConfigModelGenerator method), 11
setK() (apgl.generator.KroneckerGenerator.KroneckerGenerator method), 13
setK() (apgl.generator.SmallWorldGenerator.SmallWorldGenerator method), 14
setM() (apgl.generator.BarabasiAlbertGenerator.BarabasiAlbertGenerator method), 10
setOutDegSequence() (apgl.generator.ConfigModelGenerator.ConfigModelGenerator method), 11
setP() (apgl.generator.ErdosRenyiGenerator.ErdosRenyiGenerator method), 12
setP() (apgl.generator.SmallWorldGenerator.SmallWorldGenerator method), 14
setSelfEdges() (apgl.generator.ErdosRenyiGenerator.ErdosRenyiGenerator method), 12
setVertex() (apgl.graph.CsArrayGraph.CsArrayGraph method), 22
setVertex() (apgl.graph.DenseGraph.DenseGraph method), 31
setVertex() (apgl.graph.DictGraph.DictGraph method), 36
setVertex() (apgl.graph.DictTree.DictTree method), 41
setVertex() (apgl.graph.GeneralVertexList.GeneralVertexList method), 43
setVertex() (apgl.graph.PySparseGraph.PySparseGraph method), 51
setVertex() (apgl.graph.SparseGraph.SparseGraph method), 61
setVertex() (apgl.graph.VertexList.VertexList method), 64
setVertexList() (apgl.graph.CsArrayGraph.CsArrayGraph method), 22
setVertexList() (apgl.graph.DenseGraph.DenseGraph method), 32
setVertexList() (apgl.graph.PySparseGraph.PySparseGraph method), 51
setVertexList() (apgl.graph.SparseGraph.SparseGraph method), 61
setVertices() (apgl.graph.CsArrayGraph.CsArrayGraph method), 22
setVertices() (apgl.graph.DenseGraph.DenseGraph method), 32
setVertices() (apgl.graph.PySparseGraph.PySparseGraph method), 51
setVertices() (apgl.graph.SparseGraph.SparseGraph method), 61
U
undirected (apgl.graph.CsArrayGraph.CsArrayGraph attribute), 23
undirected (apgl.graph.DenseGraph.DenseGraph attribute), 32
undirected (apgl.graph.DictGraph.DictGraph attribute), 37
undirected (apgl.graph.DictTree.DictTree attribute), 42
undirected (apgl.graph.PySparseGraph.PySparseGraph attribute), 52
undirected (apgl.graph.SparseGraph.SparseGraph attribute), 63
union() (apgl.graph.CsArrayGraph.CsArrayGraph method), 23
union() (apgl.graph.DenseGraph.DenseGraph method), 32
union() (apgl.graph.PySparseGraph.PySparseGraph method), 52
union() (apgl.graph.SparseGraph.SparseGraph method), 63

V
V (apgl.graph.GeneralVertexList.GeneralVertexList attribute), 42
V (apgl.graph.VertexList.VertexList attribute), 63
vertexExists() (apgl.graph.DictGraph.DictGraph method), 37
vertexExists() (apgl.graph.DictTree.DictTree method), 42
VertexList (class in apgl.graph.VertexList), 63
vertices (apgl.graph.DictGraph.DictGraph attribute), 37
vertices (apgl.graph.DictTree.DictTree attribute), 42
vList (apgl.graph.CsArrayGraph.CsArrayGraph attribute), 24
vlist (apgl.graph.CsArrayGraph.CsArrayGraph attribute), 24
vList (apgl.graph.DenseGraph.DenseGraph attribute), 33
vlist (apgl.graph.DenseGraph.DenseGraph attribute), 33
vList (apgl.graph.PySparseGraph.PySparseGraph attribute), 52
vlist (apgl.graph.PySparseGraph.PySparseGraph attribute), 52
vList (apgl.graph.SparseGraph.SparseGraph attribute), 63
vlist (apgl.graph.SparseGraph.SparseGraph attribute), 63

W
W (apgl.graph.CsArrayGraph.CsArrayGraph attribute), 15
W (apgl.graph.DenseGraph.DenseGraph attribute), 24
W (apgl.graph.SparseGraph.SparseGraph attribute), 54
weightMatrixDType() (apgl.graph.CsArrayGraph.CsArrayGraph method), 24
weightMatrixDType() (apgl.graph.DenseGraph.DenseGraph method), 33
weightMatrixDType() (apgl.graph.PySparseGraph.PySparseGraph method), 53
weightMatrixDType() (apgl.graph.SparseGraph.SparseGraph method), 63
whatToDo() (apgl.util.Parameter static method), 67
writeToFile() (apgl.io.SimpleGraphWriter method), 65