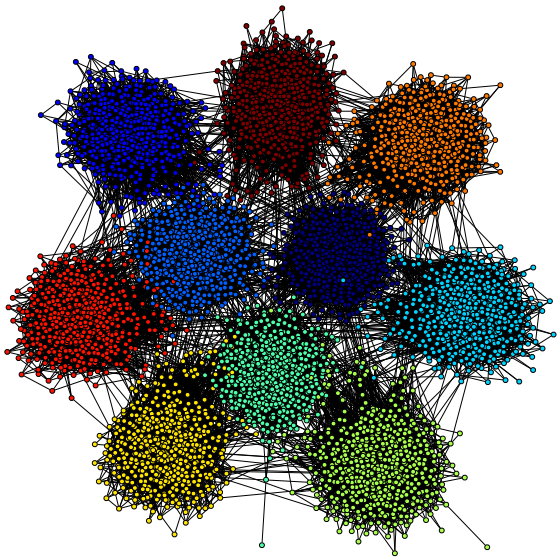


graph-tool documentation

Release 2.2.24

Tiago de Paula Peixoto

May 06, 2013

CONTENTS

1 Quick start using *graph-tool* **3**
- 1.1 Creating and manipulating graphs . . . 3
  - 1.1.1 Iterating over vertices and edges . . . 6
- 1.2 Property maps . . . 6
  - 1.2.1 Internal property maps . . . 7
- 1.3 Graph I/O . . . 8
- 1.4 An Example: Building a Price Network . . . 8
- 1.5 Graph filtering . . . 11
  - 1.5.1 Graph views . . . 17

2 Cookbook **21**
- 2.1 Animations with graph-tool . . . 21
  - 2.1.1 SIRS epidemics . . . 21
  - 2.1.2 Dynamic layout . . . 24

3 Module documentation **27**
- 3.1 graph_tool - efficient graph analysis and manipulation . . . 27
  - 3.1.1 Summary . . . 27
  - 3.1.2 How to use the documentation . . . 27
  - 3.1.3 Contents . . . 28
- 3.2 Available subpackages . . . 39
  - 3.2.1 `graph_tool.centrality` - Centrality measures . . . 39
  - 3.2.2 `graph_tool.clustering` - Clustering coefficients . . . 60
  - 3.2.3 `graph_tool.collection` - Dataset collection . . . 65
  - 3.2.4 `graph_tool.community` - Community structure . . . 68
  - 3.2.5 `graph_tool.correlations` - Correlations . . . 93
  - 3.2.6 `graph_tool.draw` - Graph drawing and layout . . . 102
  - 3.2.7 `graph_tool.flow` - Maximum flow algorithms . . . 126
  - 3.2.8 `graph_tool.generation` - Random graph generation . . . 136
  - 3.2.9 `graph_tool.run_action` - Inline C++ code embedding . . . 163
  - 3.2.10 `graph_tool.search` - Search algorithms . . . 166
  - 3.2.11 `graph_tool.spectral` - Spectral properties . . . 186
  - 3.2.12 `graph_tool.stats` - Miscellaneous statistics . . . 190
  - 3.2.13 `graph_tool.topology` - Assessing graph topology . . . 194
  - 3.2.14 `graph_tool.util` - Graph utilities . . . 222

4 Indexes and tables **225**

Bibliography **227**

Python Module Index **235**

Python Module Index **237**

Contents:

CHAPTER

ONE

QUICK START USING *GRAPH-TOOL*

The graph_tool (page 27) module provides a Graph (page 28) class and several algorithms that operate on it. The internals of this class, and of most algorithms, are written in C++ for performance, using the Boost Graph Library.

The module must be of course imported before it can be used. The package is subdivided into several sub-modules. To import everything from all of them, one can do:

```
>>> from graph_tool.all import *
```

In the following, it will always be assumed that the previous line was run.

1.1 Creating and manipulating graphs

An empty graph can be created by instantiating a Graph (page 28) class:

```
>>> g = Graph()
```

By default, newly created graphs are always directed. To construct undirected graphs, one must pass a value to the directed parameter:

```
>>> ug = Graph(directed=False)
```

A graph can always be switched *on-the-fly* from directed to undirected (and vice versa), with the set_directed() (page 29) method. The "directedness" of the graph can be queried with the is_directed() (page 30) method,

```
>>> ug = Graph()
>>> ug.set_directed(False)
>>> assert(ug.is_directed() == False)
```

A graph can also be created by providing another graph, in which case the entire graph (and its internal property maps, see *Property maps* (page 6)) is copied:

```
>>> g1 = Graph()
>>> # ... construct g1 ...
>>> g2 = Graph(g1)                  # g1 and g2 are copies
```

Above, g2 is a "deep" copy of g1, i.e. any modification of g2 will not affect g1.

Once a graph is created, it can be populated with vertices and edges. A vertex can be added with the add_vertex() (page 29) method, which returns an instance of a Vertex (page 33) class, also called a *vertex descriptor*. For instance, the following code creates two vertices, and returns vertex descriptors stored in the variables v1 and v2.

```
>>> v1 = g.add_vertex()
>>> v2 = g.add_vertex()
```

Edges can be added in an analogous manner, by calling the `add_edge()` (page 29) method, which returns an edge descriptor (an instance of the `Edge` (page 34) class):

```
>>> e = g.add_edge(v1, v2)
```

The above code creates a directed edge from `v1` to `v2`. We can visualize the graph we created so far with the `graph_draw()` (page 111) function.

```
>>> graph_draw(g, vertex_text=g.vertex_index, vertex_font_size=18,
...            output_size=(200, 200), output="two-nodes.pdf")
<...>
```

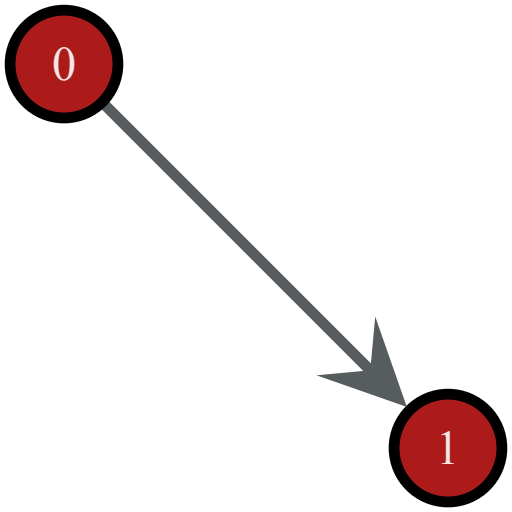


Figure 1.1: A simple directed graph with two vertices and one edge, created by the commands above.

With vertex and edge descriptors, one can examine and manipulate the graph in an arbitrary manner. For instance, in order to obtain the out-degree of a vertex, we can simply call the `out_degree()` (page 34) method:

```
>>> print(v1.out_degree())
1
```

Analogously, we could have used the `in_degree()` (page 34) method to query the in-degree.

Note: For undirected graphs, the "out-degree" is synonym for degree, and in this case the in-degree of a vertex is always zero.

Edge descriptors have two useful methods, `source()` (page 35) and `target()` (page 35), which return the source and target vertex of an edge, respectively.

```
>>> print(e.source(), e.target())
0 1
```

The `add_vertex()` (page 29) method also accepts an optional parameter which specifies the number of vertices to create. If this value is greater than 1, it returns an iterator on the added vertex descriptors:

```
>>> vlist = g.add_vertex(10)
>>> print(len(list(vlist)))
10
```

Edges and vertices can also be removed at any time with the `remove_vertex()` (page 29) and `remove_edge()` (page 29) methods,

```
>>> g.remove_edge(e)                                  # e no longer exists
>>> g.remove_vertex(v2)                  # the second vertex is also gone
```

Note: Removing a vertex is an $O(N)$ operation. The vertices are internally stored in a STL vector, so removing an element somewhere in the middle of the list requires the shifting of the rest of the list. Thus, fast $O(1)$ removals are only possible if one can guarantee that only vertices in the end of the list are removed (the ones last added to the graph).

Removing an edge is an $O(k_s + k_t)$ operation, where k_s is the out-degree of the source vertex, and k_t is the in-degree of the target vertex.

Each vertex in a graph has an unique index, which is numbered from 0 to N-1, where N is the number of vertices. This index can be obtained by using the `vertex_index` (page 30) attribute of the graph (which is a *property map*, see *Property maps* (page 6)), or by converting the vertex descriptor to an `int`.

```
>>> v = g.add_vertex()
>>> print(g.vertex_index[v])
11
>>> print(int(v))
11
```

Note: Removing a vertex will cause the index of any vertex with a larger index to be changed, so that the indexes *always* conform to the $[0, N-1]$ range. However, property map values (see *Property maps* (page 6)) are unaffected.

Since vertices are uniquely identifiable by their indexes, there is no need to keep the vertex descriptor lying around to access them at a later point. If we know its index, one can obtain the descriptor of a vertex with a given index using the `vertex()` (page 28) method,

```
>>> v = g.vertex(8)
```

which takes an index, and returns a vertex descriptor. Edges cannot be directly obtained by its index, but if the source and target vertices of a given edge is known, it can be obtained with the `edge()` (page 28) method

```
>>> g.add_edge(g.vertex(2), g.vertex(3))
<...>
>>> e = g.edge(2, 3)
```

Another way to obtain edge or vertex descriptors is to *iterate* through them, as described in section *Iterating over vertices and edges* (page 6). This is in fact the most useful way of obtaining vertex and edge descriptors.

Like vertices, edges also have unique indexes, which are given by the `edge_index` (page 30) property:

```
>>> e = g.add_edge(g.vertex(0), g.vertex(1))
>>> print(g.edge_index[e])
1
```

Differently from vertices, edge indexes do not necessarily conform to any specific range. If no edges are ever removed, the indexes will be in the range $[0, E-1]$, where E is the number of edges, and edges added earlier have lower indexes. However if an edge is removed, its index will be "vacant", and the remaining indexes will be left unmodified, and thus will not lie in the range $[0, E-1]$. If a new edge is added, it will reuse old indexes, in an increasing order.

1.1.1 Iterating over vertices and edges

Algorithms must often iterate through vertices, edges, out-edges of a vertex, etc. The `Graph` (page 28) and `Vertex` (page 33) classes provide different types of iterators for doing so. The iterators always point to edge or vertex descriptors.

Iterating over all vertices or edges

In order to iterate through all the vertices or edges of a graph, the `vertices()` (page 28) and `edges()` (page 28) methods should be used:

```
for v in g.vertices():
    print(v)
for e in e.edges():
    print(e)
```

The code above will print the vertices and edges of the graph in the order they are found.

Iterating over the neighbourhood of a vertex

The out- and in-edges of a vertex, as well as the out- and in-neighbours can be iterated through with the `out_edges()` (page 34), `in_edges()` (page 34), `out_neighbours()` (page 34) and `in_neighbours()` (page 34) methods, respectively.

```
from itertools import izip
for v in g.vertices():
   for e in v.out_edges():
       print(e)
   for w in v.out_neighbours():
       print(w)

   # the edge and neighbours order always match
   for e,w in izip(v.out_edges(), v.out_neighbours()):
       assert(e.target() == w)
```

The code above will print the out-edges and out-neighbours of all vertices in the graph.

Note: The ordering of the vertices and edges visited by the iterators is always the same as the order in which they were added to the graph (with the exception of the iterator returned by `edges()` (page 28)). Usually, algorithms do not care about this order, but if it is ever needed, this inherent ordering can be relied upon.

Warning: You should never remove vertex or edge descriptors when iterating over them, since this invalidates the iterators. If you plan to remove vertices or edges during iteration, you must first store them somewhere (such as in a list) and remove them only after no iterator is being used. Removal during iteration will cause bad things to happen.

1.2 Property maps

Property maps are a way of associating additional information to the vertices, edges or to the graph itself. There are thus three types of property maps: vertex, edge and graph. All of them are handled by the same class, `PropertyMap` (page 35). Each created property map has an associated *value type*, which must be chosen from the predefined set:

Type name	Alias
bool	uint8_t
int16_t	short
int32_t	int
int64_t	long, long long
double	float
long double	
string	
vector<bool>	vector<uint8_t>
vector<int16_t>	vector<short>
vector<int32_t>	vector<int>
vector<int64_t>	vector<long>, vector<long long>
vector<double>	vector<float>
vector<long double>	
vector<string>	
python::object	object

New property maps can be created for a given graph by calling the new_vertex_property() (page 30), new_edge_property() (page 30), or new_graph_property() (page 30), for each map type. The values are then accessed by vertex or edge descriptors, or the graph itself, as such:

```
from itertools import izip
from numpy.random import randint

g = Graph()
g.add_vertex(100)
# insert some random links
for s,t in izip(randint(0, 100, 100), randint(0, 100, 100)):
    g.add_edge(g.vertex(s), g.vertex(t))

vprop_double = g.new_vertex_property("double")            # Double-precision floating point
vprop_double[g.vertex(10)] = 3.1416

vprop_vint = g.new_vertex_property("vector<int>")         # Vector of ints
vprop_vint[g.vertex(40)] = [1, 3, 42, 54]

eprop_dict = g.new_edge_property("object")                # Arbitrary python object. In this case
eprop_dict[g.edges().next()] = {"foo": "bar", "gnu": 42}

gprop_bool = g.new_edge_property("bool")                  # Boolean
gprop_bool[g] = True
```

Property maps with scalar value types can also be accessed as a numpy.ndarray, with the get_array() (page 36) method, or the a (page 36) attribute, i.e.,

```
from numpy.random import random

# this assigns random values to the vertex properties
vprop_double.get_array()[:] = random(g.num_vertices())

# or more conveniently (this is equivalent to the above)
vprop_double.a = random(g.num_vertices())
```

1.2.1 Internal property maps

Any created property map can be made "internal" to the corresponding graph. This means that it will be copied and saved to a file together with the graph. Properties are internalized by including them in the graph's dictionary-like attributes vertex_properties (page 31), edge_properties (page 31) or graph_properties (page 31) (or their aliases, vp (page 31),

ep (page 31) or gp (page 31), respectively). When inserted in the graph, the property maps must have an unique name (between those of the same type):

```
>>> eprop = g.new_edge_property("string")
>>> g.edge_properties["some name"] = eprop
>>> g.list_properties()
some name       (edge)     (type: string)
```

Internal graph property maps behave slightly differently. Instead of returning the property map object, the value itself is returned from the dictionaries:

```
>>> gprop = g.new_graph_property("int")
>>> g.graph_properties["foo"] = gprop    # this sets the actual property map
>>> g.graph_properties["foo"] = 42       # this sets its value
>>> print(g.graph_properties["foo"])
42
>>> del g.graph_properties["foo"]        # the property map entry is deleted from the dictionary
```

1.3 Graph I/O

Graphs can be saved and loaded in three formats: graphml, dot and gml. `Graphml` is the default and preferred format, since it is by far the most complete. The `dot` and `gml` formats are fully supported, but since they contain no precise type information, all properties are read as strings (or also as double, in the case of `gml`), and must be converted by hand. Therefore you should always use graphml, since it performs an exact bit-for-bit representation of all supported *Property maps* (page 6), except when interfacing with other software, or existing data, which uses `dot` or `gml`.

A graph can be saved or loaded to a file with the `save` (page 33) and `load` (page 33) methods, which take either a file name or a file-like object. A graph can also be loaded from disc with the `load_graph()` (page 36) function, as such:

```
g = Graph()
#  ... fill the graph ...
g.save("my_graph.xml.gz")
g2 = load_graph("my_graph.xml.gz")
# g and g2 should be copies of each other
```

Graph classes can also be pickled with the `pickle` module.

1.4 An Example: Building a Price Network

A Price network is the first known model of a "scale-free" graph, invented in 1976 by de Solla Price. It is defined dynamically, where at each time step a new vertex is added to the graph, and connected to an old vertex, with probability proportional to its in-degree. The following program implements this construction using `graph-tool`.

Note: Note that it would be much faster just to use the `price_network()` (page 162) function, which is implemented in C++, as opposed to the script below which is in pure python. The code below is merely a demonstration on how to use the library.

```
#! /usr/bin/env python

# We probably will need some things from several places
import sys, os
from pylab import *  # for plotting
from numpy.random import *  # for random sampling
```

```
seed(42)

# We need to import the graph_tool module itself
from graph_tool.all import *

# let's construct a Price network (the one that existed before Barabasi). It is
# a directed network, with preferential attachment. The algorithm below is
# very naive, and a bit slow, but quite simple.

# We start with an empty, directed graph
g = Graph()

# We want also to keep the age information for each vertex and edge. For that
# let's create some property maps
v_age = g.new_vertex_property("int")
e_age = g.new_edge_property("int")

# The final size of the network
N = 100000

# We have to start with one vertex
v = g.add_vertex()
v_age[v] = 0

# we will keep a list of the vertices. The number of times a vertex is in this
# list will give the probability of it being selected.
vlist = [v]

# let's now add the new edges and vertices
for i in range(1, N):
    # create our new vertex
    v = g.add_vertex()
    v_age[v] = i

    # we need to sample a new vertex to be the target, based on its in-degree +
    # 1. For that, we simply randomly sample it from vlist.
    i = randint(0, len(vlist))
    target = vlist[i]

    # add edge
    e = g.add_edge(v, target)
    e_age[e] = i

    # put v and target in the list
    vlist.append(target)
    vlist.append(v)

# now we have a graph!

# let's do a random walk on the graph and print the age of the vertices we find,
# just for fun.

v = g.vertex(randint(0, g.num_vertices()))
while True:
    print("vertex:", v, "in-degree:", v.in_degree(), "out-degree:",\
          v.out_degree(), "age:", v_age[v])

    if v.out_degree() == 0:
        print("Nowhere else to go... We found the main hub!")
        break

    n_list = []
    for w in v.out_neighbours():
```

```
70          n_list.append(w)
71      v = n_list[randint(0, len(n_list))]
72
73  # let's save our graph for posterity. We want to save the age properties as
74  # well... To do this, they must become "internal" properties:
75
76  g.vertex_properties["age"] = v_age
77  g.edge_properties["age"] = e_age
78
79  # now we can save it
80  g.save("price.xml.gz")
81
82
83  # Let's plot its in-degree distribution
84  in_hist = vertex_hist(g, "in")
85
86  y = in_hist[0]
87  err = sqrt(in_hist[0])
88  err[err >= y] = y[err >= y] - 1e-2
89
90  figure(figsize=(6,4))
91  errorbar(in_hist[1][:-1], in_hist[0], fmt="o", yerr=err,
92           label="in")
93  gca().set_yscale("log")
94  gca().set_xscale("log")
95  gca().set_ylim(1e-1, 1e5)
96  gca().set_xlim(0.8, 1e3)
97  subplots_adjust(left=0.2, bottom=0.2)
98  xlabel("$k_{in}$")
99  ylabel("$NP(k_{in})$")
100 tight_layout()
101 savefig("price-deg-dist.pdf")
```

The following is what should happen when the program is run.

```
vertex: 36063 in-degree: 0 out-degree: 1 age: 36063
vertex: 9075 in-degree: 4 out-degree: 1 age: 9075
vertex: 5967 in-degree: 3 out-degree: 1 age: 5967
vertex: 1113 in-degree: 7 out-degree: 1 age: 1113
vertex: 25 in-degree: 84 out-degree: 1 age: 25
vertex: 10 in-degree: 541 out-degree: 1 age: 10
vertex: 5 in-degree: 140 out-degree: 1 age: 5
vertex: 2 in-degree: 459 out-degree: 1 age: 2
vertex: 1 in-degree: 520 out-degree: 1 age: 1
vertex: 0 in-degree: 210 out-degree: 0 age: 0
Nowhere else to go... We found the main hub!
```

Below is the degree distribution, with 100000 nodes. If you want to see a broader power law, try to increase the number of vertices to something like 10^6 or 10^7.

We can draw the graph to see some other features of its topology. For that we use the `graph_draw()` (page 111) function.

```
g = load_graph("price.xml.gz")
age = g.vertex_properties["age"]

graph_draw(g, output_size=(1000, 1000), vertex_color=age,
           vertex_fill_color=age, vertex_size=1, edge_pen_width=1.2,
           output="price.png")
```

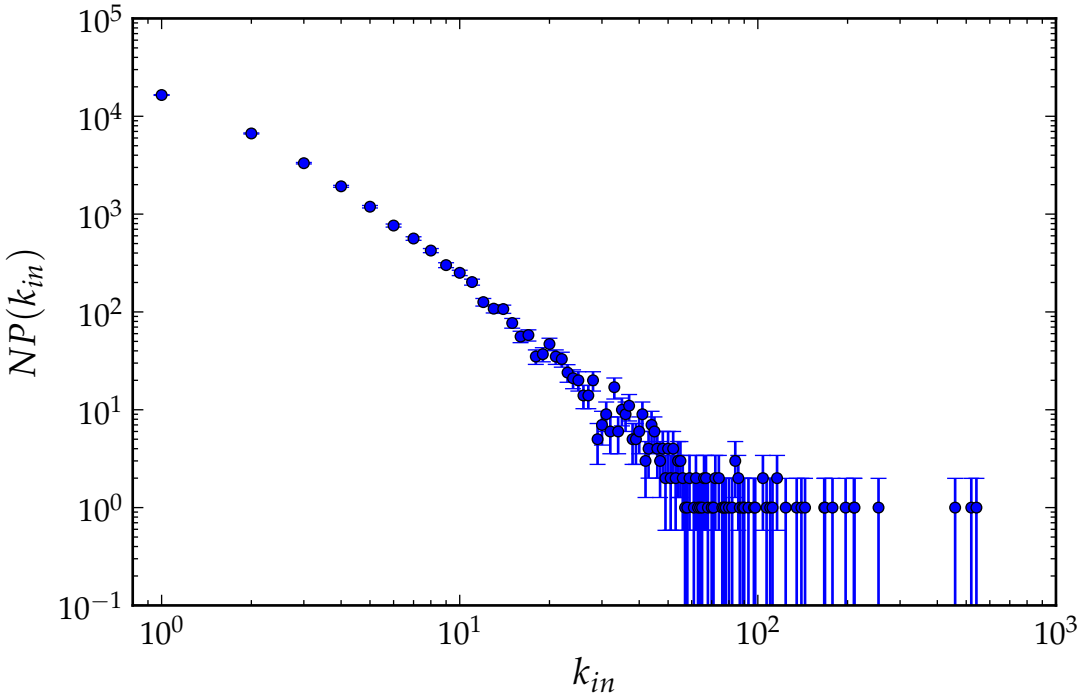


Figure 1.2: In-degree distribution of a price network with 10^5 nodes.

1.5 Graph filtering

One of the very nice features of `graph-tool` is the "on-the-fly" filtering of edges and/or vertices. Filtering means the temporary masking of vertices/edges, which are in fact not really removed, and can be easily recovered. Vertices or edges which are to be filtered should be marked with a `PropertyMap` (page 35) with value type `bool`, and then set with `set_vertex_filter()` (page 32) or `set_edge_filter()` (page 32) methods. By default, vertex or edges with value "1" are *kept* in the graphs, and those with value "0" are filtered out. This behaviour can be modified with the `inverted` parameter of the respective functions. All manipulation functions and algorithms will work as if the marked edges or vertices were removed from the graph, with minimum overhead.

Note: It is important to emphasize that the filtering functionality does not add any overhead when the graph is not being filtered. In this case, the algorithms run just as fast as if the filtering functionality didn't exist.

Here is an example which obtains the minimum spanning tree of a graph, using the function `min_spanning_tree()` (page 198) and edge filtering.

```
g, pos = triangulation(random((500, 2)) * 4, type="delaunay")
tree = min_spanning_tree(g)
graph_draw(g, pos=pos, edge_color=tree, output="min_tree.pdf")
```

The `tree` property map has a bool type, with value "1" if the edge belongs to the tree, and "0" otherwise. Below is an image of the original graph, with the marked edges.

We can now filter out the edges which don't belong to the minimum spanning tree.

```
g.set_edge_filter(tree)
graph_draw(g, pos=pos, output="min_tree_filtered.pdf")
```

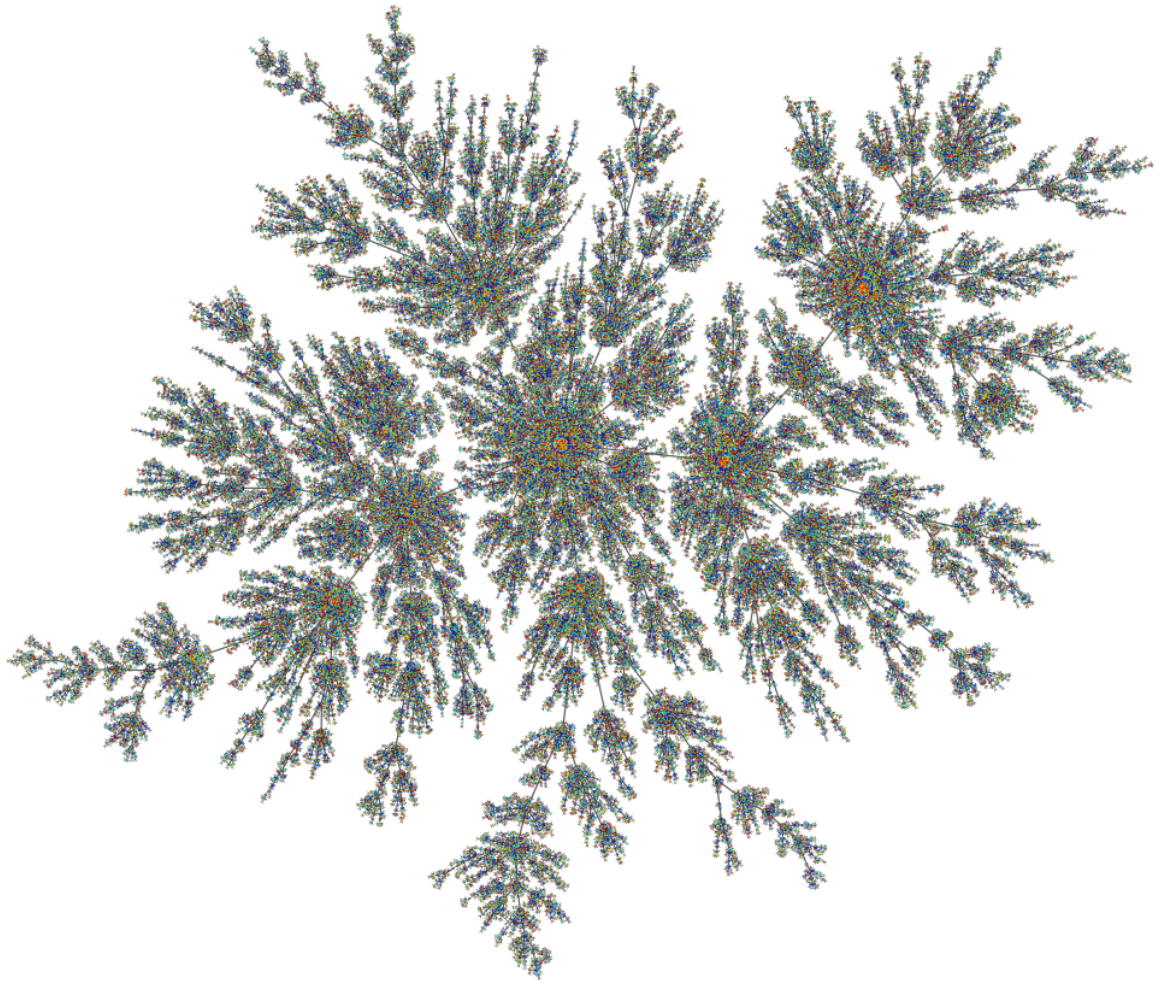
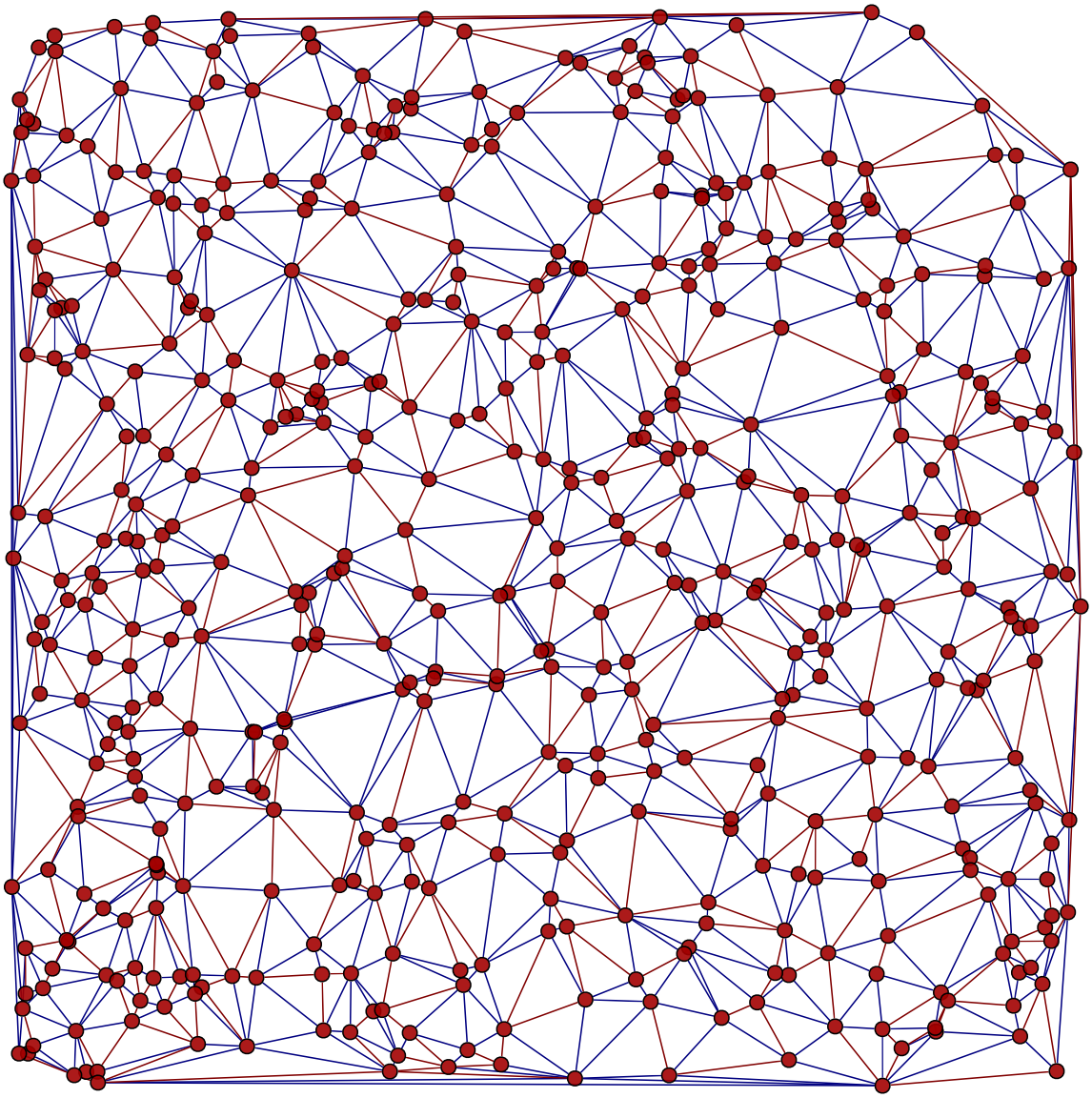
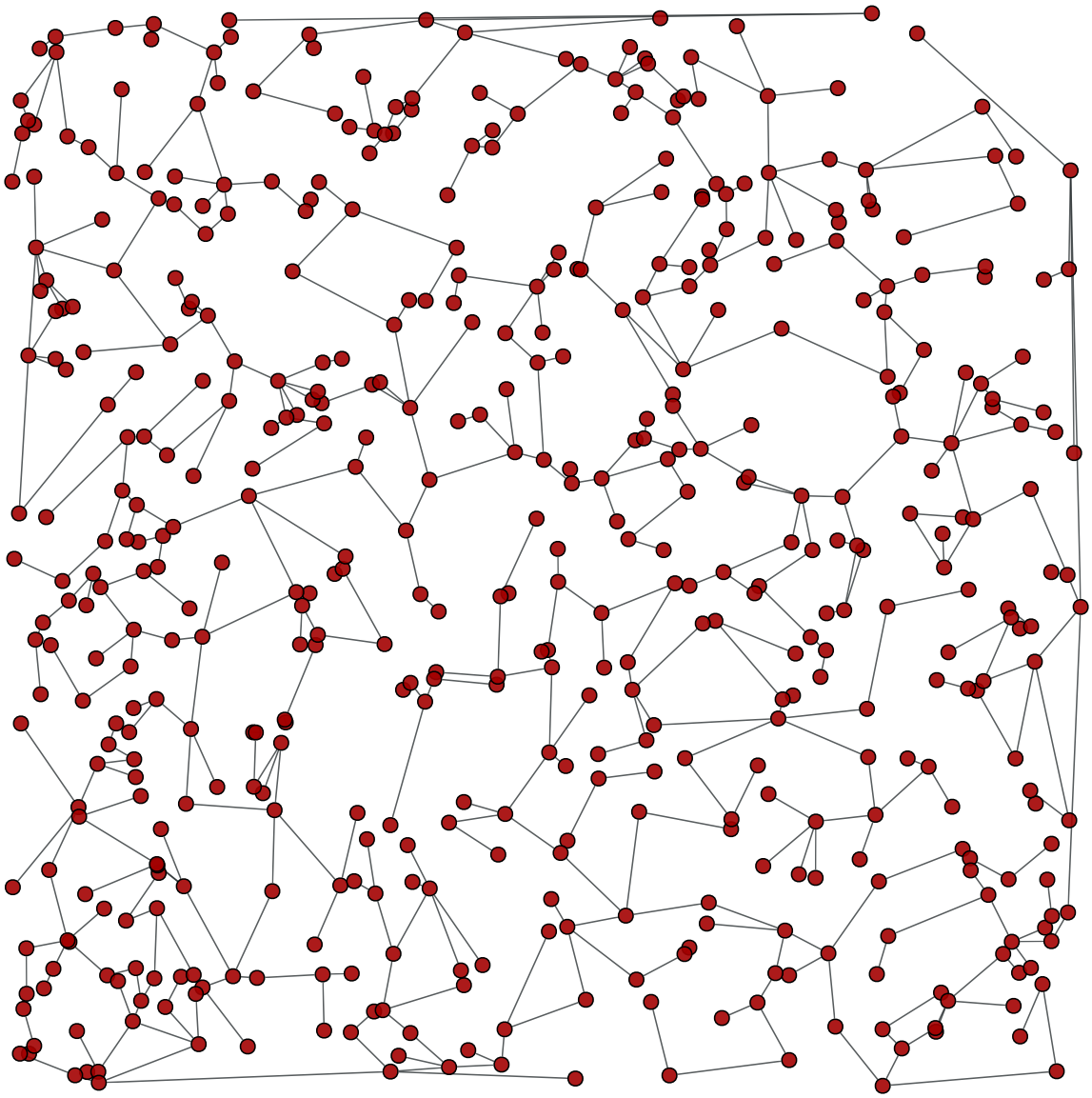


Figure 1.3: A Price network with 10^5 nodes. The vertex colors represent the age of the vertex, from older (red) to newer (blue).



This is how the graph looks when filtered:



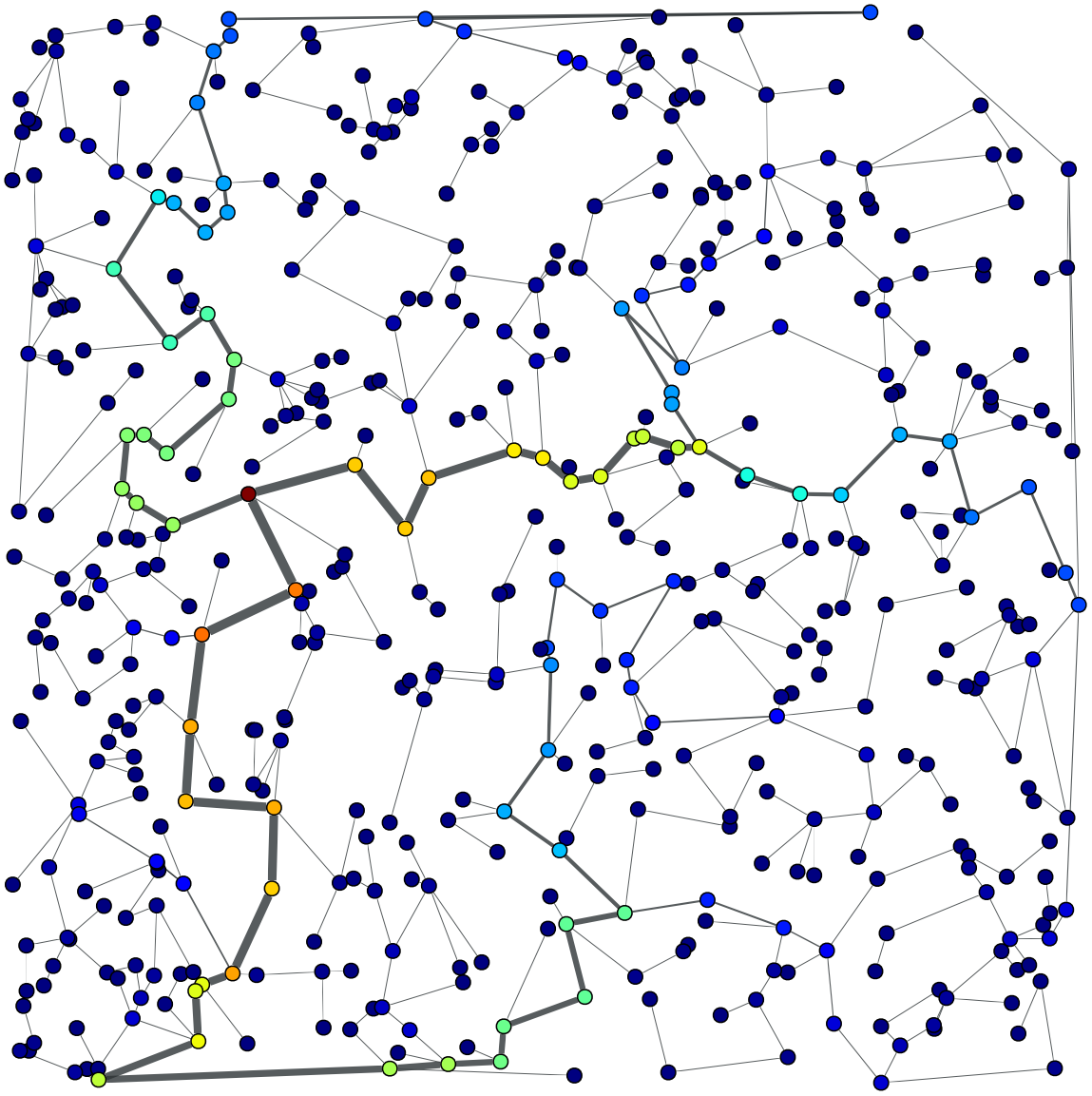
Everything should work transparently on the filtered graph, simply as if the masked edges were removed. For instance, the following code will calculate the `betweenness()` (page 42) centrality of the edges and vertices, and draws them as colors and line thickness in the graph.

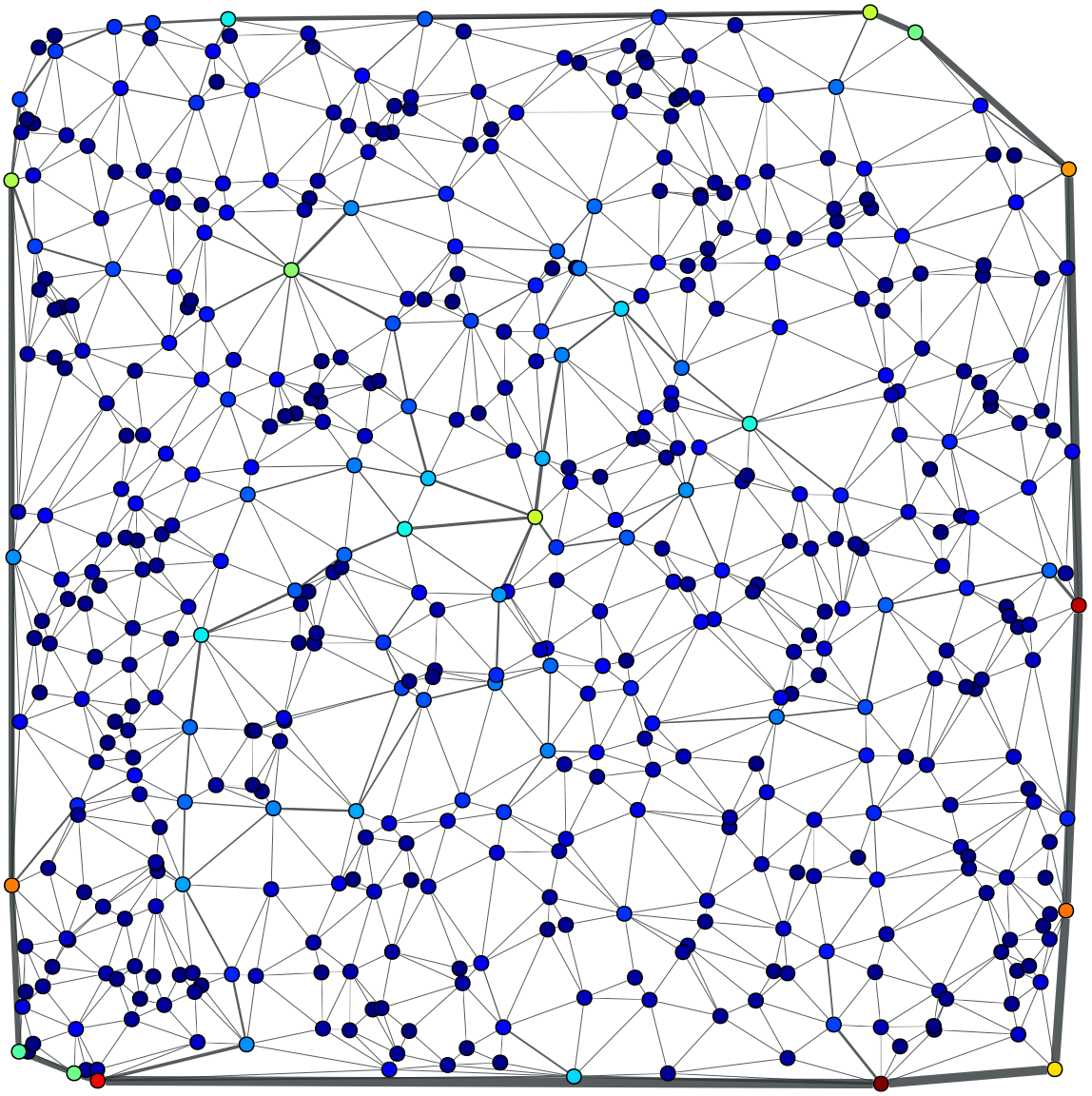
```
bv, be = betweenness(g)
be.a /= be.a.max() / 5
graph_draw(g, pos=pos, vertex_fill_color=bv, edge_pen_width=be,
           output="filtered-bt.pdf")
```

The original graph can be recovered by setting the edge filter to `None`.

```
g.set_edge_filter(None)
bv, be = betweenness(g)
be.a /= be.a.max() / 5
graph_draw(g, pos=pos, vertex_fill_color=bv, edge_pen_width=be,
           output="nonfiltered-bt.pdf")
```

Everything works in analogous fashion with vertex filtering.





Additionally, the graph can also have its edges reversed with the `set_reversed()` (page 30) method. This is also an $O(1)$ operation, which does not really modify the graph.

As mentioned previously, the directedness of the graph can also be changed "on-the-fly" with the `set_directed()` (page 29) method.

1.5.1 Graph views

It is often desired to work with filtered and unfiltered graphs simultaneously, or to temporarily create a filtered version of graph for some specific task. For these purposes, graph-tool provides a `GraphView` (page 33) class, which represents a filtered "view" of a graph, and behaves as an independent graph object, which shares the underlying data with the original graph. Graph views are constructed by instantiating a `GraphView` (page 33) class, and passing a graph object which is supposed to be filtered, together with the desired filter parameters. For example, to create a directed view of the graph `g` constructed above, one should do:

```
>>> ug = GraphView(g, directed=True)
>>> ug.is_directed()
True
```

Graph views also provide a much more direct and convenient approach to vertex/edge filtering: To construct a filtered minimum spanning tree like in the example above, one must only pass the filter property as the "efilter" parameter:

```
>>> tv = GraphView(g, efilt=tree)
```

Note that this is an $O(1)$ operation, since it is equivalent (in speed) to setting the filter in graph `g` directly, but in this case the object `g` remains unmodified.

Like above, the result should be the isolated minimum spanning tree:

```
>>> bv, be = betweenness(tv)
>>> be.a /= be.a.max() / 5
>>> graph_draw(tv, pos=pos, vertex_fill_color=bv,
...            edge_pen_width=be, output="mst-view.pdf")
<...>
```

Note: `GraphView` (page 33) objects behave *exactly* like regular `Graph` (page 28) objects. In fact, `GraphView` (page 33) is a subclass of `Graph` (page 28). The only difference is that a `GraphView` (page 33) object shares its internal data with its parent `Graph` (page 28) class. Therefore, if the original `Graph` (page 28) object is modified, this modification will be reflected immediately in the `GraphView` (page 33) object, and vice-versa.

For even more convenience, one can supply a function as filter parameter, which takes a vertex or an edge as single parameter, and returns `True` if the vertex/edge should be kept and `False` otherwise. For instance, if we want to keep only the most "central" edges, we can do:

```
>>> bv, be = betweenness(g)
>>> u = GraphView(g, efilt=lambda e: be[e] > be.a.max() / 2)
```

This creates a graph view `u` which contains only the edges of `g` which have a normalized betweenness centrality larger than half of the maximum value. Note that, differently from the case above, this is an $O(E)$ operation, where E is the number of edges, since the supplied function must be called E times to construct a filter property map. Thus, supplying a constructed filter map is always faster, but supplying a function can be more convenient.

The graph view constructed above can be visualized as

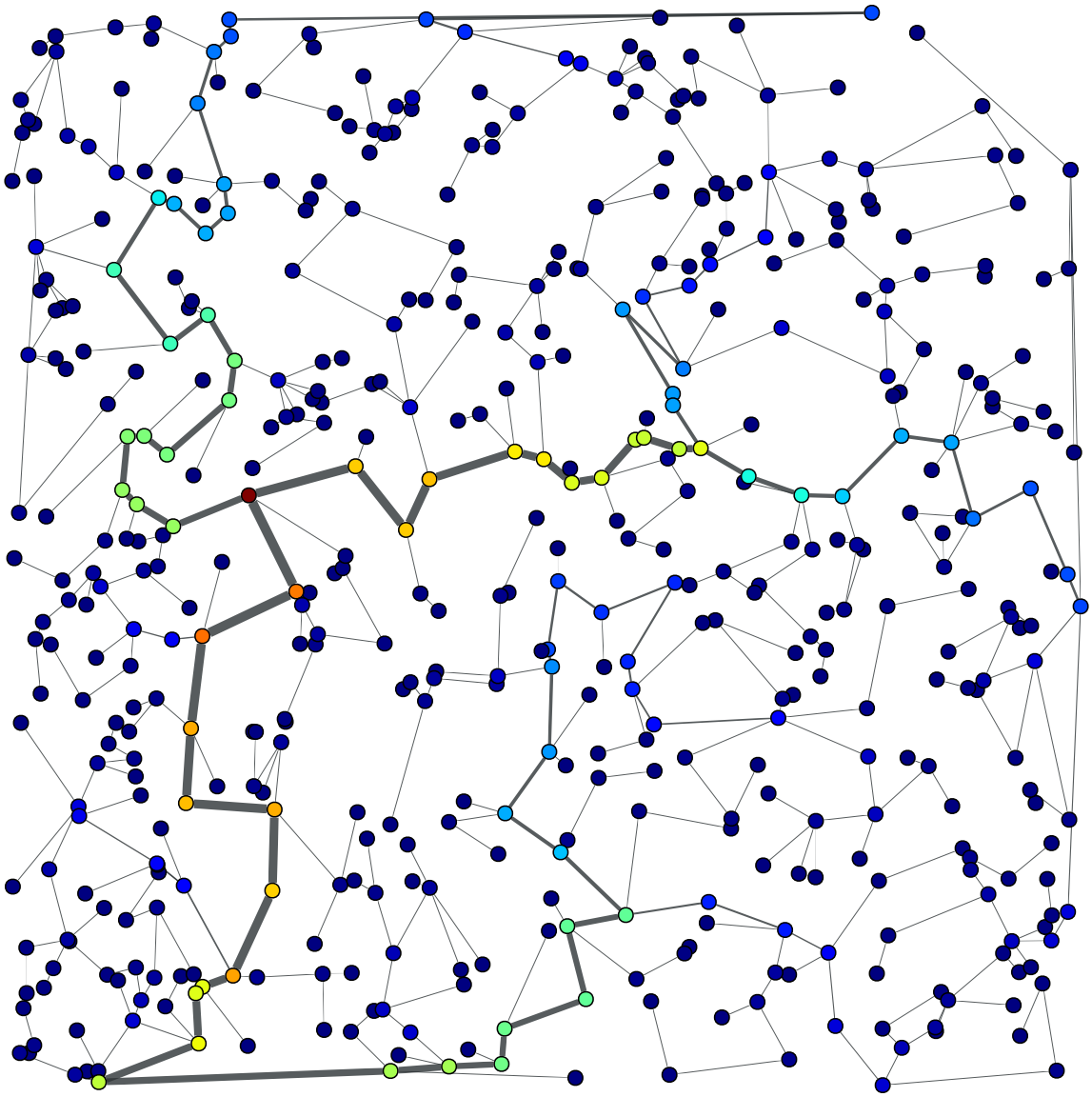


Figure 1.4: A view of the Delaunay graph, isolating only the minimum spanning tree.

```
>>> be.a /= be.a.max() / 5
>>> graph_draw(u, pos=pos, vertex_fill_color=bv, output="central-edges-view.pdf")
<...>
```

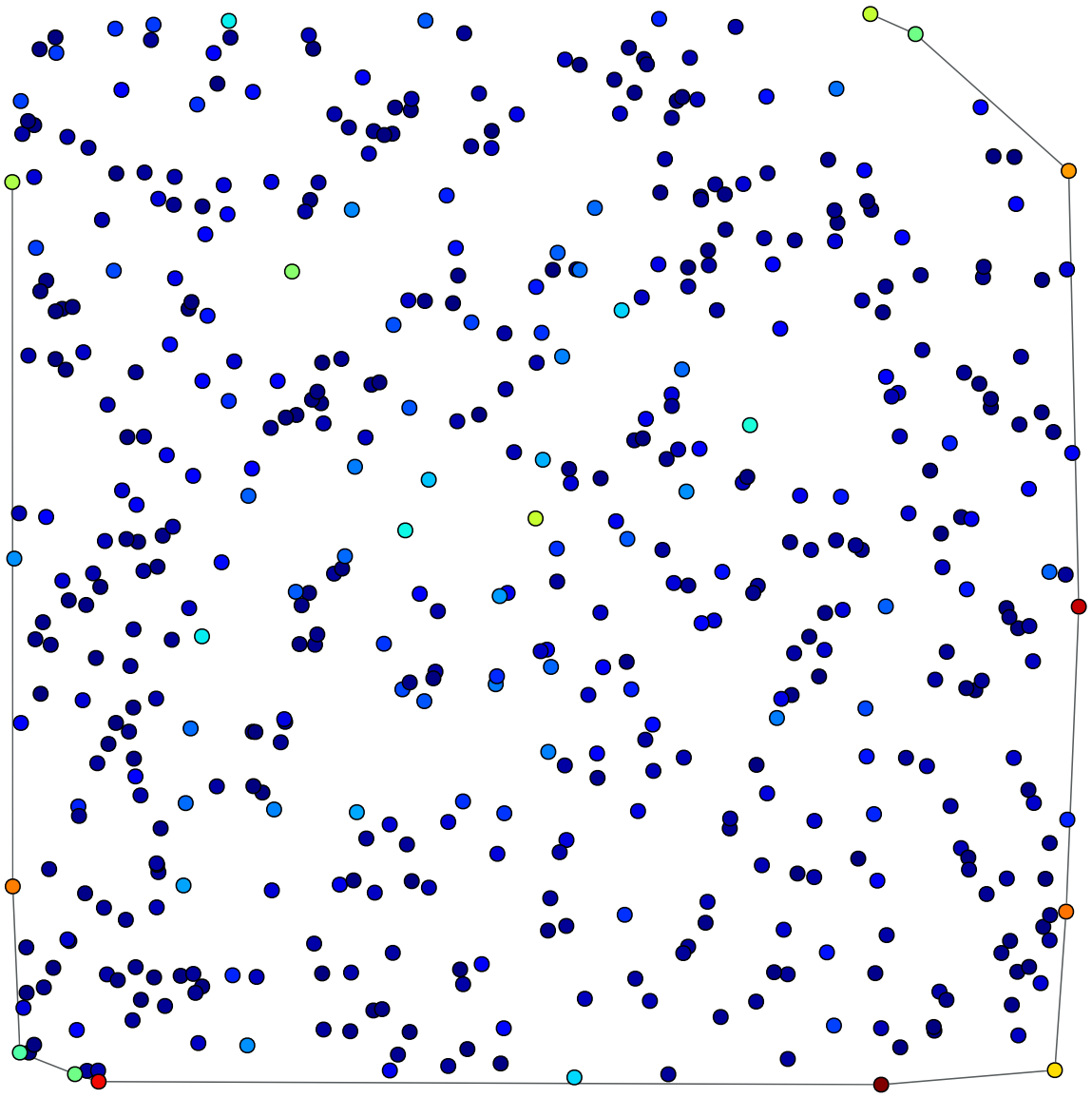


Figure 1.5: A view of the Delaunay graph, isolating only the edges with normalized betweenness centrality larger than 0.01.

Composing graph views

Since graph views are regular graphs, one can just as easily create graph views *of graph views*. This provides a convenient way of composing filters. For instance, in order to isolate the minimum spanning tree of all vertices of the example above which have a degree larger than four, one can do:

```
>>> u = GraphView(g, vfilt=lambda v: v.out_degree() > 4)
>>> tree = min_spanning_tree(u)
>>> u = GraphView(u, efilt=tree)
```

The resulting graph view can be visualized as

```
>>> graph_draw(u, pos=pos, output="composed-filter.pdf")
<...>
```

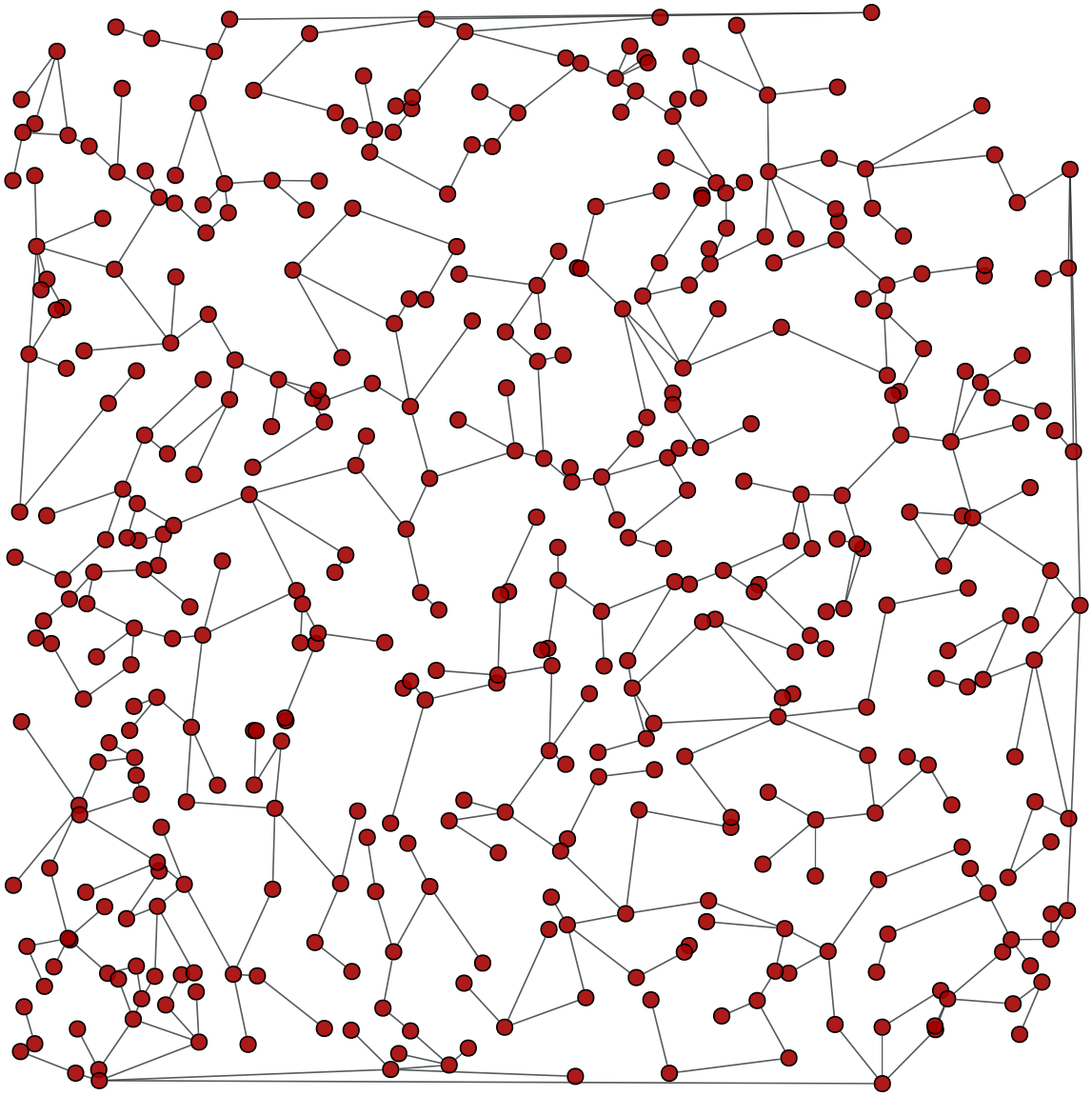


Figure 1.6: A composed view, obtained as the minimum spanning tree of all vertices in the graph which have a degree larger than four.

CHAPTER

TWO

COOKBOOK

Contents:

2.1 Animations with graph-tool

The drawing capabilities of `graph-tool` (see `draw` (page 102) module) can be harnessed to perform animations in a straightforward manner. Here we show some examples which uses GTK+ to display animations in an `interactive_window` (page 122), as well as offscreen to a file. The idea is to easily generate visualisations which can be used in presentations, and embedded in websites.

2.1.1 SIRS epidemics

Here we implement a simple SIRS epidemics on a network, and we construct an animation showing the time evolution. Nodes which are susceptible (S) are shown in white, whereas infected (I) nodes are shown in black. Recovered (R) nodes are removed from the layout, since they cannot propagate the outbreak.

The script which performs the animation is called `animation_sirs.py` and is shown below.

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

# This simple example on how to do animations using graph-tool. Here we do a
# simple simulation of an S->I->R->S epidemic model, where each vertex can be in
# one of the following states: Susceptible (S), infected (I), recovered (R). A
# vertex in the S state becomes infected either spontaneously with a probability
# 'x' or because a neighbour is infected. An infected node becomes recovered
# with probability 'r', and a recovered vertex becomes again susceptible with
# probability 's'.

# DISCLAIMER: The following code is definitely not the most efficient approach
# if you want to simulate this dynamics for very large networks, and/or for very
# long times. The main purpose is simply to highlight the animation capabilities
# of graph-tool.

from graph_tool.all import *
from numpy.random import *
import sys, os, os.path

seed(42)
seed_rng(42)

# We need some Gtk and gobject functions
from gi.repository import Gtk, Gdk, GdkPixbuf, GObject
```

```
26
27  # We will use the network of network scientists, and filter out the largest
28  # component
29  g = collection.data["netscience"]
30  g = GraphView(g, vfilt=label_largest_component(g), directed=False)
31  g = Graph(g, prune=True)
32
33  pos = g.vp["pos"]  # layout positions
34
35  # We will filter out vertices which are in the "Recovered" state, by masking
36  # them using a property map.
37  removed = g.new_vertex_property("bool")
38
39  # SIRS dynamics parameters:
40
41  x = 0.001    # spontaneous outbreak probability
42  r = 0.1      # I->R probability
43  s = 0.01     # R->S probability
44
45  # (Note that the S->I transition happens simultaneously for every vertex with a
46  #  probability equal to the fraction of non-recovered neighbours which are
47  #  infected.)
48
49  # The states would usually be represented with simple integers, but here we will
50  # use directly the color of the vertices in (R,G,B,A) format.
51
52  S = [1, 1, 1, 1]           # White color
53  I = [0, 0, 0, 1]           # Black color
54  R = [0.5, 0.5, 0.5, 1.]    # Grey color (will not actually be drawn)
55
56  # Initialize all vertices to the S state
57  state = g.new_vertex_property("vector<double>")
58  for v in g.vertices():
59      state[v] = S
60
61  # Newly infected nodes will be highlighted in red
62  newly_infected = g.new_vertex_property("bool")
63
64  # If True, the frames will be dumped to disk as images.
65  offscreen = sys.argv[1] == "offscreen" if len(sys.argv) > 1 else False
66  max_count = 500
67  if offscreen and not os.path.exists("./frames"):
68      os.mkdir("./frames")
69
70  # This creates a GTK+ window with the initial graph layout
71  if not offscreen:
72      win = GraphWindow(g, pos, geometry=(500, 400),
73                        edge_color=[0.6, 0.6, 0.6, 1],
74                        vertex_fill_color=state,
75                        vertex_halo=newly_infected,
76                        vertex_halo_color=[0.8, 0, 0, 0.6])
77  else:
78      count = 0
79      win = Gtk.OffscreenWindow()
80      win.set_default_size(500, 400)
81      win.graph = GraphWidget(g, pos,
82                              edge_color=[0.6, 0.6, 0.6, 1],
83                              vertex_fill_color=state,
84                              vertex_halo=newly_infected,
85                              vertex_halo_color=[0.8, 0, 0, 0.6])
86      win.add(win.graph)
87
88
```

```
89  # This function will be called repeatedly by the GTK+ main loop, and we use it
90  # to update the state according to the SIRS dynamics.
91  def update_state():
92      newly_infected.a = False
93      removed.a = False
94
95      # visit the nodes in random order
96      vs = list(g.vertices())
97      shuffle(vs)
98      for v in vs:
99          if state[v] == I:
100             if random() < r:
101                 state[v] = R
102         elif state[v] == S:
103             if random() < x:
104                 state[v] = I
105             else:
106                 ns = list(v.out_neighbours())
107                 if len(ns) > 0:
108                     w = ns[randint(0, len(ns))]  # choose a random neighbour
109                     if state[w] == I:
110                         state[v] = I
111                         newly_infected[v] = True
112         elif random() < s:
113             state[v] = S
114         if state[v] == R:
115             removed[v] = True
116
117     # Filter out the recovered vertices
118     g.set_vertex_filter(removed, inverted=True)
119
120     # The following will force the re-drawing of the graph, and issue a
121     # re-drawing of the GTK window.
122     win.graph.regenerate_surface(lazy=False)
123     win.graph.queue_draw()
124
125     # if doing an offscreen animation, dump frame to disk
126     if offscreen:
127         global count
128         pixbuf = win.get_pixbuf()
129         pixbuf.savev(r'./frames/sirs%06d.png' % count, 'png', [], [])
130         if count > max_count:
131             sys.exit(0)
132         count += 1
133
134     # We need to return True so that the main loop will call this function more
135     # than once.
136     return True
137
138
139 # Bind the function above as an 'idle' callback.
140 cid = GObject.idle_add(update_state)
141
142 # We will give the user the ability to stop the program by closing the window.
143 win.connect("delete_event", Gtk.main_quit)
144
145 # Actually show the window, and start the main loop.
146 win.show_all()
147 Gtk.main()
```

If called without arguments, the script will show the animation inside an `interactive_window` (page 122). If the parameter `offscreen` is passed, individual frames will be saved in the `frames` directory:

```
$ ./animation_sirs.py offscreen
```

These frames can be combined and encoded into the appropriate format. Here we use the mencoder tool from mplayer to combine all the frames into a single file with YUY format, and then we encode this with the WebM format, using vpxenc, so that it can be embedded in a website.

```
$ mencoder mf://frames/sirs*.png -mf w=500:h=400:type=png -ovc raw -of rawvideo -vf format=i420 -
$ vpxenc sirs.yuy -o sirs.webm -w 500 -h 400 --fps=25/1 --target-bitrate=1000 --good --threads=4
```

The resulting animation can be downloaded here, or played below if your browser supports WebM.

This type of animation can be extended or customized in many ways, by dynamically modifying the various drawing parameters and vertex/edge properties. For instance, one might want to represent the susceptible state as either or , depending on whether a neighbor is infected, and the infected state as . Properly modifying the script above would lead to the following movie:

The modified script can be downloaded here.

2.1.2 Dynamic layout

The graph layout can also be updated during an animation. As an illustration, here we consider a very simplistic model for spatial segregation, where the edges of the graph are repeatedly and randomly rewired, as long as the new edge has a shorter euclidean distance.

The script which performs the animation is called animation_dancing.py and is shown below.

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

# This simple example on how to do animations using graph-tool, where the layout
# changes dynamically. We start with some network, and randomly rewire its
# edges, and update the layout dynamically, where edges are rewired only if
# their euclidean distance is reduced. It is thus a very simplistic model for
# spatial segregation.

from graph_tool.all import *
from numpy.random import *
from numpy.linalg import norm
import sys, os, os.path

seed(42)
seed_rng(42)
```

```
17
18  # We need some Gtk and gobject functions
19  from gi.repository import Gtk, Gdk, GdkPixbuf, GObject
20
21  # We will generate a small random network
22  g = random_graph(150, lambda: 1 + poisson(5), directed=False)
23
24  # Parameters for the layout update
25
26  step = 0.005        # move step
27  K = 0.5             # preferred edge length
28
29  pos = sfdp_layout(g, K=K)  # initial layout positions
30
31  # If True, the frames will be dumped to disk as images.
32  offscreen = sys.argv[1] == "offscreen" if len(sys.argv) > 1 else False
33  max_count = 5000
34  if offscreen and not os.path.exists("./frames"):
35      os.mkdir("./frames")
36
37  # This creates a GTK+ window with the initial graph layout
38  if not offscreen:
39      win = GraphWindow(g, pos, geometry=(500, 400))
40  else:
41      win = Gtk.OffscreenWindow()
42      win.set_default_size(500, 400)
43      win.graph = GraphWidget(g, pos)
44      win.add(win.graph)
45
46  # list of edges
47  edges = list(g.edges())
48
49  count = 0
50
51  # This function will be called repeatedly by the GTK+ main loop, and we use it
52  # to update the vertex layout and perform the rewiring.
53  def update_state():
54      global count
55
56      # Perform one iteration of the layout step, starting from the previous positions
57      sfdp_layout(g, pos=pos, K=K, init_step=step, max_iter=1)
58
59      for i in range(100):
60          # get a chosen edge, and swap one of its end points for a random vertex,
61          # if it is closer
62          i = randint(0, len(edges))
63          e = list(edges[i])
64          shuffle(e)
65          s1, t1 = e
66
67          t2 = g.vertex(randint(0, g.num_vertices()))
68
69          if (norm(pos[s1].a - pos[t2].a) <= norm(pos[s1].a - pos[t1].a) and
70              s1 != t2 and                    # no self-loops
71              t1.out_degree() > 1 and         # no isolated vertices
72              t2 not in s1.out_neighbours()):   # no parallel edges
73
74              g.remove_edge(edges[i])
75              edges[i] = g.add_edge(s1, t2)
76
77
78      # The movement of the vertices may cause them to leave the display area. The
79      # following function rescales the layout to fit the window to avoid this.
```

```
80      if count % 1000 == 0:
81          win.graph.fit_to_window(ink=True)
82      count += 1
83
84      # The following will force the re-drawing of the graph, and issue a
85      # re-drawing of the GTK window.
86      win.graph.regenerate_surface(lazy=False)
87      win.graph.queue_draw()
88
89      # if doing an offscreen animation, dump frame to disk
90      if offscreen:
91          pixbuf = win.get_pixbuf()
92          pixbuf.savev(r'./frames/dancing%06d.png' % count, 'png', [], [])
93          if count > max_count:
94              sys.exit(0)
95
96      # We need to return True so that the main loop will call this function more
97      # than once.
98      return True
99
100
101 # Bind the function above as an 'idle' callback.
102 cid = GObject.idle_add(update_state)
103
104 # We will give the user the ability to stop the program by closing the window.
105 win.connect("delete_event", Gtk.main_quit)
106
107 # Actually show the window, and start the main loop.
108 win.show_all()
109 Gtk.main()
```

This example works like the SIRS example above, and if we pass the `offscreen` parameter, the frames will be dumped to disk, otherwise the animation is displayed inside an `interactive_window` (page 122).

```
$ ./animation_dancing.py offscreen
```

Also like the previous example, we can encode the animation with the WebM format:

```
$ mencoder mf://frames/dancing*.png -mf w=500:h=400:type=png -ovc raw -of rawvideo -vf format=i42
$ vpxenc sirs.yuy -o dancing.webm -w 500 -h 400 --fps=100/1 --target-bitrate=5000 --good --threads
```

The resulting animation can be downloaded `here`, or played below if your browser supports WebM.

CHAPTER

THREE

MODULE DOCUMENTATION

3.1 graph_tool - efficient graph analysis and manipulation

3.1.1 Summary

`Graph` (page 28)	Generic multigraph class.
`GraphView` (page 33)	A view of selected vertices or edges of another graph.
`Vertex` (page 33)	Vertex descriptor.
`Edge` (page 34)	Edge descriptor.
`PropertyMap` (page 35)	This class provides a mapping from vertices, edges or whole graph
`PropertyArray` (page 36)	This is a `ndarray` subclass which keeps a reference of its `Propert`
`load_graph` (page 36)	Load a graph from `file_name` (which can be either a string or a fi
`group_vector_property` (page 36)	Group list of properties `props` into a vector property map of the sa
`ungroup_vector_property` (page 37)	Ungroup vector property map `vprop` into a list of individual prope
`infect_vertex_property` (page 38)	Propagate the *prop* values of vertices with value *val* to all their out
`edge_difference` (page 38)	Return an edge property map corresponding to the difference betv
`value_types` (page 39)	Return a list of possible properties value types.
`show_config` (page 39)	Show `graph_tool` build configuration.

This module provides:

1. A `Graph` (page 28) class for graph representation and manipulation
2. Property maps for Vertex, Edge or Graph.
3. Fast algorithms implemented in C++.

3.1.2 How to use the documentation

Documentation is available in two forms: docstrings provided with the code, and the full documentation available in the graph-tool homepage.

We recommend exploring the docstrings using IPython, an advanced Python shell with TAB-completion and introspection capabilities.

The docstring examples assume that `graph_tool.all` has been imported as `gt`:

```
>>> import graph_tool.all as gt
```

Code snippets are indicated by three greater-than signs:

```
>>> x = x + 1
```

Use the built-in `help` function to view a function's docstring:

```
>>> help(gt.Graph)
```

3.1.3 Contents

class graph_tool.**Graph**(*g=None*, *directed=True*, *prune=False*)
Generic multigraph class.

This class encapsulates either a directed multigraph (default or if directed=True) or an undirected multigraph (if directed=False), with optional internal edge, vertex or graph properties.

If g is specified, the graph (and its internal properties) will be copied.

If prune is set to True, and g is specified, only the filtered graph will be copied, and the new graph object will not be filtered. Optionally, a tuple of three booleans can be passed as value to prune, to specify a different behavior to vertex, edge, and reversal filters, respectively.

The graph is implemented as an adjacency list, where both vertex and edge lists are C++ STL vectors.

copy()
Return a deep copy of self. All *internal property maps* (page 7) are also copied.

Iterating over vertices and edges

See *Iterating over vertices and edges* (page 6) for more documentation and examples.

vertices()
Return an iterator over the vertices.

Note: The order of the vertices traversed by the iterator **always** corresponds to the vertex index ordering, as given by the vertex_index (page 30) property map.

Examples

```
>>> g = gt.Graph()
>>> vlist = list(g.add_vertex(5))
>>> vlist2 = []
>>> for v in g.vertices():
...     vlist2.append(v)
...
>>> assert(vlist == vlist2)
```

edges()
Return an iterator over the edges.

Note: The order of the edges traversed by the iterator **does not** necessarily correspond to the edge index ordering, as given by the edge_index (page 30) property map. This will only happen after reindex_edges() (page 30) is called, or in certain situations such as just after a graph is loaded from a file. However, further manipulation of the graph may destroy the ordering.

Obtaining vertex and edge descriptors

vertex(*i*, *use_index=True*)
Return the vertex with index i. If use_index=False, the i-th vertex is returned (which can differ from the vertex with index i in case of filtered graphs).

edge(*s*, *t*, *all_edges=False*)
Return the edge from vertex `s` to `t`, if it exists. If `all_edges=True` then a list is returned with all the parallel edges from `s` to `t`, otherwise only one edge is returned.

This operation will take $O(k(s))$ time, where $k(s)$ is the out-degree of vertex s.

Number of vertices and edges

num_vertices()
Get the number of vertices.

Note: If the vertices are being filtered, this operation is $O(N)$. Otherwise it is $O(1)$.

---

num_edges()
Get the number of edges.

Note: If the edges are being filtered, this operation is $O(E)$. Otherwise it is $O(1)$.

Modifying vertices and edges

The following functions allow for addition and removal of vertices in the graph.

add_vertex(*n=1*)
Add a vertex to the graph, and return it. If `n > 1`, `n` vertices are inserted and an iterator over the new vertices is returned.

remove_vertex(*vertex*, *fast=False*)
Remove a vertex from the graph.

Note: If the option `fast == False` is given, this operation is $O(N + E)$ (this is the default). Otherwise it is $O(k + k_{\text{last}})$, where k is the (total) degree of the vertex being deleted, and k_{last} is the (total) degree of the vertex with the largest index.

Warning: If `fast == True`, the vertex being deleted is 'swapped' with the last vertex (i.e. with the largest index), which will in turn inherit the index of the vertex being deleted. All property maps associated with the graph will be properly updated, but the index ordering of the graph will no longer be the same.

The following functions allow for addition and removal of edges in the graph.

add_edge(*source*, *target*)
Add a new edge from `source` to `target` to the graph, and return it.

remove_edge(*edge*)
Remove an edge from the graph.

The following functions allow for easy removal of vertices of edges from the graph.

clear()
Remove all vertices and edges from the graph.

clear_vertex(*vertex*)
Remove all in and out-edges from the given vertex.

clear_edges()
Remove all edges from the graph.

Directedness and reversal of edges

Note: These functions do not actually modify the graph, and are fully reversible. They are also very cheap, and have an $O(1)$ complexity.

---

set_directed(*is_directed*)
Set the directedness of the graph.

is_directed()
Get the directedness of the graph.

set_reversed(*is_reversed*)
Reverse the direction of the edges, if `is_reversed` is `True`, or maintain the original direction otherwise.

is_reversed()
Return `True` if the edges are reversed, and `False` otherwise.

Creation of new property maps

new_property(*key_type*, *value_type*)
Create a new (uninitialized) vertex property map of key type `key_type` (`v`, `e` or `g`), value type `value_type`, and return it.

new_vertex_property(*value_type*)
Create a new (uninitialized) vertex property map of type `value_type`, and return it.

new_edge_property(*value_type*)
Create a new (uninitialized) edge property map of type `value_type`, and return it.

new_graph_property(*value_type*, *val=None*)
Create a new graph property map of type `value_type`, and return it. If `val` is not None, the property is initialized to its value.

New property maps can be created by copying already existing ones.

copy_property(*src*, *tgt=None*, *value_type=None*, *g=None*)
Copy contents of `src` property to `tgt` property. If `tgt` is None, then a new property map of the same type (or with the type given by the optional `value_type` parameter) is created, and returned. The optional parameter g specifies the (identical) source graph to copy properties from (defaults to self).

degree_property_map(*deg*)
Create and return a vertex property map containing the degree type given by `deg`.

Index property maps

vertex_index
Vertex index map.

It maps for each vertex in the graph an unique integer in the range [0, `num_vertices()` (page 29) - 1].

Note: Like `edge_index` (page 30), this is a special instance of a `PropertyMap` (page 35) class, which is **immutable**, and cannot be accessed as an array.

---

edge_index
Edge index map.

It maps for each edge in the graph an unique integer.

Note: Like `vertex_index` (page 30), this is a special instance of a `PropertyMap` (page 35) class, which is **immutable**, and cannot be accessed as an array.

Additionally, the indexes may not necessarily lie in the range [0, `num_edges()` (page 29) - 1]. However this will always happen whenever no edges are deleted from the graph.

---

max_edge_index
The maximum value of the edge index map.

reindex_edges()
Reset the edge indexes so that they lie in the [0, num_edges() (page 29) - 1] range. The index ordering will be compatible with the sequence returned by the edges() (page 28) function.

> **Warning:** Calling this function will invalidate all existing edge property maps, if the index ordering is modified! The property maps will still be usable, but their contents will still be tied to the old indexes, and thus may become scrambled.

Internal property maps

Internal property maps are just like regular property maps, with the only exception that they are saved and loaded to/from files together with the graph itself. See *internal property maps* (page 7) for more details.

Note: All dictionaries below are mutable. However, any dictionary returned below is only an one-way proxy to the internally-kept properties. If you modify this object, the change will be propagated to the internal dictionary, but not vice-versa. Keep this in mind if you intend to keep a copy of the returned object.

---

properties
Dictionary of internal properties. Keys must always be a tuple, where the first element if a string from the set {'v', 'e', 'g'}, representing a vertex, edge or graph property, respectively, and the second element is the name of the property map.

Examples

```
>>> g = gt.Graph()
>>> g.properties[("e", "foo")] = g.new_edge_property("vector<double>")
>>> del g.properties[("e", "foo")]
```

vertex_properties
Dictionary of internal vertex properties. The keys are the property names.

vp
Alias to vertex_properties (page 31).

edge_properties
Dictionary of internal edge properties. The keys are the property names.

ep
Alias to edge_properties (page 31).

graph_properties
Dictionary of internal graph properties. The keys are the property names.

gp
Alias to graph_properties (page 31).

list_properties()
Print a list of all internal properties.

Examples

```
>>> g = gt.Graph()
>>> g.properties[("e", "foo")] = g.new_edge_property("vector<double>")
>>> g.vertex_properties["foo"] = g.new_vertex_property("double")
>>> g.vertex_properties["bar"] = g.new_vertex_property("python::object")
>>> g.graph_properties["gnat"] = g.new_graph_property("string", "hi there!")
```

```
>>> g.list_properties()
gnat            (graph)   (type: string, val: hi there!)
foo             (vertex)  (type: double)
bar             (vertex)  (type: python::object)
foo             (edge)    (type: vector<double>)
```

Filtering of vertices and edges.

See *Graph filtering* (page 11) for more details.

Note: These functions do not actually modify the graph, and are fully reversible. They are also very cheap, and have an $O(1)$ complexity.

---

set_vertex_filter(*prop*, *inverted=False*)
Choose vertex boolean filter property. Only the vertices with value different than zero are kept in the filtered graph. If the `inverted` option is supplied with value `True`, only the vertices with value zero are kept. If the supplied property is `None`, any previous filtering is removed.

get_vertex_filter()
Return a tuple with the vertex filter property and bool value indicating whether or not it is inverted.

set_edge_filter(*prop*, *inverted=False*)
Choose edge boolean filter property. Only the edges with value different than zero are kept in the filtered graph. If the `inverted` option is supplied with value `True`, only the edges with value zero are kept. If the supplied property is `None`, any previous filtering is removed.

get_edge_filter()
Return a tuple with the edge filter property and bool value indicating whether or not it is inverted.

Warning: The purge functions below irreversibly remove the filtered vertices or edges from the graph, and return it to an unfiltered state. Note that, contrary to the functions above, these are $O(V)$ and $O(E)$ operations, respectively.

purge_vertices(*in_place=False*)
Remove all vertices of the graph which are currently being filtered out, and return it to the unfiltered state. This operation is not reversible.

If the option `in_place == True` is given, the algorithm will remove the filtered vertices and re-index all property maps which are tied with the graph. This is a slow operation which has an $O(N^2)$ complexity.

If `in_place == False`, the graph and its vertex and edge property maps are temporarily copied to a new unfiltered graph, which will replace the contents of the original graph. This is a fast operation with an $O(N + E)$ complexity. This is the default behaviour if no option is given.

purge_edges()
Remove all edges of the graph which are currently being filtered out, and return it to the unfiltered state. This operation is not reversible.

Stashing and popping the filter state

stash_filter(*edge=False*, *vertex=False*, *directed=False*, *reversed=False*, *all=True*)
Stash current filter state and set the graph to its unfiltered state. The optional keyword arguments specify which type of filter should be stashed.

pop_filter(*edge=False*, *vertex=False*, *directed=False*, *reversed=False*, *all=True*)
Pop last stashed filter state. The optional keyword arguments specify which type of filter should be recovered.

get_filter_state()
Return a copy of the filter state of the graph.

set_filter_state(*state*)
Set the filter state of the graph.

I/O operations

See *Graph I/O* (page 8) for more details.

load(*file_name*, *fmt='auto'*, *ignore_vp=None*, *ignore_ep=None*, *ignore_gp=None*)
Load graph from `file_name` (which can be either a string or a file-like object). The format is guessed from `file_name`, or can be specified by `fmt`, which can be either "xml", "dot" or "gml".

If provided, the parameters `ignore_vp`, `ignore_ep` and `ignore_gp`, should contain a list of property names (vertex, edge or graph, respectively) which should be ignored when reading the file.

save(*file_name*, *fmt='auto'*)
Save graph to `file_name` (which can be either a string or a file-like object). The format is guessed from the `file_name`, or can be specified by `fmt`, which can be either "xml", "dot" or "gml".

class graph_tool.**GraphView**(*g*, *vfilt=None*, *efilt=None*, *directed=None*, *reversed=False*)
Bases: `graph_tool.Graph` (page 28)

A view of selected vertices or edges of another graph.

This class uses shared data from another `Graph` (page 28) instance, but allows for local filtering of vertices and/or edges, edge directionality or reversal. See *Graph views* (page 17) for more details and examples.

The existence of a `GraphView` (page 33) object does not affect the original graph, except if the graph view is modified (addition or removal of vertices or edges), in which case the modification is directly reflected in the original graph (and vice-versa), since they both point to the same underlying data. Because of this, instances of `PropertyMap` (page 35) can be used interchangeably with a graph and its views.

The argument `g` must be an instance of a `Graph` (page 28) class. If specified, `vfilt` and `efilt` select which vertices and edges are filtered, respectively. These parameters can either be a boolean-valued `PropertyMap` (page 35) or a `ndarray`, which specify which vertices/edges are selected, or an unary function which returns `True` if a given vertex/edge is to be selected, or `False` otherwise.

The boolean parameter `directed` can be used to set the directionality of the graph view. If `directed = None`, the directionality is inherited from `g`.

If `reversed = True`, the direction of the edges is reversed.

If `vfilt` or `efilt` is anything other than a `PropertyMap` (page 35) instance, the instantiation running time is $O(V)$ and $O(E)$, respectively. Otherwise, the running time is $O(1)$.

base
Base graph.

class graph_tool.**Vertex**
Vertex descriptor.

This class represents a vertex in a `Graph` (page 28) instance.

Vertex (page 33) instances are hashable, and are convertible to integers, corresponding to its index (see vertex_index (page 30)).

all_edges(*self*)
Return an iterator over all edges (both in or out).

all_neighbours(*self*)
Return an iterator over all neighbours (both in or out).

get_graph()

get_graph((Vertex)arg1) -> object : Return the graph to which the vertex belongs.

C++ signature : boost::python::api::object get_graph(graph_tool::PythonVertex {lvalue})

in_degree()

in_degree((Vertex)arg1) -> int : Return the in-degree.

C++ signature : unsigned long in_degree(graph_tool::PythonVertex {lvalue})

in_edges()

in_edges((Vertex)arg1) -> object : Return an iterator over the in-edges.

C++ signature : boost::python::api::object in_edges(graph_tool::PythonVertex {lvalue})

in_neighbours(*self*)
Return an iterator over the in-neighbours.

is_valid()

is_valid((Vertex)arg1) -> bool : Return whether the vertex is valid.

C++ signature : bool is_valid(graph_tool::PythonVertex {lvalue})

out_degree()

out_degree((Vertex)arg1) -> int : Return the out-degree.

C++ signature : unsigned long out_degree(graph_tool::PythonVertex {lvalue})

out_edges()

out_edges((Vertex)arg1) -> object : Return an iterator over the out-edges.

C++ signature : boost::python::api::object out_edges(graph_tool::PythonVertex {lvalue})

out_neighbours(*self*)
Return an iterator over the out-neighbours.

class graph_tool.**Edge**
Edge descriptor.

This class represents an edge in a Graph (page 28).

Edge (page 34) instances are hashable, and are convertible to a tuple, which contains the source and target vertices.

get_graph()

get_graph((Edge)arg1) -> object : Return the graph to which the edge belongs.

C++ signature : boost::python::api::object get_graph(graph_tool::PythonEdge<boost::Undirecte long>, graph_tool::detail::MaskFilter<boost::unchecked_vector_property_map<unsigned char, boost::adj_edge_index_property_map<unsigned long> > >, graph_tool::detail::MaskFilter<boost::unchecked_vector_property_map<unsigned char, boost::typed_identity_property_map<unsigned long> > > > > {lvalue})

is_valid()

is_valid((Edge)arg1) -> bool : Return whether the edge is valid.

C++ signature : bool is_valid(graph_tool::PythonEdge<boost::UndirectedAdaptor<boost::filtere
long>, graph_tool::detail::MaskFilter<boost::unchecked_vector_property_map<unsigned
char, boost::adj_edge_index_property_map<unsigned long> > >,
graph_tool::detail::MaskFilter<boost::unchecked_vector_property_map<unsigned
char, boost::typed_identity_property_map<unsigned long> > > > > {lvalue})

source()

source((Edge)arg1) -> object : Return the source vertex.

C++ signature : boost::python::api::object source(graph_tool::PythonEdge<boost::UndirectedA
long>, graph_tool::detail::MaskFilter<boost::unchecked_vector_property_map<unsigned
char, boost::adj_edge_index_property_map<unsigned long> > >,
graph_tool::detail::MaskFilter<boost::unchecked_vector_property_map<unsigned
char, boost::typed_identity_property_map<unsigned long> > > > > {lvalue})

target()

target((Edge)arg1) -> object : Return the target vertex.

C++ signature : boost::python::api::object target(graph_tool::PythonEdge<boost::UndirectedAd
long>, graph_tool::detail::MaskFilter<boost::unchecked_vector_property_map<unsigned
char, boost::adj_edge_index_property_map<unsigned long> > >,
graph_tool::detail::MaskFilter<boost::unchecked_vector_property_map<unsigned
char, boost::typed_identity_property_map<unsigned long> > > > > {lvalue})

class graph_tool.**PropertyMap**(*pmap*, *g*, *key_type*)

This class provides a mapping from vertices, edges or whole graphs to arbitrary properties.

See *Property maps* (page 6) for more details.

The possible property value types are listed below.

Type name	Alias
bool	uint8_t
int16_t	short
int32_t	int
int64_t	long, long long
double	float
long double	
string	
vector<bool>	vector<uint8_t>
vector<int16_t>	short
vector<int32_t>	vector<int>
vector<int64_t>	vector<long>, vector<long long>
vector<double>	vector<float>
vector<long double>	
vector<string>	
python::object	object

copy(*self*, *value_type=None*)

Return a copy of the property map. If value_type is specified, the value type is converted to the chosen type.

get_graph(*self*)

Get the graph class to which the map refers.

key_type(*self*)

Return the key type of the map. Either 'g', 'v' or 'e'.

value_type(*self*)

Return the value type of the map.

python_value_type(*self*)
Return the python-compatible value type of the map.

get_array(*self*)
Get a PropertyArray (page 36) with the property values.

Note: An array is returned *only if* the value type of the property map is a scalar. For vector, string or object types, None is returned instead. For vector and string objects, indirect array access is provided via the get_2d_array() (page **??**) and set_2d_array() (page **??**) member functions.

Warning: The returned array does not own the data, which belongs to the property map. Therefore, if the graph changes, the array may become *invalid* and any operation on it will fail with a ValueError exception. Do **not** store the array if the graph is to be modified; store a **copy** instead.

a
Shortcut to the get_array() (page 36) method as an attribute. This makes assignments more convenient, e.g.:

```
>>> g = gt.Graph()
>>> g.add_vertex(10)
<...>
>>> prop = g.new_vertex_property("double")
>>> prop.a = np.random.random(10)            # Assignment from array
```

fa
The same as the a (page 36) attribute, but instead an *indexed* array is returned, which contains only entries for vertices/edges which are not filtered out. If there are no filters in place, the array is not indexed, and is identical to the a (page 36) attribute.

Note that because advanced indexing is triggered, a **copy** of the array is returned, not a view, as for the a (page 36) attribute. Nevertheless, the assignment of values to the *whole* array at once works as expected.

ma
The same as the a (page 36) attribute, but instead a MaskedArray object is returned, which contains only entries for vertices/edges which are not filtered out. If there are no filters in place, a regular PropertyArray (page 36) is returned, which is identical to the a (page 36) attribute.

get_2d_array(*self*, *pos*)
Return a two-dimensional array with a copy of the entries of the vector-valued property map. The parameter pos must be a sequence of integers which specifies the indexes of the property values which will be used.

set_2d_array(*self*, *a*, *pos=None*)
Set the entries of the vector-valued property map from a two-dimensional array a. If given, the parameter pos must be a sequence of integers which specifies the indexes of the property values which will be set.

is_writable(*self*)
Return True if the property is writable.

class graph_tool.**PropertyArray**
Bases: numpy.ndarray

This is a ndarray subclass which keeps a reference of its PropertyMap (page 35) owner, and detects if the underlying data has been invalidated.

prop_map
PropertyMap (page 35) owner instance.

graph_tool.**load_graph**(*file_name*, *fmt='auto'*, *ignore_vp=None*, *ignore_ep=None*, *ignore_gp=None*)
Load a graph from file_name (which can be either a string or a file-like object).

The format is guessed from file_name, or can be specified by fmt, which can be either "xml", "dot" or "gml".

If provided, the parameters ignore_vp, ignore_ep and ignore_gp, should contain a list of property names (vertex, edge or graph, respectively) which should be ignored when reading the file.

graph_tool.**group_vector_property**(*props*, *value_type=None*, *vprop=None*, *pos=None*)
Group list of properties props into a vector property map of the same type.

Parameters **props** : list of PropertyMap (page 35)

Properties to be grouped.

value_type : string (optional, default: None)

If supplied, defines the value type of the grouped property.

vprop : PropertyMap (page 35) (optional, default: None)

If supplied, the properties are grouped into this property map.

pos : list of ints (optional, default: None)

If supplied, should contain a list of indexes where each corresponding element of props should be inserted.

Returns **vprop** : PropertyMap (page 35)

A vector property map with the grouped values of each property map in props.

Examples

```
>>> from numpy.random import seed, randint
>>> from numpy import array
>>> seed(42)
>>> gt.seed_rng(42)
>>> g = gt.random_graph(100, lambda: (3, 3))
>>> props = [g.new_vertex_property("int") for i in range(3)]
>>> for i in range(3):
...     props[i].a = randint(0, 100, g.num_vertices())
>>> gprop = gt.group_vector_property(props)
>>> print(gprop[g.vertex(0)].a)
[51 25  8]
>>> print(array([p[g.vertex(0)] for p in props]))
[51 25  8]
```

graph_tool.**ungroup_vector_property**(*vprop*, *pos*, *props=None*)
Ungroup vector property map vprop into a list of individual property maps.

Parameters **vprop** : PropertyMap (page 35)

Vector property map to be ungrouped.

pos : list of ints

A list of indexes corresponding to where each element of vprop should be inserted into the ungrouped list.

props : list of `PropertyMap` (page 35) (optional, default: None)

If supplied, should contain a list of property maps to which `vprop` should be ungroupped.

Returns **props** : list of `PropertyMap` (page 35)

A list of property maps with the ungrouped values of `vprop`.

Examples

```
>>> from numpy.random import seed, randint
>>> from numpy import array
>>> seed(42)
>>> gt.seed_rng(42)
>>> g = gt.random_graph(100, lambda: (3, 3))
>>> prop = g.new_vertex_property("vector<int>")
>>> for v in g.vertices():
...     prop[v] = randint(0, 100, 3)
>>> uprops = gt.ungroup_vector_property(prop, [0, 1, 2])
>>> print(prop[g.vertex(0)].a)
[51 92 14]
>>> print(array([p[g.vertex(0)] for p in uprops]))
[51 92 14]
```

graph_tool.**infect_vertex_property**(*g*, *prop*, *vals=None*)

Propagate the *prop* values of vertices with value *val* to all their out-neighbours.

Parameters **prop** : `PropertyMap` (page 35)

Property map to be modified.

vals : list (optional, default: *None*)

List of values to be propagated. If not provided, all values will be propagated.

Returns **None** : None

Examples

```
>>> from numpy.random import seed
>>> seed(42)
>>> gt.seed_rng(42)
>>> g = gt.random_graph(100, lambda: (3, 3))
>>> prop = g.vertex_index.copy("int32_t")
>>> gt.infect_vertex_property(g, prop, [10])
>>> print(sum(prop.a == 10))
4
```

graph_tool.**edge_difference**(*g*, *prop*, *ediff=None*)

Return an edge property map corresponding to the difference between the values of *prop* of target and source vertices of each edge.

Parameters **prop** : `PropertyMap` (page 35)

Vertex property map to be used to compute the difference..

ediff : `PropertyMap` (page 35) (optional, default: *None*)

If provided, the difference values will be stored in this property map.

Returns **ediff** : `PropertyMap` (page 35)

Edge differences.

Examples

```
>>> gt.seed_rng(42)
>>> g = gt.random_graph(100, lambda: (3, 3))
>>> ediff = gt.edge_difference(g, g.vertex_index)
>>> print(ediff.a)
[ 63  74  70 -19   1 -41 -54 -38 -68 -87 -85 -40 -41  -6  -3   4  12 -40
  -1   1 -47 -31 -49 -39  28 -37 -50 -32 -34 -12  -1  -4   5  10   8 -51
 -27  18  -3  45 -13  42   3 -31  25 -21  44 -28 -34  53  -5  -7  47 -26
  67   7  28 -24  30  50  24  39  43  45  64  78  74  84  -7  32  73  47
  34  70   2  -2 -78 -92  81  22  80  37  66  -2   1  26  95  26  62  66
  30   7  56  79  69  80  74  84   8  47  73  54  11  79  71  60  72  57
  41 -15  33 -15 -28  -4 -29 -13  -8 -40  -6   6 -19 -22  15  10  -7 -13
 -29 -10  32  -9 -30 -14 -63 -60  -2 -13 -39  10  12  14 -37 -29 -16 -65
   1 -52 -21 -49 -43 -57  54  31  62 -40 -66 -53 -12 -71 -92 -18 -49 -65
 -83 -80 -33 -67 -70 -58 -40 -53 -44 -71 -46 -75 -37 -44 -57  -3 -15 -76
   4  16 -55 -10   1 -33  16  -6  -7 -66 -49 -57 -58 -35 -32  20 -28 -58
   9  28   7 -67  29   6 -17 -54  -8 -31  24 -37 -29 -19  -5 -13  17 -39
 -25  17  25  62  65 -17  34  -7  12   3  17 -13  -5  40  74  80  36  73
  75  52   4  75  67  43  17  33  57  44  40  34 -26 -15  -5  31  30  51
 -17  21   5 -19  34  -1  12  -1  62 -27  33 -22  43 -22 -39  33 -24  41
 -37  17 -31  45 -40 -39 -36  49  16  36 -19  44  36 -51 -35 -13   4  14
 -44 -16  -8 -13   9 -29  10 -62 -26 -47 -44   3]
```

graph_tool.**value_types**()
Return a list of possible properties value types.

graph_tool.**show_config**()
Show graph_tool build configuration.

3.2 Available subpackages

3.2.1 graph_tool.centrality - Centrality measures

This module includes centrality-related algorithms.

Summary

pagerank (page 39)	Calculate the PageRank of each vertex.
betweenness (page 42)	Calculate the betweenness centrality for each vertex and edge.
central_point_dominance (page 46)	Calculate the central point dominance of the graph, given the betw
closeness (page 43)	Calculate the closeness centrality for each vertex.
eigenvector (page 48)	Calculate the eigenvector centrality of each vertex in the graph, as
katz (page 49)	Calculate the Katz centrality of each vertex in the graph.
hits (page 51)	Calculate the authority and hub centralities of each vertex in the
eigentrust (page 53)	Calculate the eigentrust centrality of each vertex in the graph.
trust_transitivity (page 57)	Calculate the pervasive trust transitivity between chosen (or all) v

Contents

graph_tool.centrality.**pagerank**(*g*, *damping=0.85*, *pers=None*, *weight=None*, *prop=None*, *epsilon=1e-06*, *max_iter=None*, *ret_iter=False*)
Calculate the PageRank of each vertex.

Parameters **g** : Graph (page 28)

Graph to be used.

damping : float, optional (default: 0.85)

Damping factor.

pers : `PropertyMap` (page 35), optional (default: None)

Personalization vector. If omitted, a constant value of $1/N$ will be used.

weight : `PropertyMap` (page 35), optional (default: None)

Edge weights. If omitted, a constant value of 1 will be used.

prop : `PropertyMap` (page 35), optional (default: None)

Vertex property map to store the PageRank values. If supplied, it will be used uninitialized.

epsilon : float, optional (default: 1e-6)

Convergence condition. The iteration will stop if the total delta of all vertices are below this value.

max_iter : int, optional (default: None)

If supplied, this will limit the total number of iterations.

ret_iter : bool, optional (default: False)

If true, the total number of iterations is also returned.

Returns **pagerank** : `PropertyMap` (page 35)

A vertex property map containing the PageRank values.

See Also:

betweenness (page 42) betweenness centrality

eigentrust (page 53) eigentrust centrality

eigenvector (page 48) eigenvector centrality

hits (page 51) hubs and authority centralities

trust_transitivity (page 57) pervasive trust transitivity

Notes

The value of PageRank [pagerank-wikipedia] (page 227) of vertex v, $PR(v)$, is given iteratively by the relation:

$$PR(v) = \frac{1-d}{N} + d \sum_{u \in \Gamma^-(v)} \frac{PR(u)}{d^+(u)}$$

where $\Gamma^-(v)$ are the in-neighbours of v, $d^+(w)$ is the out-degree of w, and d is a damping factor.

If a personalization property $p(v)$ is given, the definition becomes:

$$PR(v) = (1-d)p(v) + d \sum_{u \in \Gamma^-(v)} \frac{PR(u)}{d^+(u)}$$

If edge weights are also given, the equation is then generalized to:

$$PR(v) = (1-d)p(v) + d \sum_{u \in \Gamma^-(v)} \frac{PR(u)w_{u\to v}}{d^+(u)}$$

where $d^+(u) = \sum_y A_{u,y} w_{u\to y}$ is redefined to be the sum of the weights of the out-going edges from u.

The implemented algorithm progressively iterates the above equations, until it no longer changes, according to the parameter epsilon. It has a topology-dependent running time.

If enabled during compilation, this algorithm runs in parallel.

References

[pagerank-wikipedia] (page 227), [lawrence-pagerank-1998] (page 227), [Langville-survey-2005] (page 227), [adamic-polblogs] (page **??**)

Examples

```
>>> g = gt.collection.data["polblogs"]
>>> g = gt.GraphView(g, vfilt=gt.label_largest_component(g))
>>> pr = gt.pagerank(g)
>>> gt.graph_draw(g, pos=g.vp["pos"], vertex_fill_color=pr,
...               vertex_size=gt.prop_to_size(pr, mi=5, ma=15),
...               vorder=pr, output="polblogs_pr.pdf")
<...>
```

Now with a personalization vector, and edge weights:

```
>>> d = g.degree_property_map("total")
>>> periphery = d.a <= 2
>>> p = g.new_vertex_property("double")
>>> p.a[periphery] = 100
>>> pr = gt.pagerank(g, pers=p)
>>> gt.graph_draw(g, pos=g.vp["pos"], vertex_fill_color=pr,
...               vertex_size=gt.prop_to_size(pr, mi=5, ma=15),
...               vorder=pr, output="polblogs_pr_pers.pdf")
<...>
```

graph_tool.centrality.**betweenness**(*g*, *vprop=None*, *eprop=None*, *weight=None*, *norm=True*)

Calculate the betweenness centrality for each vertex and edge.

Parameters **g** : Graph (page 28)

Graph to be used.

vprop : PropertyMap (page 35), optional (default: None)

Vertex property map to store the vertex betweenness values.

eprop : PropertyMap (page 35), optional (default: None)

Edge property map to store the edge betweenness values.

weight : PropertyMap (page 35), optional (default: None)

Edge property map corresponding to the weight value of each edge.

norm : bool, optional (default: True)

Whether or not the betweenness values should be normalized.

Returns **vertex_betweenness** : A vertex property map with the vertex betweenness values.

edge_betweenness : An edge property map with the edge betweenness values.

See Also:

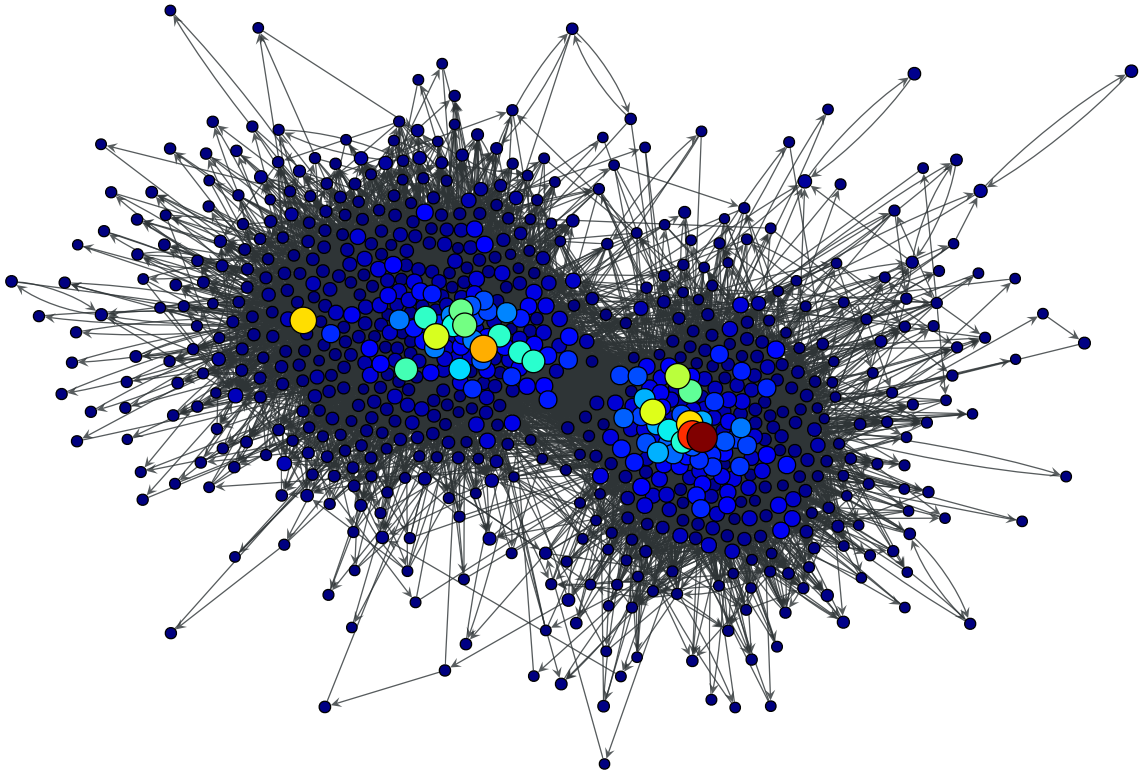


Figure 3.1: PageRank values of the a political blogs network of [adamic-polblogs] (page **??**).

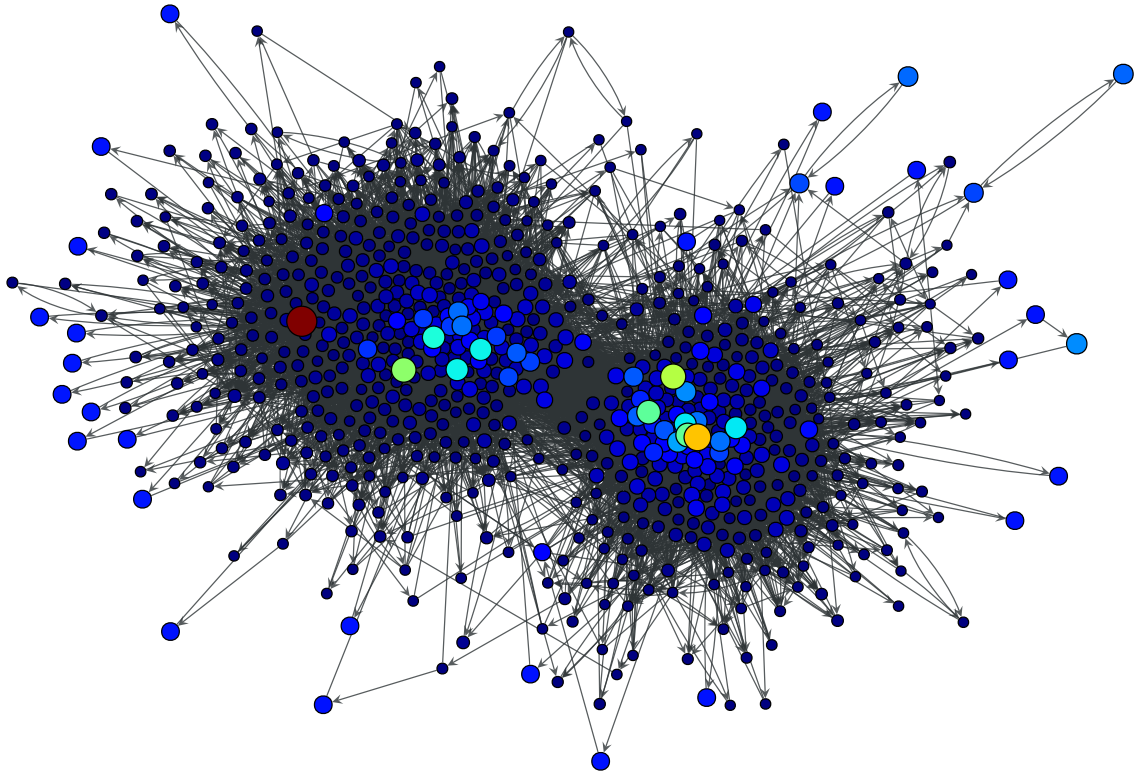


Figure 3.2: Personalized PageRank values of the a political blogs network of [adamic-polblogs] (page **??**), where vertices with very low degree are given artificially high scores.

central_point_dominance (page 46) central point dominance of the graph

pagerank (page 39) PageRank centrality

eigentrust (page 53) eigentrust centrality

eigenvector (page 48) eigenvector centrality

hits (page 51) hubs and authority centralities

trust_transitivity (page 57) pervasive trust transitivity

Notes

Betweenness centrality of a vertex $C_B(v)$ is defined as,

$$C_B(v) = \sum_{\substack{s \neq v \neq t \in V \\ s \neq t}} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where σ_{st} is the number of shortest geodesic paths from s to t, and $\sigma_{st}(v)$ is the number of shortest geodesic paths from s to t that pass through a vertex v. This may be normalised by dividing through the number of pairs of vertices not including v, which is $(n-1)(n-2)/2$.

The algorithm used here is defined in [brandes-faster-2001] (page 227), and has a complexity of $O(VE)$ for unweighted graphs and $O(VE + V(V+E)\log V)$ for weighted graphs. The space complexity is $O(VE)$.

If enabled during compilation, this algorithm runs in parallel.

References

[betweenness-wikipedia] (page 227), [brandes-faster-2001] (page 227), [adamic-polblogs] (page **??**)

Examples

```
>>> g = gt.collection.data["polblogs"]
>>> g = gt.GraphView(g, vfilt=gt.label_largest_component(g))
>>> vp, ep = gt.betweenness(g)
>>> gt.graph_draw(g, pos=g.vp["pos"], vertex_fill_color=vp,
...               vertex_size=gt.prop_to_size(vp, mi=5, ma=15),
...               edge_pen_width=gt.prop_to_size(ep, mi=0.5, ma=5),
...               vorder=vp, output="polblogs_betweenness.pdf")
<...>
```

graph_tool.centrality.**closeness**(*g*, *weight=None*, *source=None*, *vprop=None*, *norm=True*, *harmonic=False*)

Calculate the closeness centrality for each vertex.

Parameters **g** : Graph (page 28)

Graph to be used.

weight : PropertyMap (page 35), optional (default: None)

Edge property map corresponding to the weight value of each edge.

source : Vertex (page 33), optional (default: None)

If specified, the centrality is computed for this vertex alone.

vprop : PropertyMap (page 35), optional (default: None)

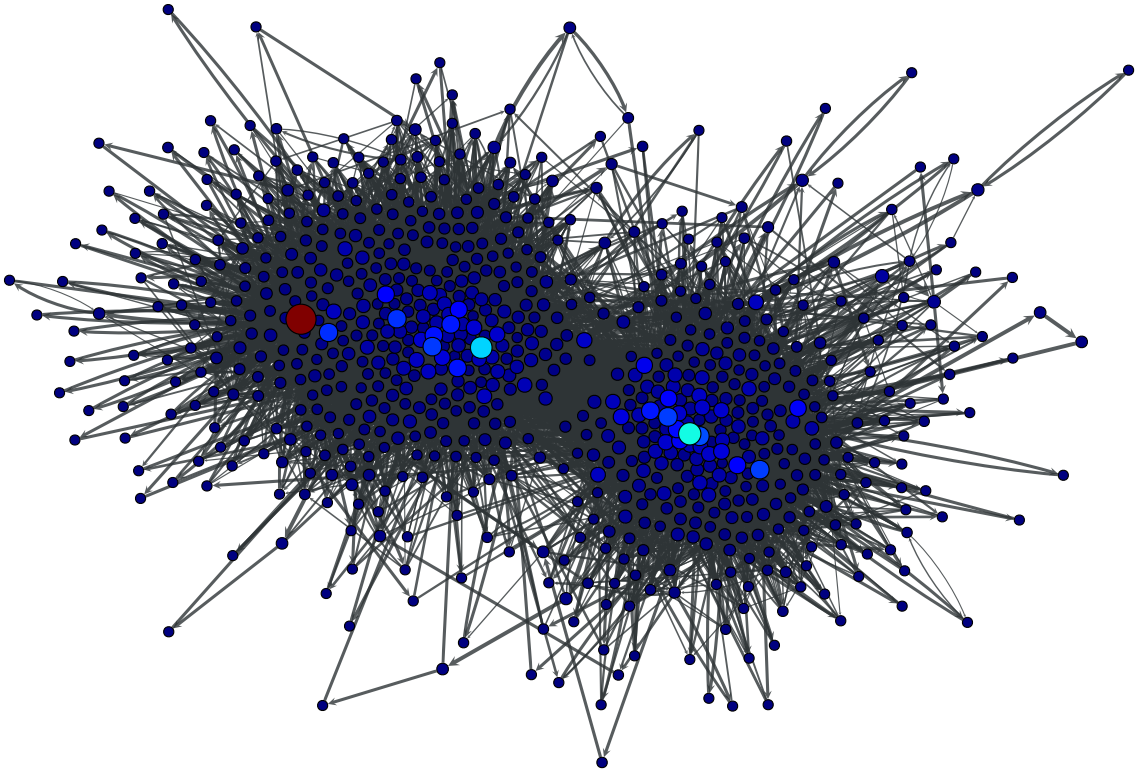


Figure 3.3: Betweenness values of the a political blogs network of [adamic-polblogs] (page **??**).

Vertex property map to store the vertex centrality values.

norm : bool, optional (default: `True`)

Whether or not the centrality values should be normalized.

harmonic : bool, optional (default: `False`)

If true, the sum of the inverse of the distances will be computed, instead of the inverse of the sum.

Returns **vertex_closeness** : `PropertyMap` (page 35)

A vertex property map with the vertex closeness values.

See Also:

`central_point_dominance` (page 46) central point dominance of the graph

`pagerank` (page 39) PageRank centrality

`eigentrust` (page 53) eigentrust centrality

`eigenvector` (page 48) eigenvector centrality

`hits` (page 51) hubs and authority centralities

`trust_transitivity` (page 57) pervasive trust transitivity

Notes

The closeness centrality of a vertex i is defined as,

$$c_i = \frac{1}{\sum_j d_{ij}}$$

where d_{ij} is the (possibly directed and/or weighted) distance from i to j. In case there is no path between the two vertices, here the distance is taken to be zero.

If `harmonic == True`, the definition becomes

$$c_i = \sum_j \frac{1}{d_{ij}},$$

but now, in case there is no path between the two vertices, we take $d_{ij} \to \infty$ such that $1/d_{ij} = 0$.

If `norm == True`, the values of c_i are normalized by $n_i - 1$ where n_i is the size of the (out-) component of i. If `harmonic == True`, they are instead simply normalized by $N - 1$.

The algorithm complexity of $O(N(N + E))$ for unweighted graphs and $O(N(N + E)\log N)$ for weighted graphs. If the option `source` is specified, this drops to $O(N + E)$ and $O((N + E)\log N)$ respectively.

If enabled during compilation, this algorithm runs in parallel.

References

[closeness-wikipedia] (page 227), [opsahl-node-2010] (page 227), [adamic-polblogs] (page **??**)

Examples

```
>>> g = gt.collection.data["polblogs"]
>>> g = gt.GraphView(g, vfilt=gt.label_largest_component(g))
>>> c = gt.closeness(g)
>>> gt.graph_draw(g, pos=g.vp["pos"], vertex_fill_color=c,
...               vertex_size=gt.prop_to_size(c, mi=5, ma=15),
...               vorder=c, output="polblogs_closeness.pdf")
<...>
```

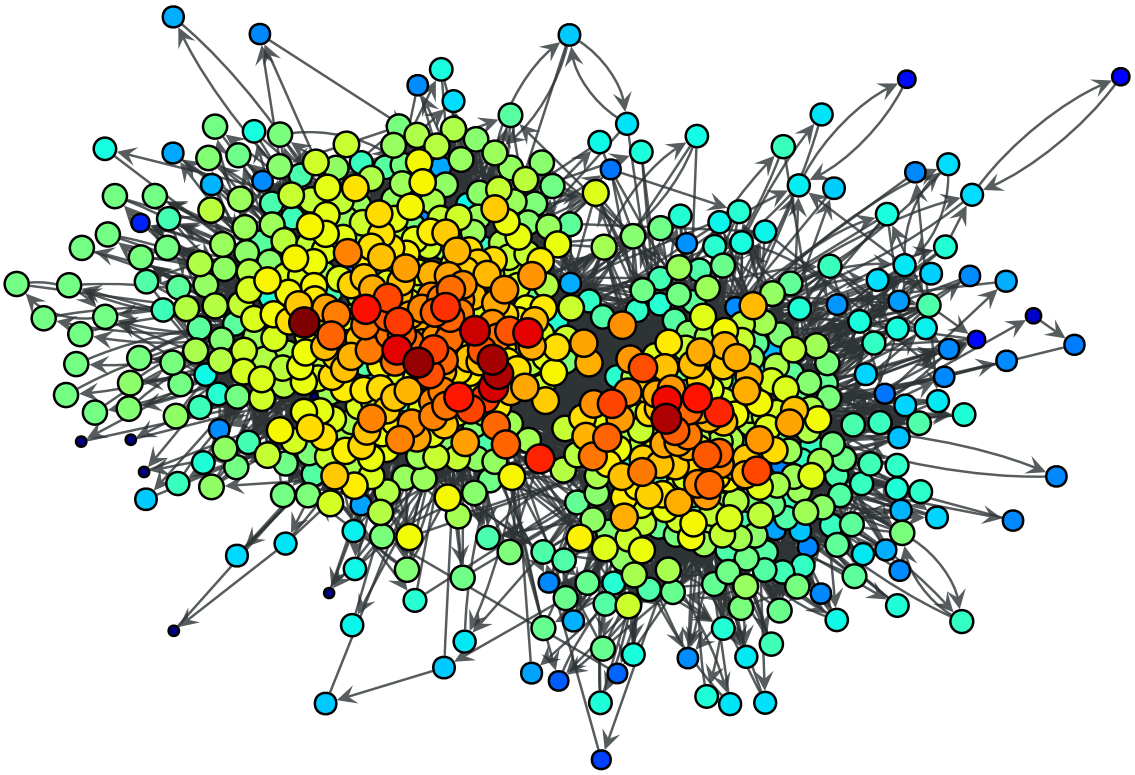


Figure 3.4: Closeness values of the a political blogs network of [adamic-polblogs] (page **??**).

graph_tool.centrality.**central_point_dominance**(*g*, *betweenness*)

Calculate the central point dominance of the graph, given the betweenness centrality of each vertex.

Parameters **g** : Graph (page 28)

Graph to be used.

betweenness : PropertyMap (page 35)

Vertex property map with the betweenness centrality values. The values must be normalized.

Returns **cp** : float

The central point dominance.

See Also:

`betweenness` **(page 42)** betweenness centrality

Notes

Let v^* be the vertex with the largest relative betweenness centrality; then, the central point dominance [freeman-set-1977] (page 227) is defined as:

$$C'_B = \frac{1}{|V| - 1} \sum_v C_B(v^*) - C_B(v)$$

where $C_B(v)$ is the normalized betweenness centrality of vertex v. The value of C_B lies in the range [0,1].

The algorithm has a complexity of $O(V)$.

References

[freeman-set-1977] (page 227)

Examples

```
>>> g = gt.collection.data["polblogs"]
>>> g = gt.GraphView(g, vfilt=gt.label_largest_component(g))
>>> vp, ep = gt.betweenness(g)
>>> print(gt.central_point_dominance(g, vp))
0.11610685614353008
```

graph_tool.centrality.**eigenvector**(*g*, *weight=None*, *vprop=None*, *epsilon=1e-06*, *max_iter=None*)

Calculate the eigenvector centrality of each vertex in the graph, as well as the largest eigenvalue.

Parameters **g** : `Graph` (page 28)

Graph to be used.

weight : `PropertyMap` (page 35) (optional, default: `None`)

Edge property map with the edge weights.

vprop : `PropertyMap` (page 35), optional (default: `None`)

Vertex property map where the values of eigenvector must be stored. If provided, it will be used uninitialized.

epsilon : float, optional (default: `1e-6`)

Convergence condition. The iteration will stop if the total delta of all vertices are below this value.

max_iter : int, optional (default: `None`)

If supplied, this will limit the total number of iterations.

Returns **eigenvalue** : float

The largest eigenvalue of the (weighted) adjacency matrix.

eigenvector : PropertyMap (page 35)

A vertex property map containing the eigenvector values.

See Also:

betweenness (page 42) betweenness centrality

pagerank (page 39) PageRank centrality

hits (page 51) hubs and authority centralities

trust_transitivity (page 57) pervasive trust transitivity

Notes

The eigenvector centrality $\mathbf{x}$ is the eigenvector of the (weighted) adjacency matrix with the largest eigenvalue λ, i.e. it is the solution of

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x},$$

where $\mathbf{A}$ is the (weighted) adjacency matrix and λ is the largest eigenvalue.

The algorithm uses the power method which has a topology-dependent complexity of $O\left(N \times \frac{-\log\epsilon}{\log|\lambda_1/\lambda_2|}\right)$, where N is the number of vertices, ϵ is the `epsilon` parameter, and λ_1 and λ_2 are the largest and second largest eigenvalues of the (weighted) adjacency matrix, respectively.

If enabled during compilation, this algorithm runs in parallel.

References

[eigenvector-centrality] (page 227), [power-method] (page **??**), [langville-survey-2005] (page **??**), [adamic-polblogs] (page **??**)

Examples

```
>>> g = gt.collection.data["polblogs"]
>>> g = gt.GraphView(g, vfilt=gt.label_largest_component(g))
>>> w = g.new_edge_property("double")
>>> w.a = np.random.random(len(w.a)) * 42
>>> ee, x = gt.eigenvector(g, w)
>>> print(ee)
0.0013713102236792602
>>> gt.graph_draw(g, pos=g.vp["pos"], vertex_fill_color=x,
...               vertex_size=gt.prop_to_size(x, mi=5, ma=15),
...               vorder=x, output="polblogs_eigenvector.pdf")
<...>
```

graph_tool.centrality.**katz**(*g*, *alpha=0.01*, *beta=None*, *weight=None*, *vprop=None*, *epsilon=1e-06*, *max_iter=None*)

Calculate the Katz centrality of each vertex in the graph.

Parameters **g** : Graph (page 28)

Graph to be used.

weight : PropertyMap (page 35) (optional, default: None)

Edge property map with the edge weights.

alpha : float, optional (default: 0.01)

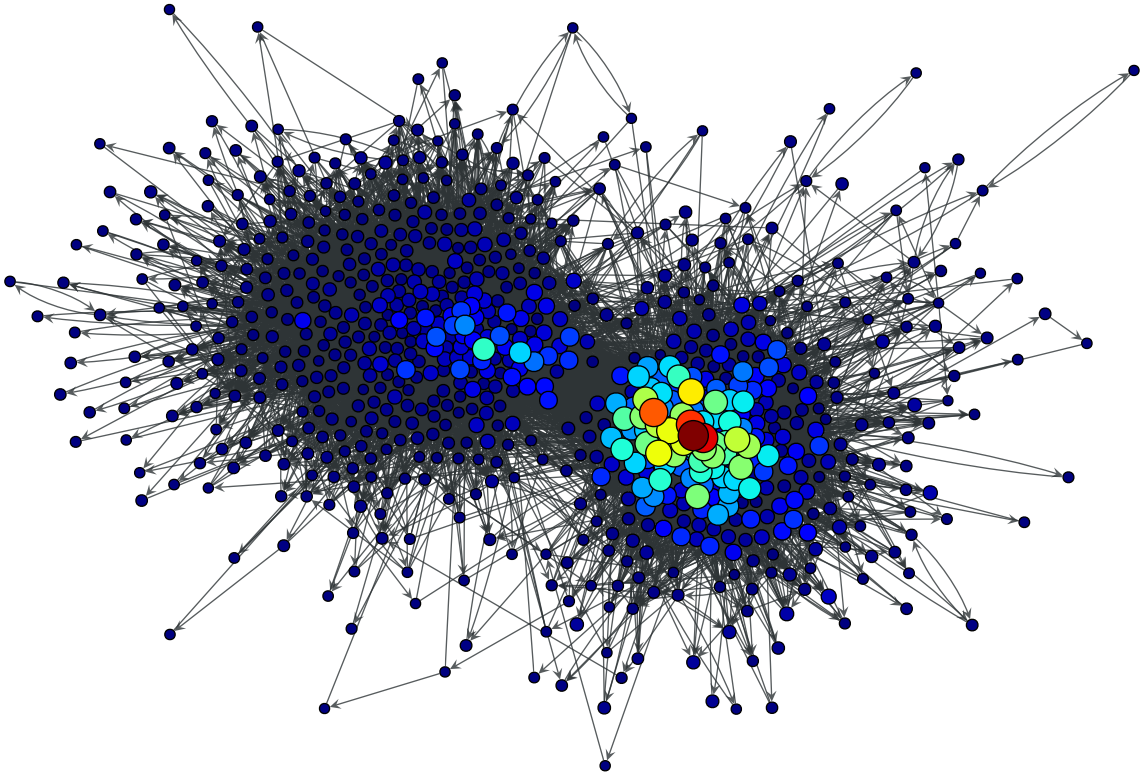


Figure 3.5: Eigenvector values of the a political blogs network of [adamic-polblogs] (page **??**), with random weights attributed to the edges.

Free parameter α. This must be smaller than the inverse of the largest eigenvalue of the adjacency matrix.

beta : PropertyMap (page 35), optional (default: None)

Vertex property map where the local personalization values. If not provided, the global value of 1 will be used.

vprop : PropertyMap (page 35), optional (default: None)

Vertex property map where the values of eigenvector must be stored. If provided, it will be used uninitialized.

epsilon : float, optional (default: 1e-6)

Convergence condition. The iteration will stop if the total delta of all vertices are below this value.

max_iter : int, optional (default: None)

If supplied, this will limit the total number of iterations.

Returns **centrality** : PropertyMap (page 35)

A vertex property map containing the Katz centrality values.

See Also:

betweenness (page 42) betweenness centrality

pagerank (page 39) PageRank centrality

eigenvector (page 48) eigenvector centrality

hits (page 51) hubs and authority centralities

trust_transitivity (page 57) pervasive trust transitivity

Notes

The Katz centrality $\mathbf{x}$ is the solution of the nonhomogeneous linear system

$$\mathbf{x} = \alpha\mathbf{A}\mathbf{x} + \beta,$$

where $\mathbf{A}$ is the (weighted) adjacency matrix and β is the personalization vector (if not supplied, $\beta = \mathbf{1}$ is assumed).

The algorithm uses successive iterations of the equation above, which has a topology-dependent convergence complexity.

If enabled during compilation, this algorithm runs in parallel.

References

[katz-centrality] (page 227), [katz-new] (page 227), [adamic-polblogs] (page **??**)

Examples

```
>>> g = gt.collection.data["polblogs"]
>>> g = gt.GraphView(g, vfilt=gt.label_largest_component(g))
>>> w = g.new_edge_property("double")
>>> w.a = np.random.random(len(w.a)) * 42
>>> x = gt.katz(g, weight=w)
>>> gt.graph_draw(g, pos=g.vp["pos"], vertex_fill_color=x,
```

```
...                 vertex_size=gt.prop_to_size(x, mi=5, ma=15),
...                 vorder=x, output="polblogs_katz.pdf")
<...>
```

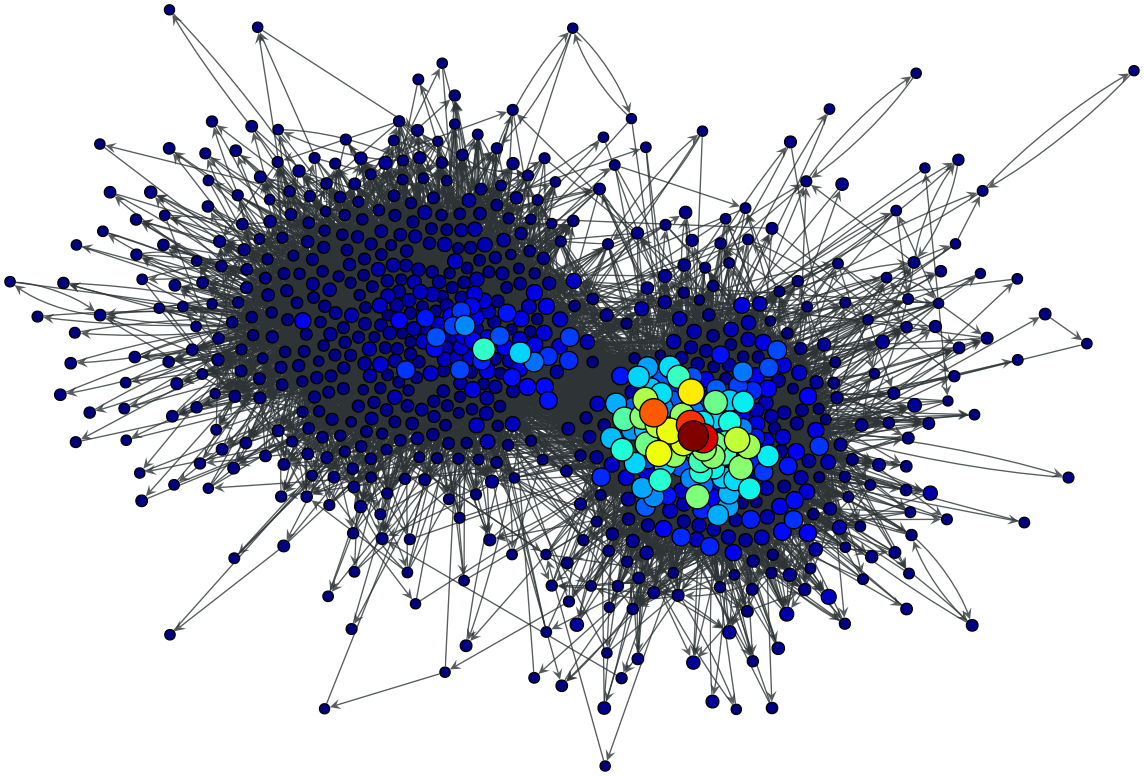


Figure 3.6: Katz centrality values of the a political blogs network of [adamic-polblogs] (page **??**), with random weights attributed to the edges.

graph_tool.centrality.**hits**(*g*, *weight=None*, *xprop=None*, *yprop=None*, *epsilon=1e-06*, *max_iter=None*)

Calculate the authority and hub centralities of each vertex in the graph.

Parameters **g** : Graph (page 28)

Graph to be used.

weight : PropertyMap (page 35) (optional, default: None)

Edge property map with the edge weights.

xprop : PropertyMap (page 35), optional (default: None)

Vertex property map where the authority centrality must be stored.

yprop : PropertyMap (page 35), optional (default: None)

Vertex property map where the hub centrality must be stored.

epsilon : float, optional (default: 1e-6)

Convergence condition. The iteration will stop if the total delta of all vertices are below this value.

max_iter : int, optional (default: None)

If supplied, this will limit the total number of iterations.

Returns **eig** : *float*

The largest eigenvalue of the cocitation matrix.

x : PropertyMap (page 35)

A vertex property map containing the authority centrality values.

y : PropertyMap (page 35)

A vertex property map containing the hub centrality values.

See Also:

betweenness (page 42) betweenness centrality

eigenvector (page 48) eigenvector centrality

pagerank (page 39) PageRank centrality

trust_transitivity (page 57) pervasive trust transitivity

Notes

The Hyperlink-Induced Topic Search (HITS) centrality assigns hub (y) and authority (x) centralities to the vertices, following:

$$\mathbf{x} = \alpha \mathbf{A}\mathbf{y} \tag{3.1}$$

$$\mathbf{y} = \beta \mathbf{A}^T\mathbf{x} \tag{3.2}$$

where $\mathbf{A}$ is the (weighted) adjacency matrix and $\lambda = 1/(\alpha\beta)$ is the largest eigenvalue of the cocitation matrix, $\mathbf{A}\mathbf{A}^T$. (Without loss of generality, we set $\beta = 1$ in the algorithm.)

The algorithm uses the power method which has a topology-dependent complexity of $O\left(N \times \frac{-\log\epsilon}{\log|\lambda_1/\lambda_2|}\right)$, where N is the number of vertices, ϵ is the epsilon parameter, and λ_1 and λ_2 are the largest and second largest eigenvalues of the (weighted) cocitation matrix, respectively.

If enabled during compilation, this algorithm runs in parallel.

References

[hits-algorithm] (page 228), [kleinberg-authoritative] (page 228), [power-method] (page **??**), [adamic-polblogs] (page **??**)

Examples

```
>>> g = gt.collection.data["polblogs"]
>>> g = gt.GraphView(g, vfilt=gt.label_largest_component(g))
>>> ee, x, y = gt.hits(g)
>>> gt.graph_draw(g, pos=g.vp["pos"], vertex_fill_color=x,
...               vertex_size=gt.prop_to_size(x, mi=5, ma=15),
...               vorder=x, output="polblogs_hits_auths.pdf")
<...>
>>> gt.graph_draw(g, pos=g.vp["pos"], vertex_fill_color=y,
...               vertex_size=gt.prop_to_size(y, mi=5, ma=15),
...               vorder=y, output="polblogs_hits_hubs.pdf")
<...>
```

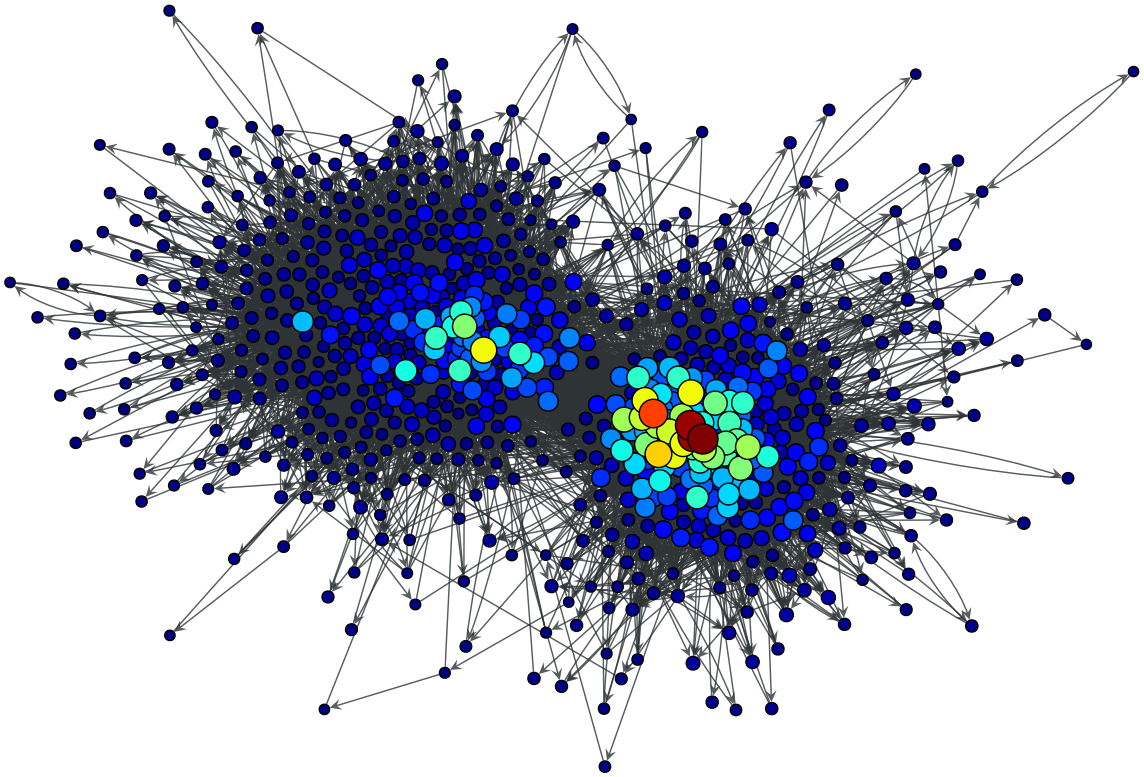


Figure 3.7: HITS authority values of the a political blogs network of [adamic-polblogs] (page **??**).

graph_tool.centrality.**eigentrust**(*g*, *trust_map*, *vprop=None*, *norm=False*, *epsilon=1e-06*, *max_iter=0*, *ret_iter=False*)

Calculate the eigentrust centrality of each vertex in the graph.

Parameters **g** : Graph (page 28)

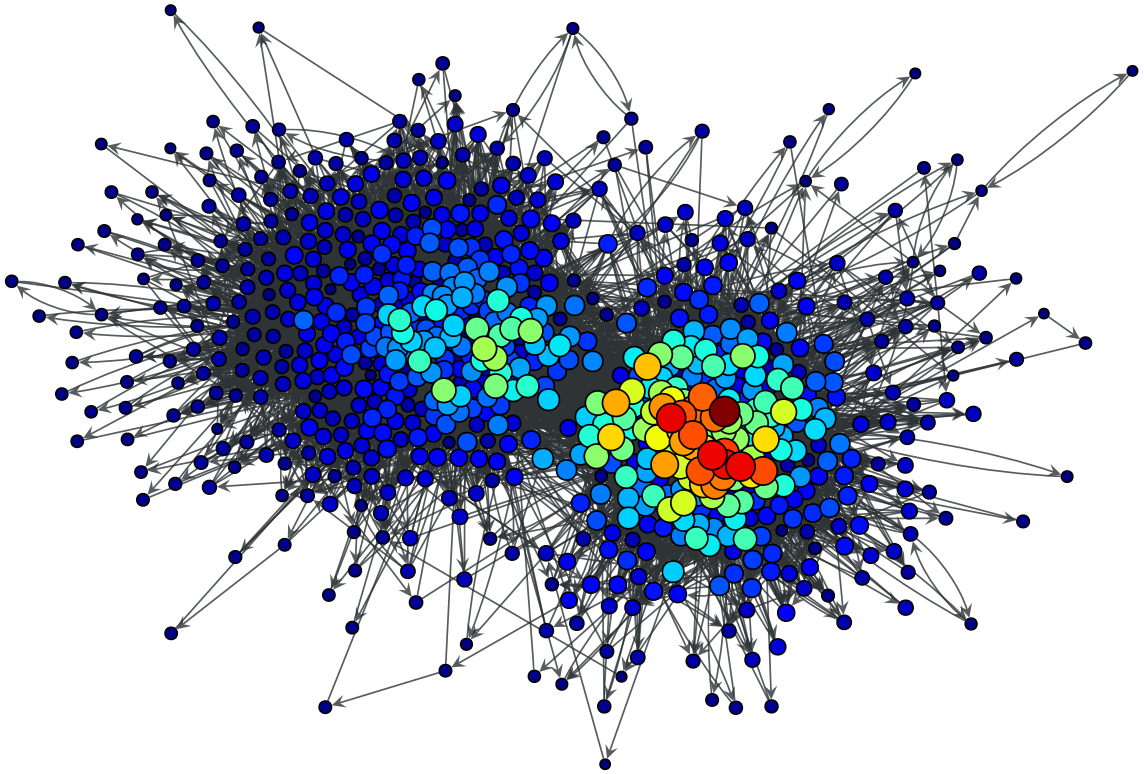


Figure 3.8: HITS hub values of the a political blogs network of [adamic-polblogs] (page **??**).

Graph to be used.

trust_map : PropertyMap (page 35)

Edge property map with the values of trust associated with each edge. The values must lie in the range [0,1].

vprop : PropertyMap (page 35), optional (default: None)

Vertex property map where the values of eigentrust must be stored.

norm : bool, optional (default: False)

Norm eigentrust values so that the total sum equals 1.

epsilon : float, optional (default: 1e-6)

Convergence condition. The iteration will stop if the total delta of all vertices are below this value.

max_iter : int, optional (default: None)

If supplied, this will limit the total number of iterations.

ret_iter : bool, optional (default: False)

If true, the total number of iterations is also returned.

Returns **eigentrust** : PropertyMap (page 35)

A vertex property map containing the eigentrust values.

See Also:

betweenness (page 42) betweenness centrality

pagerank (page 39) PageRank centrality

trust_transitivity (page 57) pervasive trust transitivity

Notes

The eigentrust [kamvar-eigentrust-2003] (page 228) values t_i correspond the following limit

$$\mathbf{t} = \lim_{n\to\infty} \left(C^T\right)^n \mathbf{c}$$

where $c_i = 1/|V|$ and the elements of the matrix C are the normalized trust values:

$$c_{ij} = \frac{\max(s_{ij}, 0)}{\sum_j \max(s_{ij}, 0)}$$

The algorithm has a topology-dependent complexity.

If enabled during compilation, this algorithm runs in parallel.

References

[kamvar-eigentrust-2003] (page 228), [adamic-polblogs] (page **??**)

Examples

```
>>> g = gt.collection.data["polblogs"]
>>> g = gt.GraphView(g, vfilt=gt.label_largest_component(g))
>>> w = g.new_edge_property("double")
>>> w.a = np.random.random(len(w.a)) * 42
>>> t = gt.eigentrust(g, w)
>>> gt.graph_draw(g, pos=g.vp["pos"], vertex_fill_color=t,
...               vertex_size=gt.prop_to_size(t, mi=5, ma=15),
...               vorder=t, output="polblogs_eigentrust.pdf")
<...>
```

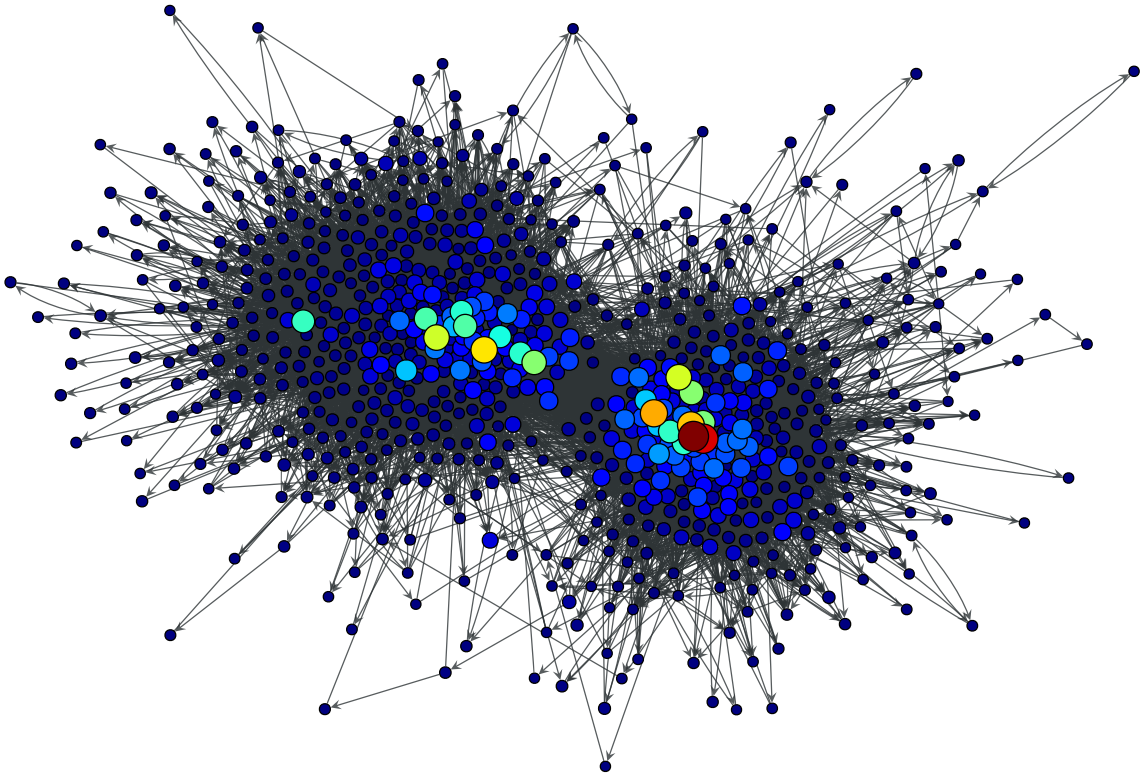


Figure 3.9: Eigentrust values of the a political blogs network of [adamic-polblogs] (page **??**), with random weights attributed to the edges.

graph_tool.centrality.**trust_transitivity**(*g*, *trust_map*, *source=None*, *target=None*, *vprop=None*)

Calculate the pervasive trust transitivity between chosen (or all) vertices in the graph.

Parameters **g** : Graph (page 28)

Graph to be used.

trust_map : PropertyMap (page 35)

Edge property map with the values of trust associated with each edge. The values must lie in the range [0,1].

source : Vertex (page 33) (optional, default: None)

Source vertex. All trust values are computed relative to this vertex. If left unspecified, the trust values for all sources are computed.

target : Vertex (page 33) (optional, default: None)

The only target for which the trust value will be calculated. If left unspecified, the trust values for all targets are computed.

vprop : PropertyMap (page 35) (optional, default: None)

A vertex property map where the values of transitive trust must be stored.

Returns **trust_transitivity** : PropertyMap (page 35) or float

A vertex vector property map containing, for each source vertex, a vector with the trust values for the other vertices. If only one of *source* or *target* is specified, this will be a single-valued vertex property map containing the trust vector from/to the source/target vertex to/from the rest of the network. If both *source* and *target* are specified, the result is a single float, with the corresponding trust value for the target.

See Also:

eigentrust (page 53) eigentrust centrality

betweenness (page 42) betweenness centrality

pagerank (page 39) PageRank centrality

Notes

The pervasive trust transitivity between vertices i and j is defined as

$$t_{ij} = \frac{\sum_m A_{m,j} w^2_{G\setminus\{j\}}(i \to m) c_{m,j}}{\sum_m A_{m,j} w_{G\setminus\{j\}}(i \to m)}$$

where A_{ij} is the adjacency matrix, c_{ij} is the direct trust from i to j, and $w_G(i \to j)$ is the weight of the path with maximum weight from i to j, computed as

$$w_G(i \to j) = \prod_{e \in i \to j} c_e.$$

The algorithm measures the transitive trust by finding the paths with maximum weight, using Dijkstra's algorithm, to all in-neighbours of a given target. This search needs to be performed repeatedly for every target, since it needs to be removed from the graph first. For each given source, the resulting complexity is therefore $O(N^2 \log N)$ for all targets, and $O(N \log N)$ for a single target. For a given target, the complexity for obtaining the trust from all given sources is $O(kN \log N)$, where k is the in-degree of the target. Thus, the complexity for obtaining the complete trust matrix is $O(EN \log N)$, where E is the number of edges in the network.

If enabled during compilation, this algorithm runs in parallel.

References

[richters-trust-2010] (page 228), [adamic-polblogs] (page **??**)

Examples

```
>>> g = gt.collection.data["polblogs"]
>>> g = gt.GraphView(g, vfilt=gt.label_largest_component(g))
>>> g = gt.Graph(g, prune=True)
>>> w = g.new_edge_property("double")
>>> w.a = np.random.random(len(w.a))
>>> g.vp["label"][g.vertex(42)]
'blogforamerica.com'
>>> t = gt.trust_transitivity(g, w, source=g.vertex(42))
>>> gt.graph_draw(g, pos=g.vp["pos"], vertex_fill_color=t,
...               vertex_size=gt.prop_to_size(t, mi=5, ma=15),
...               vorder=t, output="polblogs_trust_transitivity.pdf")
<...>
```

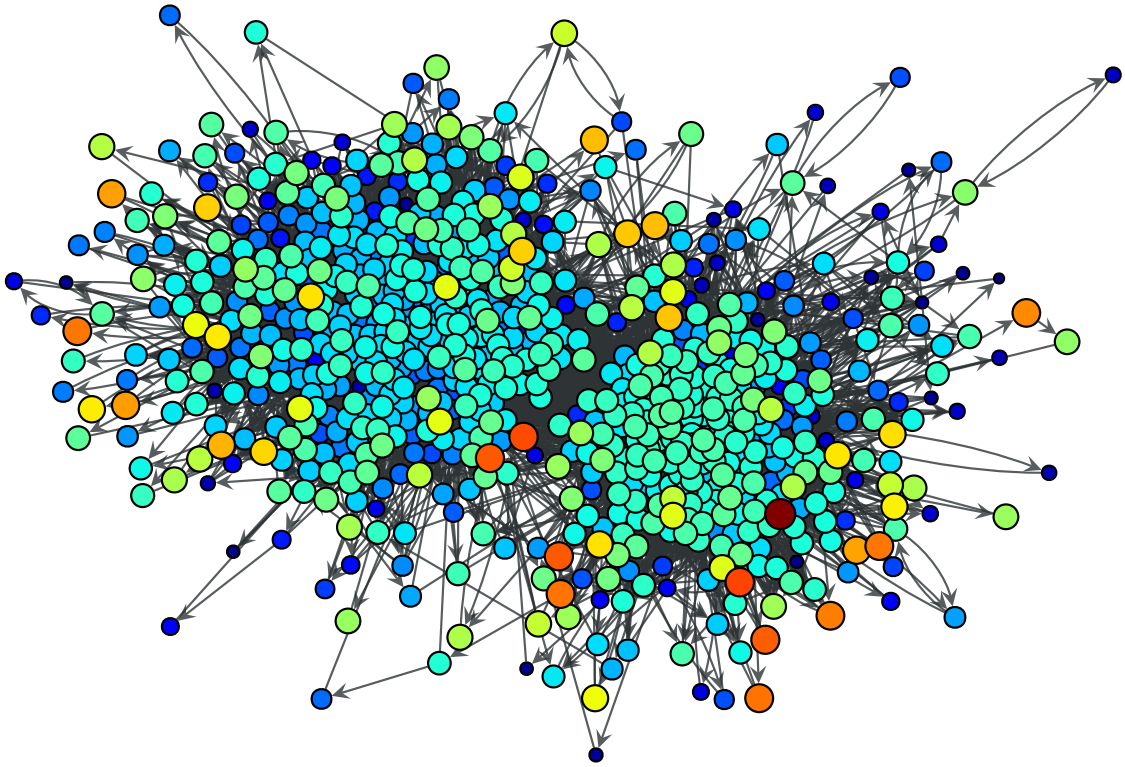


Figure 3.10: Trust transitivity values from source vertex 42 of the a political blogs network of [adamic-polblogs] (page **??**), with random weights attributed to the edges.

3.2.2 graph_tool.clustering - Clustering coefficients

Provides algorithms for calculation of clustering coefficients, aka. transitivity.

Summary

local_clustering (page 60)	Return the local clustering coefficients for all vertices.
global_clustering (page 61)	Return the global clustering coefficient.
extended_clustering (page 61)	Return the extended clustering coefficients for all vertices.
motifs (page 63)	Count the occurrence of k-size subgraphs (motifs).
motif_significance (page 64)	Obtain the motif significance profile, for subgraphs with k vertices.

Contents

graph_tool.clustering.**local_clustering**(*g*, *prop=None*, *undirected=True*)

Return the local clustering coefficients for all vertices.

Parameters **g** : Graph (page 28)

Graph to be used.

prop : PropertyMap (page 35) or string, optional

Vertex property map where results will be stored. If specified, this parameter will also be the return value.

undirected : bool (default: True)

Calculate the *undirected* clustering coefficient, if graph is directed (this option has no effect if the graph is undirected).

Returns **prop** : PropertyMap (page 35)

Vertex property containing the clustering coefficients.

See Also:

global_clustering (page 61) global clustering coefficient

extended_clustering (page 61) extended (generalized) clustering coefficient

motifs (page 63) motif counting

Notes

The local clustering coefficient [watts-collective-1998] (page 228) c_i is defined as

$$c_i = \frac{|\{e_{jk}\}|}{k_i(k_i-1)} : v_j, v_k \in N_i,\, e_{jk} \in E$$

where k_i is the out-degree of vertex i, and

$$N_i = \{v_j : e_{ij} \in E\}$$

is the set of out-neighbours of vertex i. For undirected graphs the value of c_i is normalized as

$$c_i' = 2c_i.$$

The implemented algorithm runs in $O(|V|\langle k\rangle^3)$ time, where $\langle k\rangle$ is the average out-degree.

If enabled during compilation, this algorithm runs in parallel.

References

[watts-collective-1998] (page 228)

Examples

```
>>> g = gt.random_graph(1000, lambda: (5,5))
>>> clust = gt.local_clustering(g)
>>> print(gt.vertex_average(g, clust))
(0.0072, 0.00039746045691969764)
```

graph_tool.clustering.**global_clustering**(*g*)

Return the global clustering coefficient.

Parameters **g** : Graph (page 28)

Graph to be used.

Returns **c** : tuple of floats

Global clustering coefficient and standard deviation (jacknife method)

See Also:

local_clustering (page 60) local clustering coefficient

extended_clustering (page 61) extended (generalized) clustering coefficient

motifs (page 63) motif counting

Notes

The global clustering coefficient [newman-structure-2003] (page 228) c is defined as

$$c = 3 \times \frac{\text{number of triangles}}{\text{number of connected triples}}$$

The implemented algorithm runs in $O(|V|\langle k\rangle^3)$ time, where $\langle k\rangle$ is the average (total) degree.

If enabled during compilation, this algorithm runs in parallel.

References

[newman-structure-2003] (page 228)

Examples

```
>>> g = gt.random_graph(1000, lambda: (5,5))
>>> print(gt.global_clustering(g))
(0.007182172103638073, 0.0003970213508956326)
```

graph_tool.clustering.**extended_clustering**(*g*, *props=None*, *max_depth=3*, *undirected=False*)

Return the extended clustering coefficients for all vertices.

Parameters **g** : Graph (page 28)

Graph to be used.

props : list of PropertyMap (page 35) objects, optional

list of vertex property maps where results will be stored. If specified, this parameter will also be the return value.

max_depth : int, optional

Maximum clustering order (default: 3).

undirected : bool, optional

Calculate the *undirected* clustering coefficients, if graph is directed (this option has no effect if the graph is undirected).

Returns **prop** : list of PropertyMap (page 35) objects

List of vertex properties containing the clustering coefficients.

See Also:

local_clustering (page 60) local clustering coefficient

global_clustering (page 61) global clustering coefficient

motifs (page 63) motif counting

Notes

The extended clustering coefficient c_i^d of order d is defined as

$$c_i^d = \frac{|\{\,\{u,v\}; u,v \in N_i | d_{G(V\setminus\{i\})}(u,v) = d\,\}|}{\binom{|N_i|}{2}},$$

where $d_G(u,v)$ is the shortest distance from vertex u to v in graph G, and

$$N_i = \{v_j : e_{ij} \in E\}$$

is the set of out-neighbours of i. According to the above definition, we have that the traditional local clustering coefficient is recovered for $d = 1$, i.e., $c_i^1 = c_i$.

The implemented algorithm runs in $O(|V|\,\langle k\rangle^{2+\text{max-depth}})$ worst time, where $\langle k\rangle$ is the average out-degree.

If enabled during compilation, this algorithm runs in parallel.

References

[abdo-clustering] (page 228)

Examples

```
>>> g = gt.random_graph(1000, lambda: (5,5))
>>> clusts = gt.extended_clustering(g, max_depth=5)
>>> for i in range(0, 5):
...     print(gt.vertex_average(g, clusts[i]))
...
(0.005535, 0.00046266726404860573)
(0.02351, 0.0009395843702876762)
(0.11651833333333333, 0.0019837472110181336)
(0.391765, 0.0030221072114043567)
(0.4439983333333333, 0.0030393922085216675)
```

graph_tool.clustering.**motifs**(*g*, *k*, *p=1.0*, *motif_list=None*)

Count the occurrence of k-size subgraphs (motifs). A tuple with two lists is returned: the list of motifs found, and the list with their respective counts.

Parameters **g** : Graph (page 28)

Graph to be used.

k : int

number of vertices of the motifs

p : float or float list (optional, default: 1.0)

uniform fraction of the motifs to be sampled. If a float list is provided, it will be used as the fraction at each depth $[1, \ldots, k]$ in the algorithm. See [wernicke-efficient-2006] (page 228) for more details.

motif_list : list of Graph (page 28) objects, optional

If supplied, the algorithms will only search for the motifs in this list (or isomorphisms).

Returns **motifs** : list of Graph (page 28) objects

List of motifs of size k found in the Graph. Graphs are grouped according to their isomorphism class, and only one of each class appears in this list. The list is sorted according to in-degree sequence, out-degree-sequence, and number of edges (in this order).

counts : list of ints

The number of times the respective motif in the motifs list was counted

See Also:

motif_significance (page 64) significance profile of motifs

local_clustering (page 60) local clustering coefficient

global_clustering (page 61) global clustering coefficient

extended_clustering (page 61) extended (generalized) clustering coefficient

Notes

This functions implements the ESU and RAND-ESU algorithms described in [wernicke-efficient-2006] (page 228).

If enabled during compilation, this algorithm runs in parallel.

References

[wernicke-efficient-2006] (page 228)

Examples

```
>>> g = gt.random_graph(1000, lambda: (5,5))
>>> motifs, counts = gt.motifs(gt.GraphView(g, directed=False), 4)
>>> print(len(motifs))
63
>>> print(counts)
[28448, 28343, 28929, 29600, 159144, 98111, 98187, 33446, 284, 176, 70, 139, 210, 37, 526, 28
```

graph_tool.clustering.**motif_significance**(*g*, *k*, *n_shuffles=100*, *p=1.0*, *motif_list=None*, *threshold=0*, *self_loops=False*, *parallel_edges=False*, *full_output=False*, *shuffle_model='uncorrelated'*)

Obtain the motif significance profile, for subgraphs with k vertices. A tuple with two lists is returned: the list of motifs found, and their respective z-scores.

Parameters **g** : Graph (page 28)

Graph to be used.

k : int

Number of vertices of the motifs

n_shuffles : int (optional, default: 100)

Number of shuffled networks to consider for the z-score

p : float or float list (optional, default: 1.0)

Uniform fraction of the motifs to be sampled. If a float list is provided, it will be used as the fraction at each depth $[1, \ldots, k]$ in the algorithm. See [wernicke-efficient-2006] (page 228) for more details.

motif_list : list of Graph (page 28) objects (optional, default: None)

If supplied, the algorithms will only search for the motifs in this list (isomorphisms)

threshold : int (optional, default: 0)

If a given motif count is below this level, it is not considered.

self_loops : bool (optional, default: False)

Whether or not the shuffled graphs are allowed to contain self-loops

parallel_edges : bool (optional, default: False)

Whether or not the shuffled graphs are allowed to contain parallel edges.

full_output : bool (optional, default: False)

If set to True, three additional lists are returned: the count of each motif, the average count of each motif in the shuffled networks, and the standard deviation of the average count of each motif in the shuffled networks.

shuffle_model : string (optional, default: "uncorrelated")

Shuffle model to use. See random_rewire() (page 142) for details.

Returns **motifs** : list of `Graph` (page 28) objects

List of motifs of size k found in the Graph. Graphs are grouped according to their isomorphism class, and only one of each class appears in this list. The list is sorted according to in-degree sequence, out-degree-sequence, and number of edges (in this order).

z-scores : list of floats

The z-score of the respective motives. See below for the definition of the z-score.

See Also:

`motifs` (page 63) motif counting or sampling

`local_clustering` (page 60) local clustering coefficient

`global_clustering` (page 61) global clustering coefficient

`extended_clustering` (page 61) extended (generalized) clustering coefficient

Notes

The z-score z_i of motif i is defined as

$$z_i = \frac{N_i - \langle N_i^s \rangle}{\sqrt{\langle (N_i^s)^2 \rangle - \langle N_i^s \rangle^2}},$$

where N_i is the number of times motif i found, and N_i^s is the count of the same motif but on a shuffled network. It measures how many standard deviations is each motif count, in respect to an ensemble of randomly shuffled graphs with the same degree sequence.

The z-scores values are not normalized.

If enabled during compilation, this algorithm runs in parallel.

Examples

```
>>> from numpy import random
>>> random.seed(10)
>>> g = gt.random_graph(100, lambda: (3,3))
>>> motifs, zscores = gt.motif_significance(g, 3)
>>> print(len(motifs))
11
>>> print(zscores)
[1.0170744045563587, 0.90350945788797254, 0.93486676613275743, -0.84952469609377301, -0.95933
```

3.2.3 `graph_tool.collection` - Dataset collection

This module contains an assortment of useful networks.

`graph_tool.collection.`**`data`**

Dictionary containing `Graph` (page 28) objects, indexed by the name of the graph. This is a "lazy" dictionary, i.e. it only loads the graphs from disk when the items are accessed for the first time. The description for each graph is given in the `descriptions` (page 66) dictionary, or alternatively in the `"description"` graph property which accompanies each graph object.

```
>>> g = gt.collection.data["karate"]
>>> print(g)
<Graph object, undirected, with 34 vertices and 78 edges at 0x9ca5d90>
```

graph_tool.collection.**descriptions**

Dictionary with a short description and source information on each graph.

A summary, with some extra information, is available in the following table.

Name	N	E	Directed	Description
adjnoun	112	425	False	Word adjacencies: adjacency network of common adjectives and nouns in the novel David Copperfield by Charles Dickens. Please cite M. E. J. Newman, Phys. Rev. E 74, 036104 (2006). Retrieved from Mark Newman's website.
as22july06	22963	48436	False	Internet: a symmetrized snapshot of the structure of the Internet at the level of autonomous systems, reconstructed from BGP tables posted by the University of Oregon Route Views Project. This snapshot was created by Mark Newman from data for July 22, 2006 and is not previously published. Retrieved from Mark Newman's website.
astroph	16706	121251	False	Astrophysics collaborations: weighted network of coauthorships between scientists posting preprints on the Astrophysics E-Print Archive between Jan 1, 1995 and December 31, 1999. Please cite M. E. J. Newman, Proc. Natl. Acad. Sci. USA 98, 404-409 (2001). Retrieved from Mark Newman's website.
celegansneural	297	2359	True	Neural network: A directed, weighted network representing the neural network of C. Elegans. Data compiled by D. Watts and S. Strogatz and made available on the web here. Please cite D. J. Watts and S. H. Strogatz, Nature 393, 440-442 (1998). Original experimental data taken from J. G. White, E. Southgate, J. N. Thompson, and S. Brenner, Phil. Trans. R. Soc. London 314, 1-340 (1986). Retrieved from Mark Newman's website.
condmat	16726	47594	False	Condensed matter collaborations 1999: weighted network of coauthorships between scientists posting preprints on the Condensed Matter E-Print Archive between Jan 1, 1995 and December 31, 1999. Please cite M. E. J. Newman, The structure of scientific collaboration networks, Proc. Natl. Acad. Sci. USA 98, 404-409 (2001). Retrieved from Mark Newman's website.
condmat2003	31163	120029	False	Condensed matter collaborations 2003: updated network of coauthorships between scientists posting preprints on the Condensed Matter E-Print Archive. This version includes all preprints posted between Jan 1, 1995 and June 30, 2003. The largest component of this network, which contains 27519 scientists, has been used by several authors as a test-bed for community-finding algorithms for large networks; see for example J. Duch and A. Arenas, Phys. Rev. E 72, 027104 (2005). These data can be cited as M. E. J. Newman, Proc. Natl. Acad. Sci. USA 98, 404-409 (2001). Retrieved from Mark Newman's website.
condmat2005	40421	175693	False	Condensed matter collaborations 2005: updated network of coauthorships between scientists posting preprints on the Condensed Matter E-Print Archive. This version includes all preprints posted between Jan 1, 1995 and March 31, 2005. Please cite M. E. J. Newman, Proc. Natl. Acad. Sci. USA 98, 404-409 (2001). Retrieved from Mark Newman's website.
dolphins	62	159	False	Dolphin social network: an undirected social network of frequent associations between 62 dolphins in a community living off Doubtful Sound, New Zealand. Please cite D. Lusseau, K. Schneider, O. J. Boisseau, P. Haase, E. Slooten, and S. M. Dawson, Behavioral Ecology and Sociobiology 54, 396-405 (2003). Retrieved from Mark Newman's website.
email-Enron	36692	367662	False	Enron email communication network covers all the email communication within a dataset of around half million

Contents

graph_tool.collection.**get_data_path**(*name*)
Return the full path of the corresponding dataset.

3.2.4 graph_tool.community - Community structure

This module contains algorithms for the computation of community structure on graphs.

Stochastic blockmodel inference

Summary

minimize_blockmodel_dl (page 68)	Find the block partition of an unspecified size which minimizes t
BlockState (page 70)	This class encapsulates the block state of a given graph.
mcmc_sweep (page 74)	Performs a Monte Carlo Markov chain sweep on the network, to
collect_vertex_marginals (page 78)	Collect the vertex marginal histogram, which counts the number
collect_edge_marginals (page 76)	Collect the edge marginal histogram, which counts the number o
mf_entropy (page 80)	Compute the "mean field" entropy given the vertex block member
bethe_entropy (page 78)	Compute the Bethe entropy given the edge block membership ma
model_entropy (page 81)	Computes the amount of information necessary for the paramete
get_max_B (page 81)	Return the maximum detectable number of blocks, obtained by
get_akc (page 82)	Return the minimum value of the average degree of the network,
min_dist (page 82)	Return the minimum distance between all blocks, and the block
condensation_graph (page 82)	Obtain the condensation graph, where each vertex with the same

Modularity-based community detection

Summary

community_structure (page 84)	Obtain the community structure for the given graph, using a Potts mod
modularity (page 92)	Calculate Newman's modularity.

Contents

graph_tool.community.**minimize_blockmodel_dl**(*g*, *deg_corr=True*, *nsweeps=100*, *adaptive_convergence=True*, *anneal=1.0*, *greedy_cooling=True*, *max_B=None*, *min_B=1*, *mid_B=None*, *b_cache=None*, *b_start=None*, *clabel=None*, *checkpoint=None*, *verbose=False*)
Find the block partition of an unspecified size which minimizes the description length of the network, according to the stochastic blockmodel ensemble which best describes it.

Parameters **g** : Graph (page 28)

Graph being used.

deg_corr : bool (optional, default: True)

If True, the degree-corrected version of the blockmodel ensemble will be assumed, otherwise the traditional variant will be used.

nsweeps : int (optional, default: *50*)

Number of sweeps per value of B tried. If *adaptive_convergence* == *True*, this corresponds to the number of sweeps observed to determine convergence, not the total number of sweeps performed, which can be much larger.

adaptive_convergence : `bool` (optional, default: `True`)

If `True`, the parameter `nsweeps` represents not the total number of sweeps performed per value of `B`, but instead the number of sweeps without an improvement on the value of $S_{c/t}$ so that convergence is assumed.

anneal : `float` (optional, default: `1.`)

Annealing factor which multiplies the inverse temperature β after each convergence. If `anneal <= 1.`, no annealing is performed.

greedy_cooling : `bool` (optional, default: `True`)

If `True`, a final abrupt cooling step is performed after the Markov chain has equilibrated.

max_B : `int` (optional, default: `None`)

Maximum number of blocks tried. If not supplied, it will be automatically determined.

min_B : `int` (optional, default: *1*)

Minimum number of blocks tried.

mid_B : `int` (optional, default: `None`)

Middle of the range which brackets the minimum. If not supplied, will be automatically determined.

b_cache : `dict` with `int` keys and (`float`, `PropertyMap` (page 35)) values (optional, default: `None`)

If provided, this corresponds to a dictionary where the keys are the number of blocks, and the values are tuples containing two values: the description length and its associated vertex partition. Values present in this dictionary will not be computed, and will be used unmodified as the solution for the corresponding number of blocks. This can be used to continue from a previously unfinished run.

b_start : `dict` with `int` keys and (`float`, `PropertyMap` (page 35)) values (optional, default: `None`)

Like *b_cache*, but the partitions present in the dictionary will be used as the starting point of the minimization.

clabel : `PropertyMap` (page 35) (optional, default: `None`)

Constraint labels on the vertices, such that vertices with different labels cannot belong to the same block.

checkpoint : function (optional, default: `None`)

If provided, this function will be called after each call to `mcmc_sweep()` (page 74). This can be used to store the current state, so it can be continued later. The function must have the following signature:

```
def checkpoint(state, L, delta, nmoves):
    ...
```

where *state* is either a BlockState (page 70) instance or None, L is the current description length, *delta* is the entropy difference in the last MCMC sweep, and *nmoves* is the number of accepted block membership moves.

This function will also be called when the MCMC has finished for the current value of B, in which case `state == None`, and the remaining parameters will be zero.

verbose : bool (optional, default: False)

If True, verbose information is displayed.

Returns **b** : PropertyMap (page 35)

Vertex property map with the best block partition.

min_dl : *float*

Minimum value of the description length (in nats).

b_cache : dict with int keys and (float, PropertyMap (page 35)) values

Dictionary where the keys are the number of blocks visited during the algorithm, and the values are tuples containing two values: the description length and its associated vertex partition.

Notes

This algorithm attempts to find a block partition of an unspecified size which minimizes the description length of the network,

$$\Sigma_{t/c} = \mathcal{S}_{t/c} + \mathcal{L}_{t/c},$$

where $\mathcal{S}_{t/c}$ is the blockmodel entropy (as described in the docstring of mcmc_sweep() (page 74) and BlockState.entropy() (page 72)) and $\mathcal{L}_{t/c}$ is the information necessary to describe the model (as described in the docstring of model_entropy() (page 81) and BlockState.entropy() (page 72)).

The algorithm works by minimizing the entropy $\mathcal{S}_{t/c}$ for specific values of B via mcmc_sweep() (page 74) (with $\beta = 1$ and $\beta \to \infty$), and minimizing $\Sigma_{t/c}$ via an one-dimensional Fibonacci search on B. See [peixoto-parsimonious-2013] (page **??**) for more details.

This algorithm has a complexity of $O(\tau E \ln B_{\max})$, where E is the number of edges in the network, τ is the mixing time of the MCMC, and $B_{\max}$ is the maximum number of blocks considered. If $B_{\max}$ is not supplied, it is computed as $\sim \sqrt{E}$ via get_max_B() (page 81), in which case the complexity becomes $O(\tau E \ln E)$.

References

[holland-stochastic-1983] (page **??**), [faust-blockmodels-1992] (page **??**), [karrer-stochastic-2011] (page **??**), [peixoto-entropy-2012] (page **??**), [peixoto-parsimonious-2013] (page **??**)

Examples

```
>>> g = gt.collection.data["polbooks"]
>>> b, mdl, b_cache = gt.minimize_blockmodel_dl(g)
>>> gt.graph_draw(g, pos=g.vp["pos"], vertex_fill_color=b, vertex_shape=b, output="polbooks_b
<...>
```

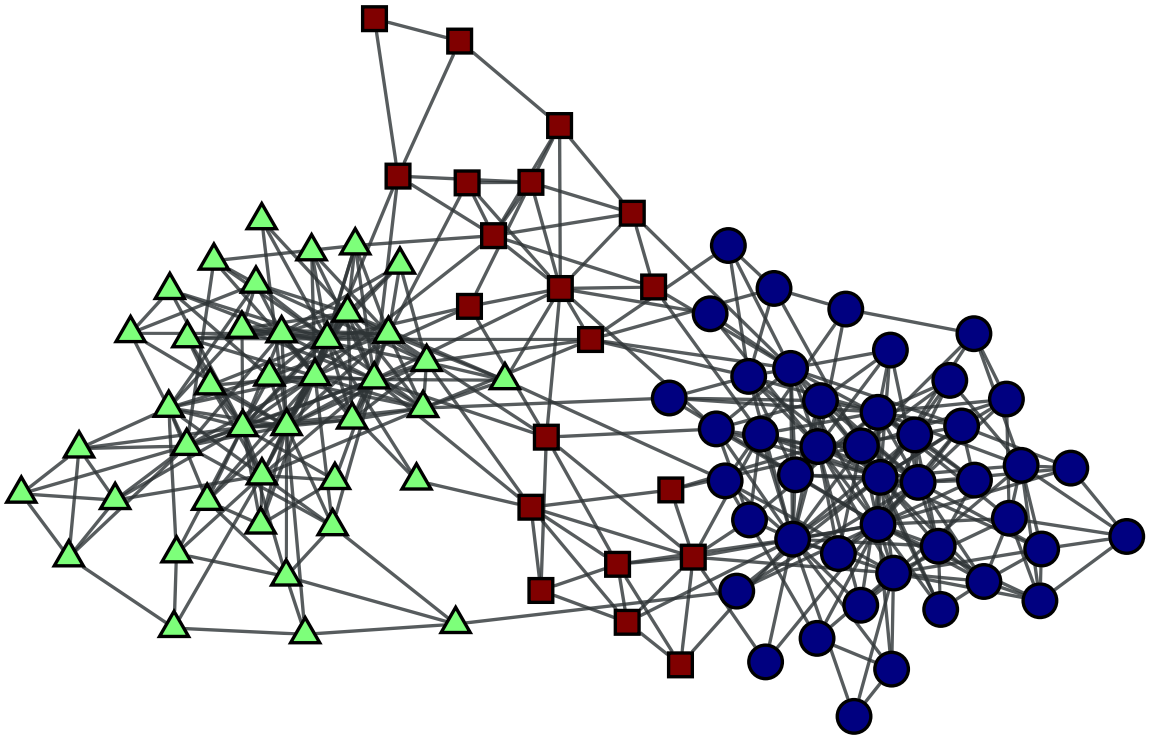


Figure 3.11: Block partition of a political books network, which minimizes the description lenght of the network according to the degree-corrected stochastic blockmodel.

class graph_tool.community.**BlockState**(*g*, *eweight=None*, *vweight=None*, *b=None*, *clabel=None*, *B=None*, *deg_corr=True*)

This class encapsulates the block state of a given graph.

This must be instantiated and used by functions such as mcmc_sweep() (page 74).

Parameters **g** : Graph (page 28)

Graph to be used.

eweight : PropertyMap (page 35) (optional, default: None)

Edge weights (i.e. multiplicity).

vweight : PropertyMap (page 35) (optional, default: None)

Vertex weights (i.e. multiplicity).

b : PropertyMap (page 35) (optional, default: None)

Initial block labels on the vertices. If not supplied, it will be randomly sampled.

B : int (optional, default: None)

Number of blocks. If not supplied it will be either obtained from the parameter b, or set to the maximum possible value according to the minimum description length.

clabel : PropertyMap (page 35) (optional, default: None)

This parameter provides a constraint label, such that vertices with different labels will not be allowed to belong to the same block. If not given, all labels will be assumed to be the same.

deg_corr : bool (optional, default: True)

If True, the degree-corrected version of the blockmodel ensemble will be assumed, otherwise the traditional variant will be used.

add_vertex(*self*, *v*, *r*)

Add vertex *v* to block *r*.

dist(*self*, *r*, *s*)

Compute the "distance" between blocks *r* and *s*, i.e. the entropy difference after they are merged together.

entropy(*self*, *complete=False*, *random=False*, *dl=False*)

Calculate the entropy per edge associated with the current block partition.

Parameters **complete** : bool (optional, default: False)

If True, the complete entropy will be returned, including constant terms not relevant to the block partition.

random : bool (optional, default: False)

If True, the entropy entropy corresponding to an equivalent random graph (i.e. no block partition) will be returned.

dl : bool (optional, default: False)

If True, the full description length will be returned.

Notes

For the traditional blockmodel (`deg_corr == False`), the entropy is given by

$$\mathcal{S}_t \cong E - \frac{1}{2}\sum_{rs} e_{rs} \ln\left(\frac{e_{rs}}{n_r n_s}\right),$$
$$\mathcal{S}_t^d \cong E - \sum_{rs} e_{rs} \ln\left(\frac{e_{rs}}{n_r n_s}\right),$$

for undirected and directed graphs, respectively, where e_{rs} is the number of edges from block r to s (or the number of half-edges for the undirected case when $r = s$), and n_r is the number of vertices in block r .

For the degree-corrected variant with "hard" degree constraints the equivalent expressions are

$$\mathcal{S}_c \cong -E - \sum_k N_k \ln k! - \frac{1}{2}\sum_{rs} e_{rs} \ln\left(\frac{e_{rs}}{e_r e_s}\right),$$
$$\mathcal{S}_c^d \cong -E - \sum_{k^+} N_{k^+} \ln k^+! - \sum_{k^-} N_{k^-} \ln k^-! - \sum_{rs} e_{rs} \ln\left(\frac{e_{rs}}{e_r^+ e_s^-}\right),$$

where $e_r = \sum_s e_{rs}$ is the number of half-edges incident on block r, and $e_r^+ = \sum_s e_{rs}$ and $e_r^- = \sum_s e_{sr}$ are the number of out- and in-edges adjacent to block r, respectively.

If `complete == False` only the last term of the equations above will be returned. If `random == True` it will be assumed that $B = 1$ despite the actual e_{rs} matrix. If `dl == True`, the description length $\mathcal{L}_t$ of the model will be returned as well, as described in `model_entropy()` (page 81). Note that for the degree-corrected version the description length is

$$\mathcal{L}_c = \mathcal{L}_t - N\sum_k p_k \ln p_k,$$

where p_k is the fraction of nodes with degree p_k, and we have instead $k \to (k^-, k^+)$ for directed graphs.

Note that the value returned corresponds to the entropy *per edge*, i.e. $(\mathcal{S}_{t/c}\,[+\mathcal{L}_{t/c}])/E$.

get_bg(*self*)
Returns the block graph.

get_blocks(*self*)
Returns the property map which contains the block labels for each vertex.

get_clabel(*self*)
Obtain the constraint label associated with each block.

get_dist_matrix(*self*)
Return the distance matrix between all blocks. The distance is defined as the entropy difference after two blocks are merged.

get_er(*self*)
Returns the vertex property map of the block graph which contains the number e_r of half-edges incident on block r. If the graph is directed, a pair of property maps is returned, with the number of out-edges e_r^+ and in-edges e_r^-, respectively.

get_ers(*self*)
Returns the edge property map of the block graph which contains the e_{rs} matrix entries.

get_eweight(*self*)
Returns the block edge counts associated with the block matrix e_{rs}. For directed graphs it is identical to e_{rs}, but for undirected graphs it is identical except for the diagonal, which is $e_{rr}/2$.

get_matrix(*self*, *reorder=False*, *clabel=None*, *niter=0*, *ret_order=False*)
Returns the block matrix.

Parameters **reorder** : `bool` (optional, default: `False`)

If `True`, the matrix is reordered so that blocks which are 'similar' are close together.

clabel : `PropertyMap` (page 35) (optional, default: `None`)

Constraint labels to be imposed during reordering. Only has effect if `reorder == True`.

niter : `int` (optional, default: *0*)

Number of iterations performed to obtain the best ordering. If `niter == 0` it will automatically determined. Only has effect if `reorder == True`.

ret_order : `bool` (optional, default: `False`)

If `True`, the vertex ordering is returned. Only has effect if `reorder == True`.

Examples

```
>>> g = gt.collection.data["polbooks"]
>>> state = gt.BlockState(g, B=5, deg_corr=True)
>>> for i in range(1000):
...     ds, nmoves = gt.mcmc_sweep(state)
>>> m = state.get_matrix(reorder=True)
>>> figure()
<...>
>>> matshow(m)
<...>
>>> savefig("bloc_mat.pdf")
```

get_nr(*self*)
Returns the vertex property map of the block graph which contains the block sizes n_r.

join(*self*, *r*, *s*)
Merge blocks *r* and *s* into a single block.

modularity(*self*)
Computes the modularity of the current block structure.

move_vertex(*self*, *v*, *nr*)
Move vertex *v* to block *r*, and return the entropy difference.

remove_vertex(*self*, *v*)
Remove vertex *v* from its current block.

graph_tool.community.**mcmc_sweep**(*state*, *beta=1.0*, *sequential=True*, *verbose=False*, *vertices=None*)
Performs a Monte Carlo Markov chain sweep on the network, to sample the block partition according to a probability $\propto e^{-\beta \mathcal{S}_{t/c}}$, where $\mathcal{S}_{t/c}$ is the blockmodel entropy.

Parameters **state** : `BlockState` (page 70)

The block state.

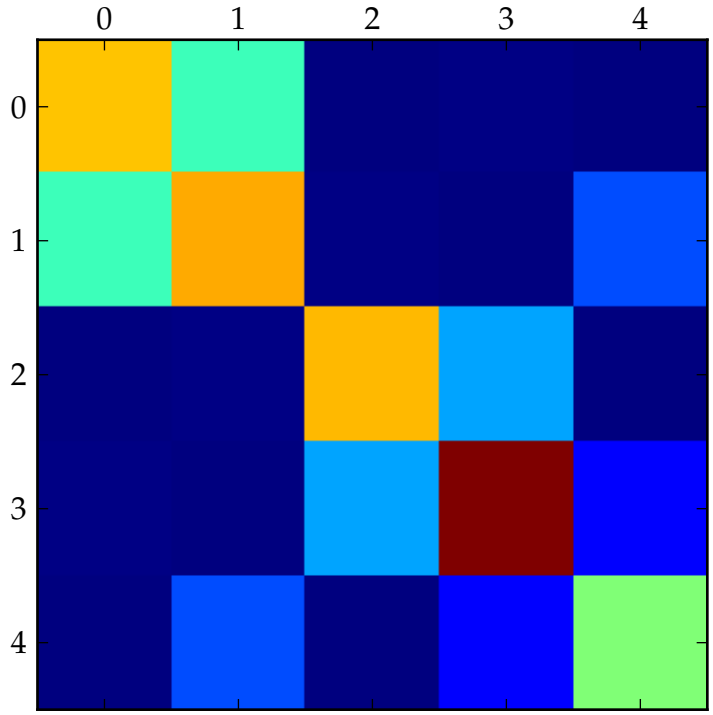


Figure 3.12: A 5x5 block matrix.

beta : *float* (optional, default: *1.0*)

The inverse temperature parameter β.

sequential : `bool` (optional, default: `True`)

If `True`, the move attempts on the vertices are done in sequential random order. Otherwise a total of N moves attempts are made, where N is the number of vertices, where each vertex can be selected with equal probability.

verbose : `bool` (optional, default: `False`)

If `True`, verbose information is displayed.

Returns **dS** : *float*

The entropy difference (per edge) after a full sweep.

nmoves : `int`

The number of accepted block membership moves.

Notes

This algorithm performs a Monte Carlo Markov chain sweep on the network, where the block memberships are randomly moved, and either accepted or rejected, so that after sufficiently many sweeps the partitions are sampled with probability proportional to

$e^{-\beta\mathcal{S}_{t/c}}$, where $\mathcal{S}_{t/c}$ is the blockmodel entropy, given by

$$\mathcal{S}_t \cong -\frac{1}{2}\sum_{rs} e_{rs}\ln\left(\frac{e_{rs}}{n_r n_s}\right),$$
$$\mathcal{S}_t^d \cong -\sum_{rs} e_{rs}\ln\left(\frac{e_{rs}}{n_r n_s}\right),$$

for undirected and directed traditional blockmodels (`deg_corr == False`), respectively, where e_{rs} is the number of edges from block r to s (or the number of half-edges for the undirected case when $r = s$), and n_r is the number of vertices in block r, and constant terms which are independent of the block partition were dropped (see `BlockState.entropy()` (page 72) for the complete entropy). For the degree-corrected variant with "hard" degree constraints the equivalent expressions are

$$\mathcal{S}_c \cong -\frac{1}{2}\sum_{rs} e_{rs}\ln\left(\frac{e_{rs}}{e_r e_s}\right),$$
$$\mathcal{S}_c^d \cong -\sum_{rs} e_{rs}\ln\left(\frac{e_{rs}}{e_r^+ e_s^-}\right),$$

where $e_r = \sum_s e_{rs}$ is the number of half-edges incident on block r, and $e_r^+ = \sum_s e_{rs}$ and $e_r^- = \sum_s e_{sr}$ are the number of out- and in-edges adjacent to block r, respectively.

The Monte Carlo algorithm employed attempts to improve the mixing time of the markov chain by proposing membership moves $r \to s$ with probability $p(r \to s|t) \propto e_{ts} + 1$, where t is the block label of a random neighbour of the vertex being moved. See [peixoto-parsimonious-2013] (page **??**) for more details.

This algorithm has a complexity of $O(E)$, where E is the number of edges in the network.

References

[holland-stochastic-1983] (page **??**), [faust-blockmodels-1992] (page **??**), [karrer-stochastic-2011] (page **??**), [peixoto-entropy-2012] (page **??**), [peixoto-parsimonious-2013] (page **??**)

Examples

```
>>> g = gt.collection.data["polbooks"]
>>> state = gt.BlockState(g, B=3, deg_corr=True)
>>> pv = None
>>> for i in range(1000):         # remove part of the transient
...     ds, nmoves = gt.mcmc_sweep(state)
>>> for i in range(1000):
...     ds, nmoves = gt.mcmc_sweep(state)
...     pv = gt.collect_vertex_marginals(state, pv)
>>> gt.graph_draw(g, pos=g.vp["pos"], vertex_shape="pie", vertex_pie_fractions=pv, output="po
<...>
```

graph_tool.community.**collect_edge_marginals**(*state*, *p=None*)

Collect the edge marginal histogram, which counts the number of times the endpoints of each node have been assigned to a given block pair.

This should be called multiple times, after repeated runs of the `mcmc_sweep()` (page 74) function.

Parameters **state** : `BlockState` (page 70)

The block state.

p : `PropertyMap` (page 35) (optional, default: `None`)

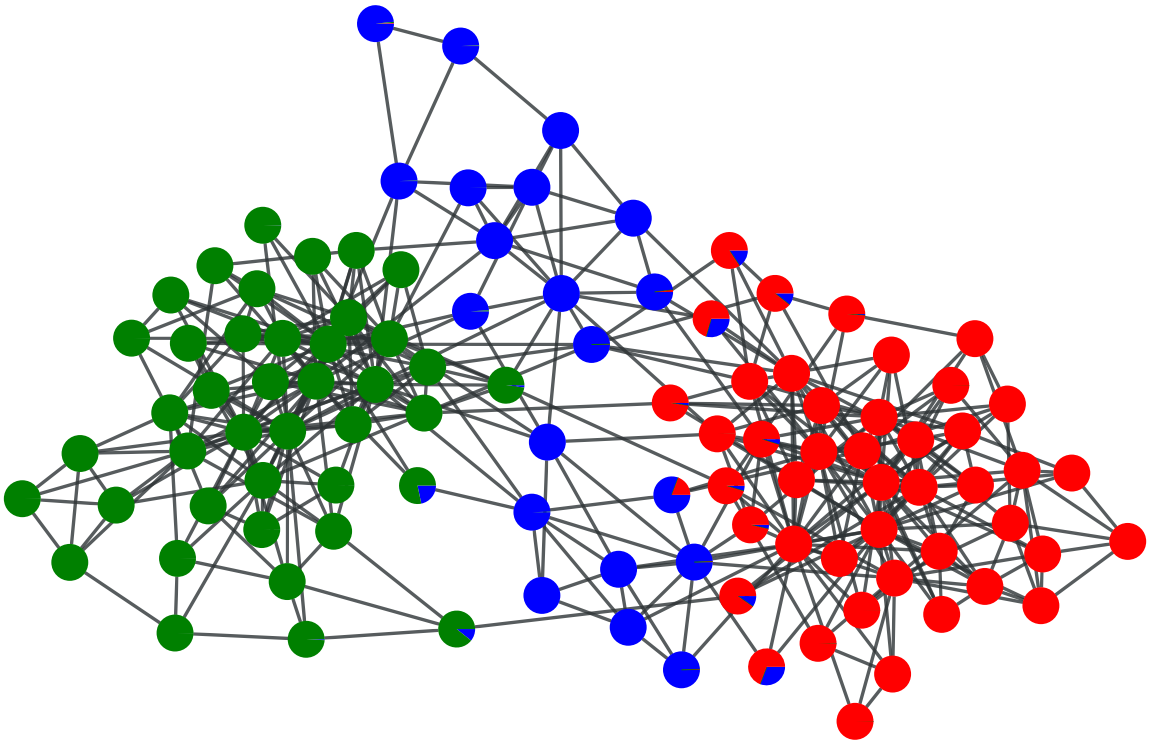


Figure 3.13: “Soft” block partition of a political books network with $B = 3$.

Edge property map with vector-type values, storing the previous block membership counts. Each vector entry corresponds to `b[i] + B * b[j]`, where `b` is the block membership and `i = min(source(e), target(e))` and `j = max(source(e), target(e))`. If not provided, an empty histogram will be created.

Returns **p** : `PropertyMap` (page 35) (optional, default: `None`)

Vertex property map with vector-type values, storing the accumulated block membership counts.

Examples

```
>>> g = gt.collection.data["polbooks"]
>>> state = gt.BlockState(g, B=4, deg_corr=True)
>>> pe = None
>>> for i in range(1000):        # remove part of the transient
...     ds, nmoves = gt.mcmc_sweep(state)
>>> for i in range(1000):
...     ds, nmoves = gt.mcmc_sweep(state)
...     pe = gt.collect_edge_marginals(state, pe)
>>> gt.bethe_entropy(state, pe)[0]
17.17007316834295
```

graph_tool.community.**collect_vertex_marginals**(*state*, *p=None*)

Collect the vertex marginal histogram, which counts the number of times a node was assigned to a given block.

This should be called multiple times, after repeated runs of the `mcmc_sweep()` (page 74) function.

Parameters **state** : `BlockState` (page 70)

The block state.

p : `PropertyMap` (page 35) (optional, default: `None`)

Vertex property map with vector-type values, storing the previous block membership counts. If not provided, an empty histogram will be created.

Returns **p** : `PropertyMap` (page 35) (optional, default: `None`)

Vertex property map with vector-type values, storing the accumulated block membership counts.

Examples

```
>>> g = gt.collection.data["polbooks"]
>>> state = gt.BlockState(g, B=4, deg_corr=True)
>>> pv = None
>>> for i in range(1000):        # remove part of the transient
...     ds, nmoves = gt.mcmc_sweep(state)
>>> for i in range(1000):
...     ds, nmoves = gt.mcmc_sweep(state)
...     pv = gt.collect_vertex_marginals(state, pv)
>>> gt.mf_entropy(state, pv)
20.001666525168361
>>> gt.graph_draw(g, pos=g.vp["pos"], vertex_shape="pie", vertex_pie_fractions=pv, output="pc
<...>
```

graph_tool.community.**bethe_entropy**(*state*, *p*)

Compute the Bethe entropy given the edge block membership marginals.

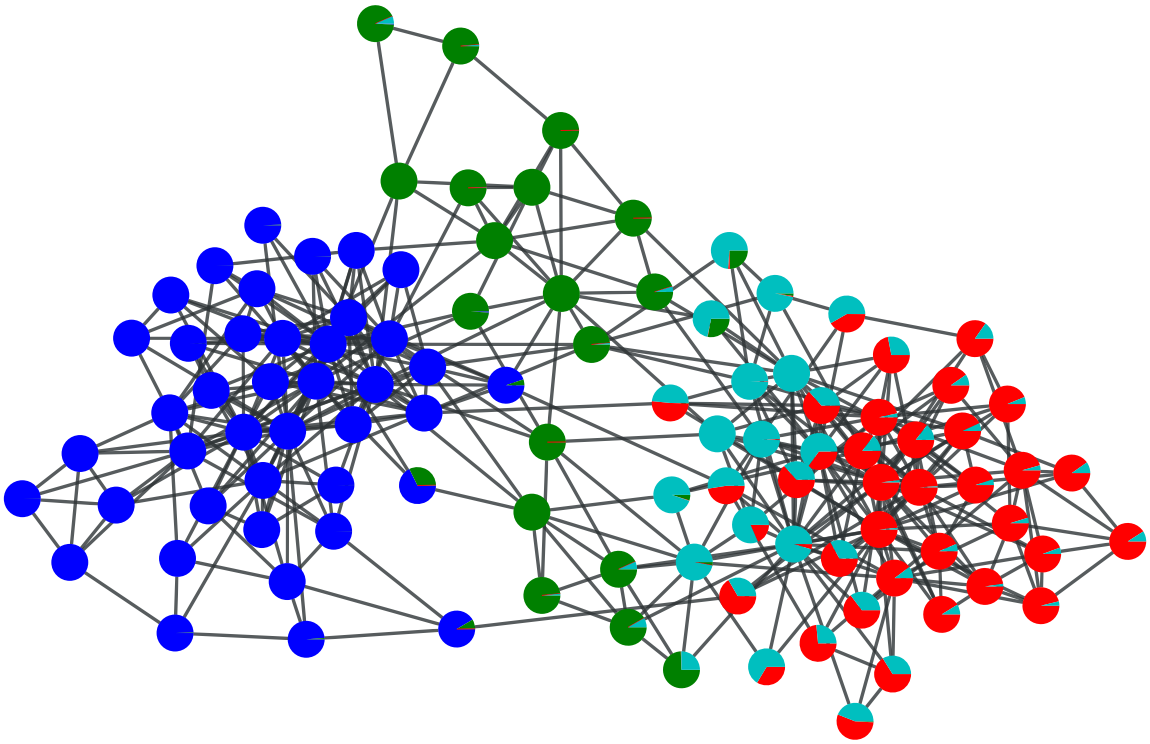


Figure 3.14: “Soft” block partition of a political books network with $B = 4$.

Parameters **state** : `BlockState` (page 70)

The block state.

p : `PropertyMap` (page 35)

Edge property map with vector-type values, storing the previous block membership counts. Each vector entry corresponds to `b[i] + B * b[j]`, where `b` is the block membership and `i = min(source(e), target(e))` and `j = max(source(e), target(e))`.

Returns **H** : `float`

The Bethe entropy value (in nats)

Hmf : `float`

The "mean field" entropy value (in nats), as would be returned by the `mf_entropy()` (page 80) function.

pv : `PropertyMap` (page 35) (optional, default: `None`)

Vertex property map with vector-type values, storing the accumulated block membership counts. These are the node marginals, as would be returned by the `collect_vertex_marginals()` (page 78) function.

Notes

The Bethe entropy is defined as,

$$H = -\sum_{e,(r,s)} \pi^e_{(r,s)} \ln \pi^e_{(r,s)} - \sum_{v,r} (1 - k_i)\pi^v_r \ln \pi^v_r,$$

where $\pi^e_{(r,s)}$ is the marginal probability that the endpoints of the edge e belong to blocks (r, s), and π^v_r is the marginal probability that vertex v belongs to block r, and k_i is the degree of vertex v (or total degree for directed graphs).

References

[mezard-information-2009] (page **??**)

`graph_tool.community.`**`mf_entropy`**`(`*state*, *p*`)`

Compute the "mean field" entropy given the vertex block membership marginals.

Parameters **state** : `BlockState` (page 70)

The block state.

p : `PropertyMap` (page 35)

Vertex property map with vector-type values, storing the accumulated block membership counts.

Returns **Hmf** : `float`

The "mean field" entropy value (in nats).

Notes

The "mean field" entropy is defined as,

$$H = -\sum_{v,r} \pi_r^v \ln \pi_r^v,$$

where π_r^v is the marginal probability that vertex v belongs to block r.

References

[mezard-information-2009] (page **??**)

graph_tool.community.**model_entropy**(*B*, *N*, *E*, *directed*)

Computes the amount of information necessary for the parameters of the traditional blockmodel ensemble, for `B` blocks, `N` vertices, `E` edges, and either a directed or undirected graph.

A traditional blockmodel is defined as a set of N vertices which can belong to one of B blocks, and the matrix e_{rs} describes the number of edges from block r to s (or twice that number if $r = s$ and the graph is undirected).

For an undirected graph, the number of distinct e_{rs} matrices is given by,

$$\Omega_m = \left(\!\!\binom{\binom{B}{2}}{E}\!\!\right)$$

and for a directed graph,

$$\Omega_m = \left(\!\!\binom{B^2}{E}\!\!\right)$$

where $\left(\!\binom{n}{k}\!\right) = \binom{n+k-1}{k}$ is the number of k combinations with repetitions from a set of size n.

The total information necessary to describe the model is then,

$$\mathcal{L}_t = \ln \Omega_m + N \ln B,$$

where $N \ln B$ is the information necessary to describe the block partition.

References

[peixoto-parsimonious-2013] (page **??**)

graph_tool.community.**get_max_B**(*N*, *E*, *directed=False*)

Return the maximum detectable number of blocks, obtained by minimizing:

$$\mathcal{L}_t(B, N, E) - E \ln B$$

where $\mathcal{L}_t(B, N, E)$ is the information necessary to describe a traditional blockmodel with B blocks, N nodes and E edges (see `model_entropy()` (page 81)).

References

[peixoto-parsimonious-2013] (page **??**)

Examples

```
>>> gt.get_max_B(N=1e6, E=5e6)
1572
```

graph_tool.community.**get_akc**(*B*, *I*, *N=inf*, *directed=False*)

Return the minimum value of the average degree of the network, so that some block structure with B blocks can be detected, according to the minimum description length criterion.

This is obtained by solving

$$\Sigma_b = \mathcal{L}_t(B, N, E) - E\mathcal{I}_{t/c} = 0,$$

where $\mathcal{L}_t$ is the necessary information to describe the blockmodel parameters (see `model_entropy()` (page 81)), and $\mathcal{I}_{t/c}$ characterizes the planted block structure, and is given by

$$\mathcal{I}_t = \sum_{rs} m_{rs} \ln\left(\frac{m_{rs}}{w_r w_s}\right),$$
$$\mathcal{I}_c = \sum_{rs} m_{rs} \ln\left(\frac{m_{rs}}{m_r m_s}\right),$$

where $m_{rs} = e_{rs}/2E$ (or $m_{rs} = e_{rs}/E$ for directed graphs) and $w_r = n_r/N$. We note that $\mathcal{I}_{t/c} \in [0, \ln B]$. In the case where $E \gg B^2$, this simplifies to

$$\langle k \rangle_c = \frac{2\ln B}{\mathcal{I}_{t/c}},$$
$$\left\langle k^{-/+} \right\rangle_c = \frac{\ln B}{\mathcal{I}_{t/c}},$$

for undirected and directed graphs, respectively. This limit is assumed if `N == inf`.

References

[peixoto-parsimonious-2013] (page **??**)

Examples

```
>>> gt.get_akc(10, log(10) / 100, N=100)
2.4199998936261204
```

graph_tool.community.**min_dist**(*state*, *n=0*)

Return the minimum distance between all blocks, and the block pair which minimizes it.

The parameter *state* must be an instance of the `BlockState` (page 70) class, and *n* is the number of block pairs to sample. If *n == 0* all block pairs are sampled.

Examples

```
>>> g = gt.collection.data["polbooks"]
>>> state = gt.BlockState(g, B=4, deg_corr=True)
>>> for i in range(1000):
...     ds, nmoves = gt.mcmc_sweep(state)
>>> gt.min_dist(state)
(795.7694502418633, 2, 3)
```

graph_tool.community.**condensation_graph**(*g*, *prop*, *vweight=None*, *eweight=None*, *avprops=None*, *aeprops=None*, *self_loops=False*)

Obtain the condensation graph, where each vertex with the same 'prop' value is condensed in one vertex.

Parameters **g** : Graph (page 28)

Graph to be used.

prop : PropertyMap (page 35)

Vertex property map with the community partition.

vweight : PropertyMap (page 35) (optional, default: None)

Vertex property map with the optional vertex weights.

eweight : PropertyMap (page 35) (optional, default: None)

Edge property map with the optional edge weights.

avprops : list of PropertyMap (page 35) (optional, default: None)

If provided, the average value of each property map in this list for each vertex in the condensed graph will be computed and returned.

aeprops : list of PropertyMap (page 35) (optional, default: None)

If provided, the average value of each property map in this list for each edge in the condensed graph will be computed and returned.

self_loops : bool (optional, default: False)

If True, self-loops due to intra-block edges are also included in the condensation graph.

Returns **condensation_graph** : Graph (page 28)

The community network

prop : PropertyMap (page 35)

The community values.

vcount : PropertyMap (page 35)

A vertex property map with the vertex count for each community.

ecount : PropertyMap (page 35)

An edge property map with the inter-community edge count for each edge.

va : list of PropertyMap (page 35)

A list of vertex property maps with average values of the properties passed via the avprops parameter.

ea : list of PropertyMap (page 35)

A list of edge property maps with average values of the properties passed via the avprops parameter.

See Also:

community_structure (page 84) Obtain the community structure

modularity (page 92) Calculate the network modularity

condensation_graph (page 82) Network of communities, or blocks

Notes

Each vertex in the condensation graph represents one community in the original graph (vertices with the same 'prop' value), and the edges represent existent edges between vertices of the respective communities in the original graph.

Examples

Let's first obtain the best block partition with `B=5`.

```
>>> g = gt.collection.data["polbooks"]
>>> state = gt.BlockState(g, B=5, deg_corr=True)
>>> for i in range(1000):         # remove part of the transient
...     ds, nmoves = gt.mcmc_sweep(state)
>>> for i in range(1000):
...     ds, nmoves = gt.mcmc_sweep(state, beta=float("inf"))
>>> b = state.get_blocks()
>>> gt.graph_draw(g, pos=g.vp["pos"], vertex_fill_color=b, vertex_shape=b, output="polbooks_b
<...>
```

Now we get the condensation graph:

```
>>> bg, bb, vcount, ecount, avp, aep = gt.condensation_graph(g, b, avprops=[g.vp["pos"]], sel
>>> gt.graph_draw(bg, pos=avp[0], vertex_fill_color=bb, vertex_shape=bb,
...               vertex_size=gt.prop_to_size(vcount, mi=40, ma=100),
...               edge_pen_width=gt.prop_to_size(ecount, mi=2, ma=10),
...               output="polbooks_blocks_B5_cond.pdf")
<...>
```

graph_tool.community.**community_structure**(*g*, *n_iter*, *n_spins*, *gamma=1.0*, *corr='erdos'*, *spins=None*, *weight=None*, *t_range=(100.0, 0.01)*, *verbose=False*, *history_file=None*)

Obtain the community structure for the given graph, using a Potts model approach.

Parameters **g** : Graph (page 28)

Graph to be used.

n_iter : int

Number of iterations.

n_spins : int

Number of maximum spins to be used.

gamma : float (optional, default: 1.0)

The γ parameter of the hamiltonian.

corr : string (optional, default: "erdos")

Type of correlation to be assumed: Either "erdos", "uncorrelated" and "correlated".

spins : PropertyMap (page 35)

Vertex property maps to store the spin variables. If this is specified, the values will not be initialized to a random value.

weight : PropertyMap (page 35) (optional, default: None)

Edge property map with the optional edge weights.

t_range : tuple of floats (optional, default: (100.0, 0.01))

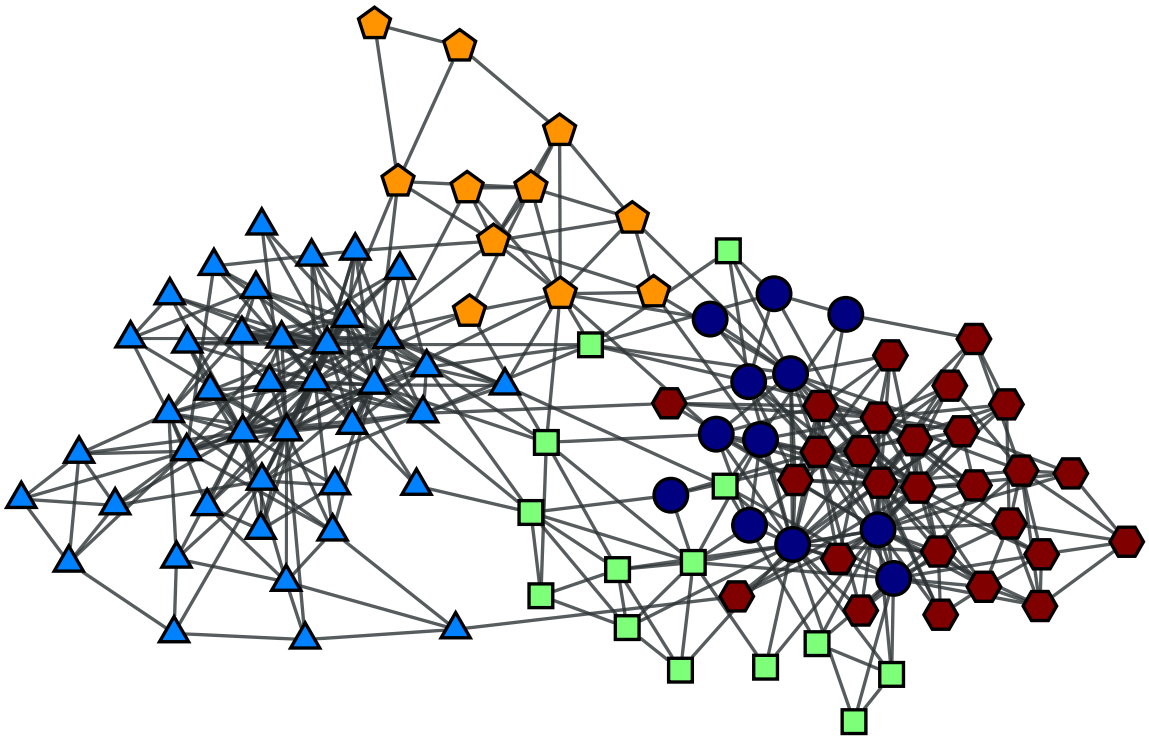


Figure 3.15: Block partition of a political books network with $B = 5$.

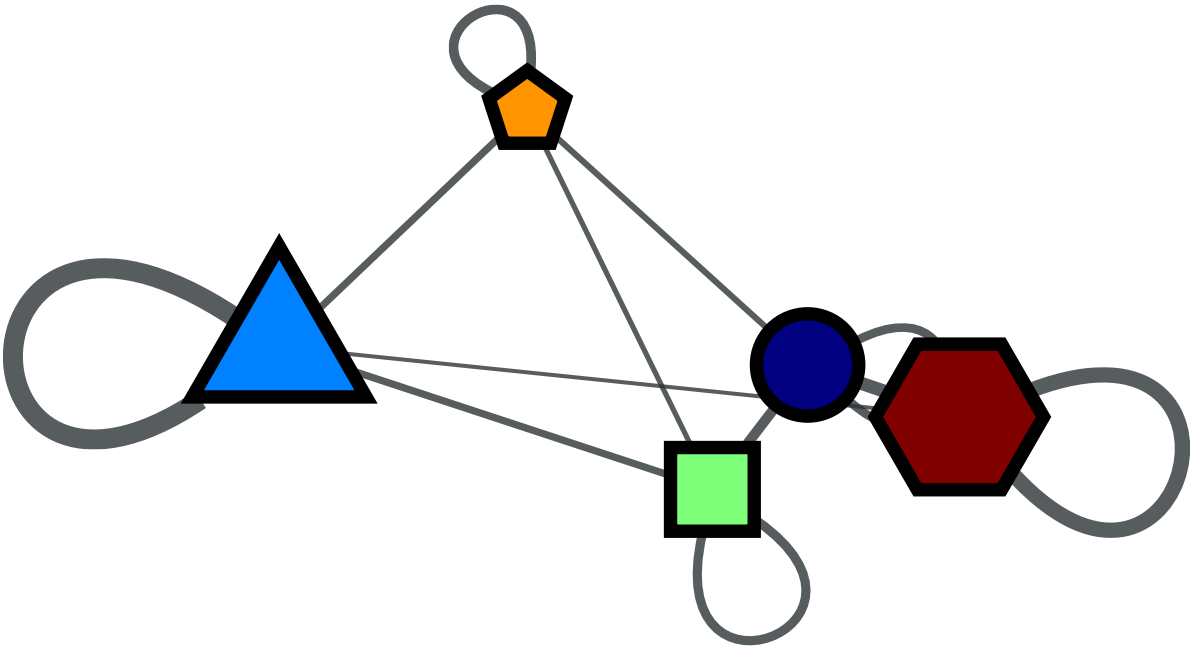


Figure 3.16: Condensation graph of the obtained block partition.

Temperature range.

verbose : bool (optional, default: False)

Display verbose information.

history_file : string (optional, default: None)

History file to keep information about the simulated annealing.

Returns **spins** : `PropertyMap` (page 35)

Vertex property map with the spin values.

See Also:

`community_structure` (page 84) Obtain the community structure

`modularity` (page 92) Calculate the network modularity

`condensation_graph` (page 82) Network of communities, or blocks

Notes

The method of community detection covered here is an implementation of what was proposed in [reichard-statistical-2006] (page 229). It consists of a simulated annealing algorithm which tries to minimize the following hamiltonian:

$$\mathcal{H}(\{\sigma\}) = -\sum_{i\neq j}\left(A_{ij} - \gamma p_{ij}\right)\delta(\sigma_i,\sigma_j)$$

where p_{ij} is the probability of vertices i and j being connected, which reduces the problem of community detection to finding the ground states of a Potts spin-glass model. It can be shown that minimizing this hamiltonan, with $\gamma = 1$, is equivalent to maximizing Newman's modularity ([newman-modularity-2006] (page **??**)). By increasing the parameter γ, it's possible also to find sub-communities.

It is possible to select three policies for choosing p_{ij} and thus choosing the null model: "erdos" selects a Erdos-Reyni random graph, "uncorrelated" selects an arbitrary random graph with no vertex-vertex correlations, and "correlated" selects a random graph with average correlation taken from the graph itself. Optionally a weight property can be given by the *weight* option.

The most important parameters for the algorithm are the initial and final temperatures (*t_range*), and total number of iterations (*max_iter*). It normally takes some trial and error to determine the best values for a specific graph. To help with this, the *history* option can be used, which saves to a chosen file the temperature and number of spins per iteration, which can be used to determined whether or not the algorithm converged to the optimal solution. Also, the *verbose* option prints the computation status on the terminal.

Note: If the spin property already exists before the computation starts, it's not resampled at the beginning. This means that it's possible to continue a previous run, if you saved the graph, by properly setting *t_range* value, and using the same *spin* property.

If enabled during compilation, this algorithm runs in parallel.

References

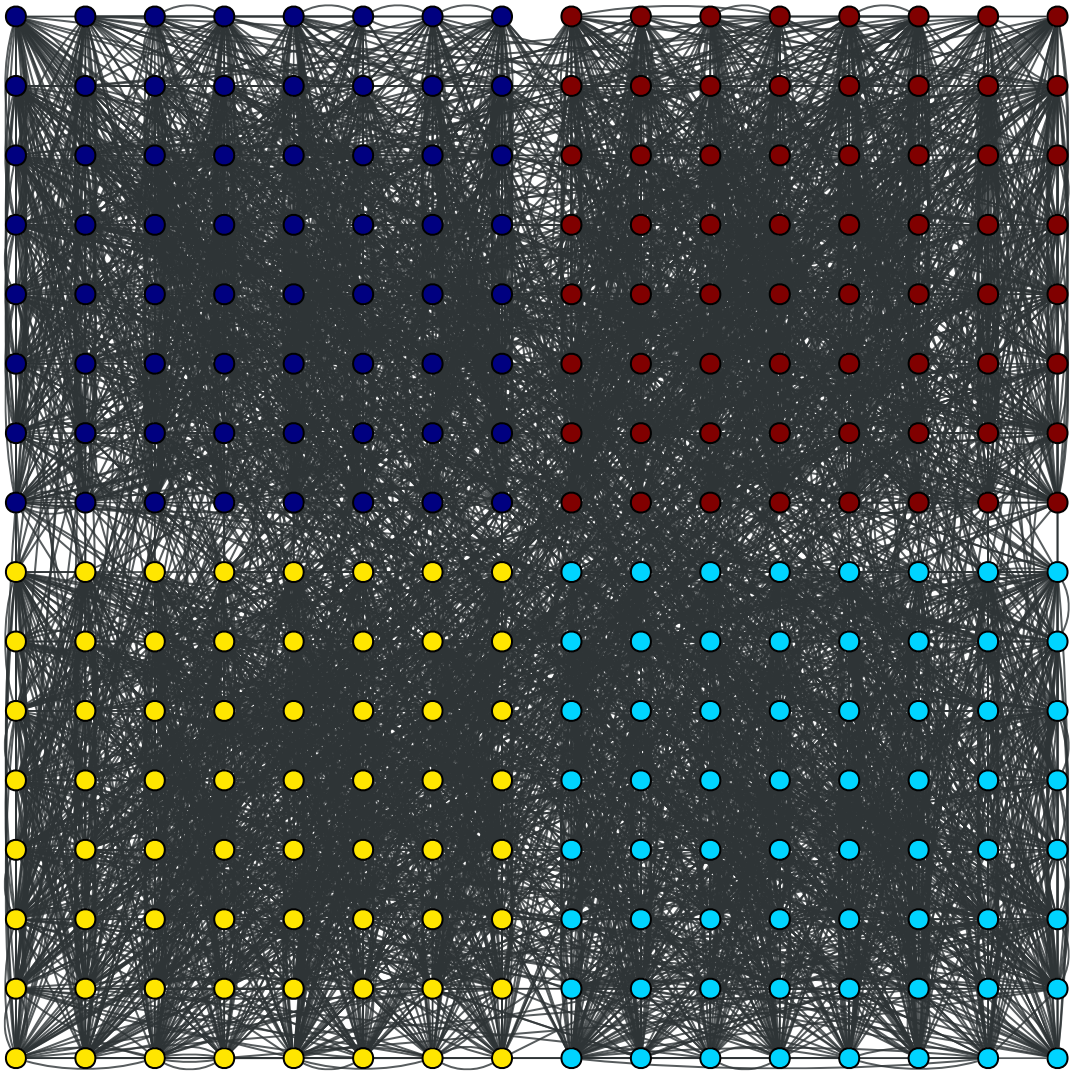
[reichard-statistical-2006] (page 229), [newman-modularity-2006] (page **??**)

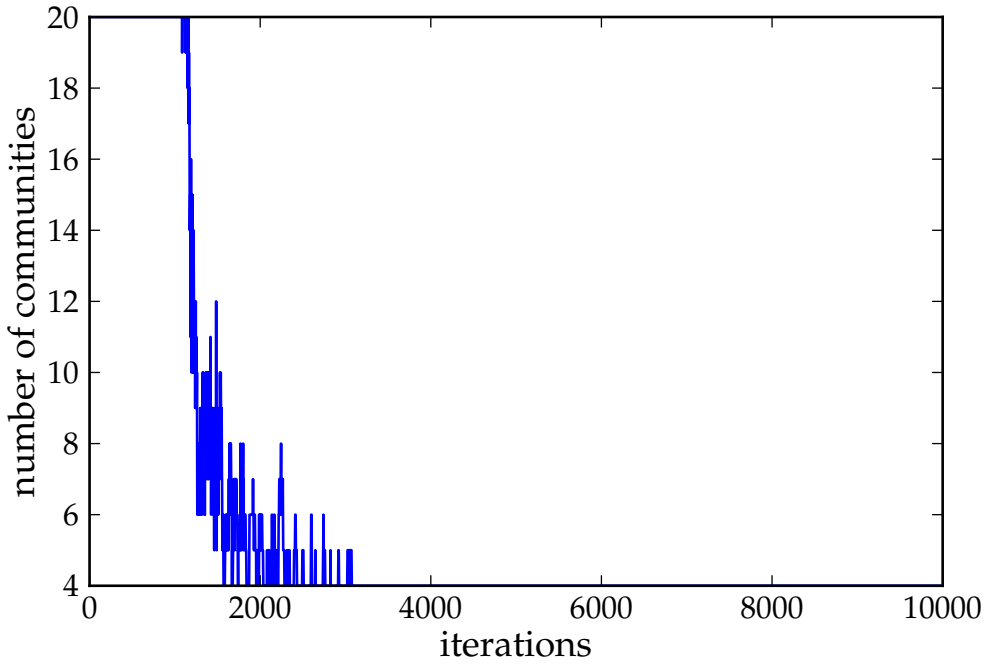
Examples

This example uses the network `community.xml`.

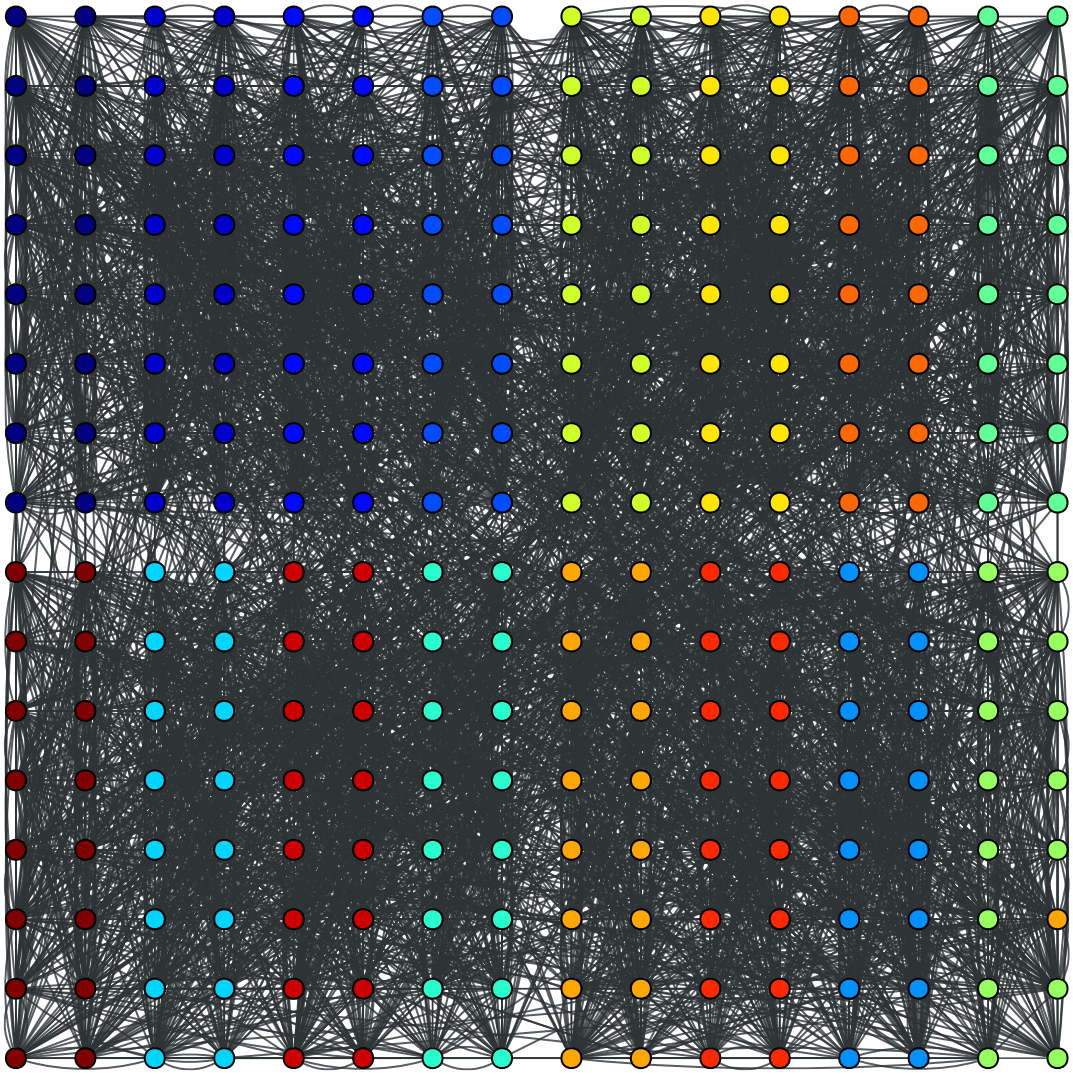
```
>>> from pylab import *
>>> from numpy.random import seed
>>> seed(42)
>>> g = gt.load_graph("community.xml")
>>> pos = g.vertex_properties["pos"]
>>> spins = gt.community_structure(g, 10000, 20, t_range=(5, 0.1),
...                                history_file="community-history1")
>>> gt.graph_draw(g, pos=pos, vertex_fill_color=spins, output_size=(420, 420), output="comm1.
<...>
>>> spins = gt.community_structure(g, 10000, 40, t_range=(5, 0.1),
...                                gamma=2.5, history_file="community-history2")
>>> gt.graph_draw(g, pos=pos, vertex_fill_color=spins, output_size=(420, 420), output="comm2.
<...>
>>> figure(figsize=(6, 4))
<...>
>>> xlabel("iterations")
<...>
>>> ylabel("number of communities")
<...>
>>> a = loadtxt("community-history1").transpose()
>>> plot(a[0], a[2])
[...]
>>> savefig("comm1-hist.pdf")
>>> clf()
>>> xlabel("iterations")
<...>
>>> ylabel("number of communities")
<...>
>>> a = loadtxt("community-history2").transpose()
>>> plot(a[0], a[2])
[...]
>>> savefig("comm2-hist.pdf")
```

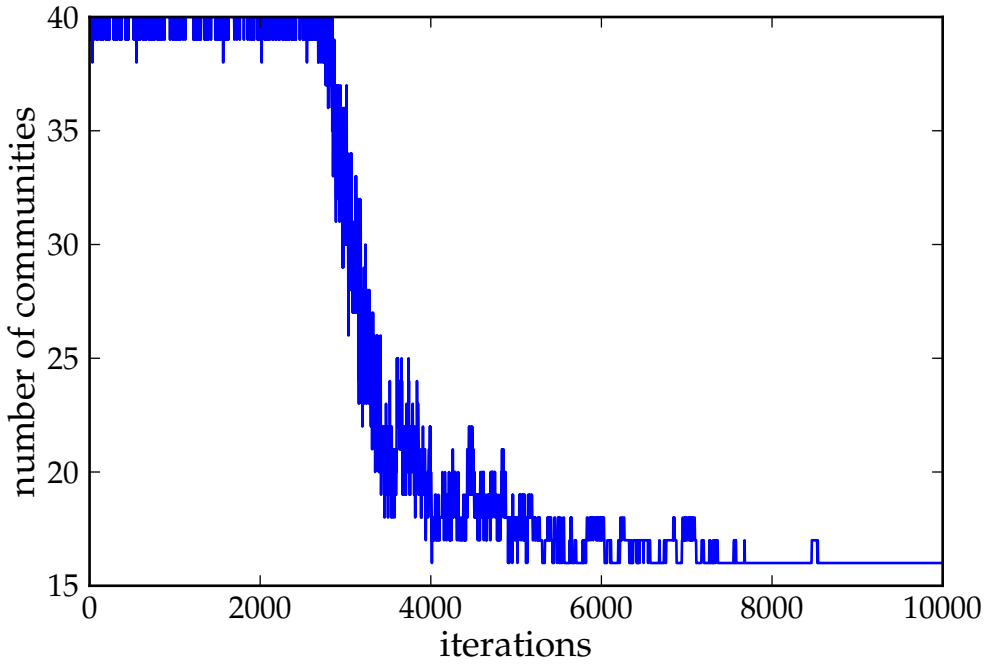
The community structure with $\gamma = 1$:





The community structure with $\gamma = 2.5$:





graph_tool.community.**modularity**(*g*, *prop*, *weight=None*)

Calculate Newman's modularity.

Parameters **g** : Graph (page 28)

Graph to be used.

prop : PropertyMap (page 35)

Vertex property map with the community partition.

weight : PropertyMap (page 35) (optional, default: None)

Edge property map with the optional edge weights.

Returns **modularity** : float

Newman's modularity.

See Also:

community_structure (page 84) obtain the community structure

modularity (page 92) calculate the network modularity

condensation_graph (page 82) Network of communities, or blocks

Notes

Given a specific graph partition specified by *prop*, Newman's modularity [newman-modularity-2006] (page **??**) is defined by:

$$Q = \sum_s e_{ss} - \left(\sum_r e_{rs}\right)^2$$

where e_{rs} is the fraction of edges which fall between vertices with spin s and r.

If enabled during compilation, this algorithm runs in parallel.

References

[newman-modularity-2006] (page **??**)

Examples

```
>>> from pylab import *
>>> from numpy.random import seed
>>> seed(42)
>>> g = gt.load_graph("community.xml")
>>> spins = gt.community_structure(g, 10000, 10)
>>> gt.modularity(g, spins)
0.535314188562404
```

3.2.5 graph_tool.correlations - Correlations

Summary

assortativity (page 93)	Obtain the assortativity coefficient for the given graph.
scalar_assortativity (page 94)	Obtain the scalar assortativity coefficient for the given graph.
corr_hist (page 95)	Obtain the correlation histogram for the given graph.
combined_corr_hist (page 97)	Obtain the single-vertex combined correlation histogram for the given
avg_neighbour_corr (page 98)	Obtain the average neighbour-neighbour correlation for the given gra
avg_combined_corr (page 100)	Obtain the single-vertex combined correlation histogram for the given

Contents

graph_tool.correlations.**assortativity**(*g*, *deg*)

Obtain the assortativity coefficient for the given graph.

Parameters **g** : Graph (page 28)

Graph to be used.

deg : string or PropertyMap (page 35)

degree type ("in", "out" or "total") or vertex property map, which specifies the vertex types.

Returns **assortativity coefficient** : tuple of two floats

The assortativity coefficient, and its variance.

See Also:

assortativity (page 93) assortativity coefficient

scalar_assortativity (page 94) scalar assortativity coefficient

corr_hist (page 95) vertex-vertex correlation histogram

combined_corr_hist (page 97) combined single-vertex correlation histogram

avg_neighbour_corr (page 98) average nearest-neighbour correlation

avg_combined_corr (page 100) average combined single-vertex correlation

Notes

The assortativity coefficient [newman-mixing-2003] (page **??**) tells in a concise fashion how vertices of different types are preferentially connected amongst themselves, and is defined by

$$r = \frac{\sum_i e_{ii} - \sum_i a_i b_i}{1 - \sum_i a_i b_i}$$

where $a_i = \sum_j e_{ij}$ and $b_j = \sum_i e_{ij}$, and e_{ij} is the fraction of edges from a vertex of type i to a vertex of type j.

The variance is obtained with the jackknife method.

If enabled during compilation, this algorithm runs in parallel.

References

[newman-mixing-2003] (page **??**)

Examples

```
>>> def sample_k(max):
...     accept = False
...     while not accept:
...         k = np.random.randint(1,max+1)
...         accept = random() < 1.0/k
...     return k
...
>>> g = gt.random_graph(1000, lambda: sample_k(40), model="probabilistic",
...                     vertex_corr=lambda i,k: 1.0 / (1 + abs(i - k)), directed=False,
...                     n_iter=100)
>>> gt.assortativity(g, "out")
(0.13903518011375607, 0.005051876804786422)
```

graph_tool.correlations.**scalar_assortativity**(*g*, *deg*)

Obtain the scalar assortativity coefficient for the given graph.

Parameters **g** : Graph (page 28)

Graph to be used.

deg : string or PropertyMap (page 35)

degree type ("in", "out" or "total") or vertex property map, which specifies the vertex types.

Returns **scalar assortativity coefficient** : tuple of two floats

The scalar assortativity coefficient, and its variance.

See Also:

assortativity (page 93) assortativity coefficient

scalar_assortativity (page 94) scalar assortativity coefficient

corr_hist (page 95) vertex-vertex correlation histogram

combined_corr_hist (page 97) combined single-vertex correlation histogram

avg_neighbour_corr (page 98) average nearest-neighbour correlation

avg_combined_corr (page 100) average combined single-vertex correlation

Notes

The scalar assortativity coefficient [newman-mixing-2003] (page **??**) tells in a concise fashion how vertices of different types are preferentially connected amongst themselves, and is defined by

$$r = \frac{\sum_{xy} xy(e_{xy} - a_x b_y)}{\sigma_a \sigma_b}$$

where $a_x = \sum_y e_{xy}$ and $b_y = \sum_x e_{xy}$, and e_{xy} is the fraction of edges from a vertex of type x to a vertex of type y.

The variance is obtained with the jackknife method.

If enabled during compilation, this algorithm runs in parallel.

References

[newman-mixing-2003] (page **??**)

Examples

```
>>> def sample_k(max):
...     accept = False
...     while not accept:
...         k = np.random.randint(1,max+1)
...         accept = random() < 1.0/k
...     return k
...
>>> g = gt.random_graph(1000, lambda: sample_k(40), model="probabilistic",
...                     vertex_corr=lambda i,k: abs(i-k),
...                     directed=False, n_iter=100)
>>> gt.scalar_assortativity(g, "out")
(-0.44070158356400696, 0.010592022444678632)

>>> g = gt.random_graph(1000, lambda: sample_k(40), model="probabilistic",
...                     vertex_corr=lambda i, k: 1.0 / (1 + abs(i - k)),
...                     directed=False, n_iter=100)
>>> gt.scalar_assortativity(g, "out")
(0.6007430887839058, 0.011569809783643956)
```

graph_tool.correlations.**corr_hist**(*g*, *deg_source*, *deg_target*, *bins=[[0, 1], [0, 1]]*, *weight=None*, *float_count=True*)

Obtain the correlation histogram for the given graph.

Parameters **g** : Graph (page 28)

Graph to be used.

deg_source : string or PropertyMap (page 35)

degree type ("in", "out" or "total") or vertex property map for the source vertex.

deg_target : string or PropertyMap (page 35)

degree type ("in", "out" or "total") or vertex property map for the target vertex.

bins : list of lists (optional, default: [[0, 1], [0, 1]])

A list of bin edges to be used for the source and target degrees. If any list has size 2, it is used to create an automatically generated bin range starting from the first value, and with constant bin width given by the second value.

weight : edge property map (optional, default: None)

Weight (multiplicative factor) to be used on each edge.

float_count : bool (optional, default: True)

If True, the bin counts are converted float variables, which is useful for normalization, and other processing. It False, the bin counts will be unsigned integers.

Returns **bin_counts** : ndarray

Two-dimensional array with the bin counts.

source_bins : ndarray

Source degree bins

target_bins : ndarray

Target degree bins

See Also:

assortativity (page 93) assortativity coefficient

scalar_assortativity (page 94) scalar assortativity coefficient

corr_hist (page 95) vertex-vertex correlation histogram

combined_corr_hist (page 97) combined single-vertex correlation histogram

avg_neighbour_corr (page 98) average nearest-neighbour correlation

avg_combined_corr (page 100) average combined single-vertex correlation

Notes

The correlation histogram counts, for every vertex with degree (or scalar property) 'source_deg', the number of out-neighbours with degree (or scalar property) 'target_deg'.

If enabled during compilation, this algorithm runs in parallel.

Examples

```
>>> def sample_k(max):
...     accept = False
...     while not accept:
...         k = np.random.randint(1,max+1)
...         accept = random() < 1.0/k
...     return k
...
>>> g = gt.random_graph(10000, lambda: sample_k(40), model="probabilistic",
...                     vertex_corr=lambda i, j: (sin(i / pi) * sin(j / pi) + 1) / 2,
...                     directed=False, n_iter=100)
>>> h = gt.corr_hist(g, "out", "out")
>>> clf()
>>> xlabel("Source out-degree")
<...>
>>> ylabel("Target out-degree")
<...>
```

```
>>> imshow(h[0].T, interpolation="nearest", origin="lower")
<...>
>>> colorbar()
<...>
>>> savefig("corr.pdf")
```

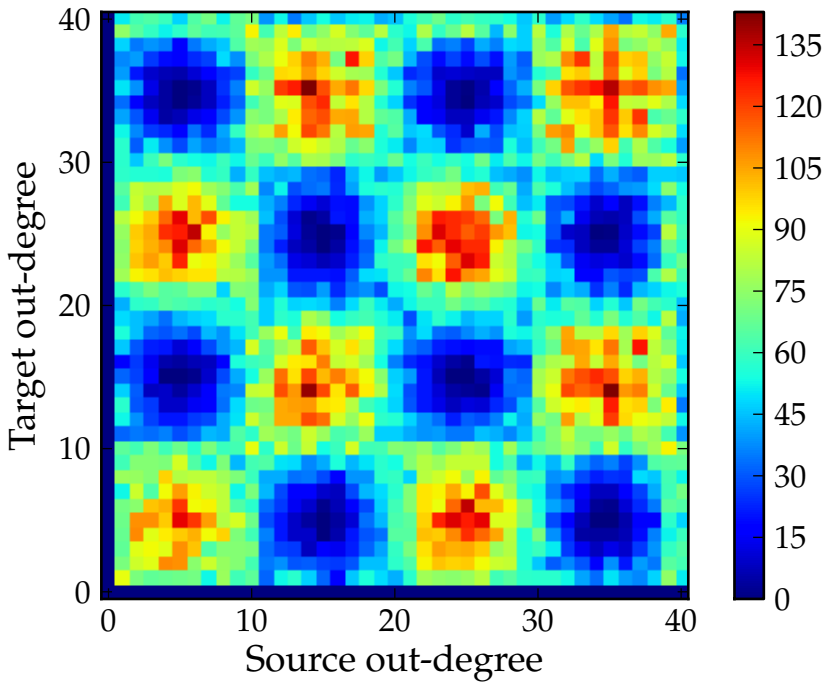


Figure 3.17: Out/out-degree correlation histogram.

graph_tool.correlations.**combined_corr_hist**(*g*, *deg1*, *deg2*, *bins=[[0, 1], [0, 1]]*, *float_count=True*)

Obtain the single-vertex combined correlation histogram for the given graph.

Parameters **g** : Graph (page 28)

Graph to be used.

deg1 : string or PropertyMap (page 35)

first degree type ("in", "out" or "total") or vertex property map.

deg2 : string or PropertyMap (page 35)

second degree type ("in", "out" or "total") or vertex property map.

bins : list of lists (optional, default: [[0, 1], [0, 1]])

A list of bin edges to be used for the first and second degrees. If any list has size 2, it is used to create an automatically generated bin range starting from the first value, and with constant bin width given by the second value.

float_count : bool (optional, default: True)

If True, the bin counts are converted float variables, which is useful for normalization, and other processing. It False, the bin counts will be unsigned integers.

Returns **bin_counts** : ndarray

Two-dimensional array with the bin counts.

first_bins : ndarray

First degree bins

second_bins : ndarray

Second degree bins

See Also:

assortativity (page 93) assortativity coefficient

scalar_assortativity (page 94) scalar assortativity coefficient

corr_hist (page 95) vertex-vertex correlation histogram

combined_corr_hist (page 97) combined single-vertex correlation histogram

avg_neighbour_corr (page 98) average nearest-neighbour correlation

avg_combined_corr (page 100) average combined single-vertex correlation

Notes

If enabled during compilation, this algorithm runs in parallel.

Examples

```
>>> def sample_k(max):
...     accept = False
...     while not accept:
...         i = randint(1, max + 1)
...         j = randint(1, max + 1)
...         accept = random() < (sin(i / pi) * sin(j / pi) + 1) / 2
...     return i,j
...
>>> g = gt.random_graph(10000, lambda: sample_k(40))
>>> h = gt.combined_corr_hist(g, "in", "out")
>>> clf()
>>> xlabel("In-degree")
<...>
>>> ylabel("Out-degree")
<...>
>>> imshow(h[0].T, interpolation="nearest", origin="lower")
<...>
>>> colorbar()
<...>
>>> savefig("combined_corr.pdf")
```

graph_tool.correlations.**avg_neighbour_corr**(*g*, *deg_source*, *deg_target*, *bins=[0, 1]*, *weight=None*)

Obtain the average neighbour-neighbour correlation for the given graph.

Parameters **g** : Graph (page 28)

Graph to be used.

deg_source : string or PropertyMap (page 35)

degree type ("in", "out" or "total") or vertex property map for the source vertex.

deg_target : string or PropertyMap (page 35)

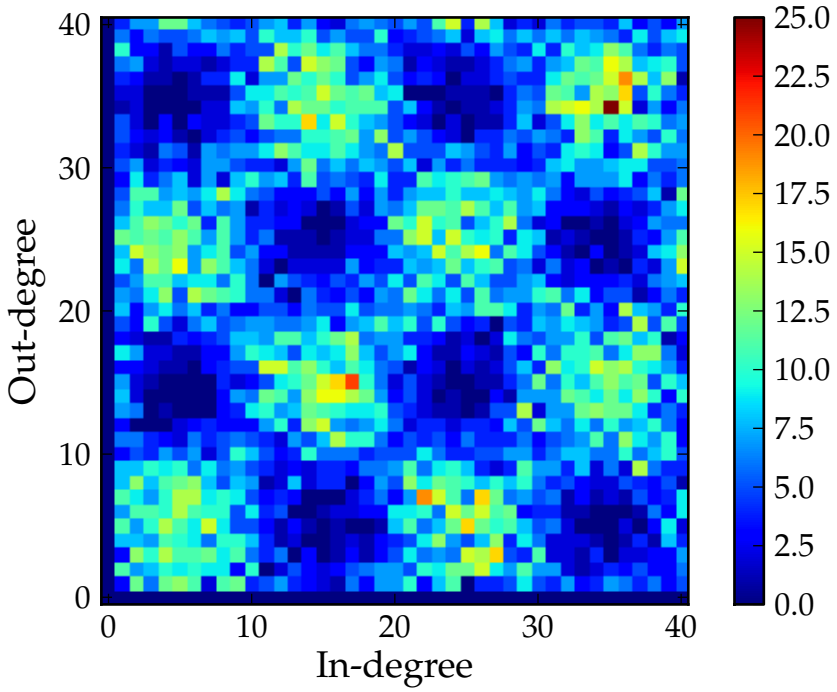


Figure 3.18: Combined in/out-degree correlation histogram.

degree type ("in", "out" or "total") or vertex property map for the target vertex.

bins : list (optional, default: [0, 1])

Bins to be used for the source degrees. If the list has size 2, it is used as the constant width of an automatically generated bin range, starting from the first value.

weight : edge property map (optional, default: None)

Weight (multiplicative factor) to be used on each edge.

Returns **bin_avg** : `ndarray`

Array with the deg_target average for the get_source bins.

bin_dev : `ndarray`

Array with the standard deviation of the deg_target average for the get_source bins.

bins : `ndarray`

Source degree bins,

See Also:

`assortativity` **(page 93)** assortativity coefficient

`scalar_assortativity` **(page 94)** scalar assortativity coefficient

`corr_hist` **(page 95)** vertex-vertex correlation histogram

`combined_corr_hist` **(page 97)** combined single-vertex correlation histogram

`avg_neighbour_corr` **(page 98)** average nearest-neighbour correlation

`avg_combined_corr` **(page 100)** average combined single-vertex correlation

Notes

The average correlation is the average, for every vertex with degree (or scalar property) 'source_deg', the of the 'target_deg' degree (or scalar property) of its neighbours.

If enabled during compilation, this algorithm runs in parallel.

Examples

```
>>> def sample_k(max):
...     accept = False
...     while not accept:
...         k = randint(1,max+1)
...         accept = random() < 1.0 / k
...     return k
...
>>> g = gt.random_graph(10000, lambda: sample_k(40), model="probabilistic",
...                     vertex_corr=lambda i, j: (sin(i / pi) * sin(j / pi) + 1) / 2,
...                     directed=False, n_iter=100)
>>> h = gt.avg_neighbour_corr(g, "out", "out")
>>> clf()
>>> xlabel("Source out-degree")
<...>
>>> ylabel("Target out-degree")
<...>
>>> errorbar(h[2][:-1], h[0], yerr=h[1], fmt="o")
<...>
>>> savefig("avg_corr.pdf")
```

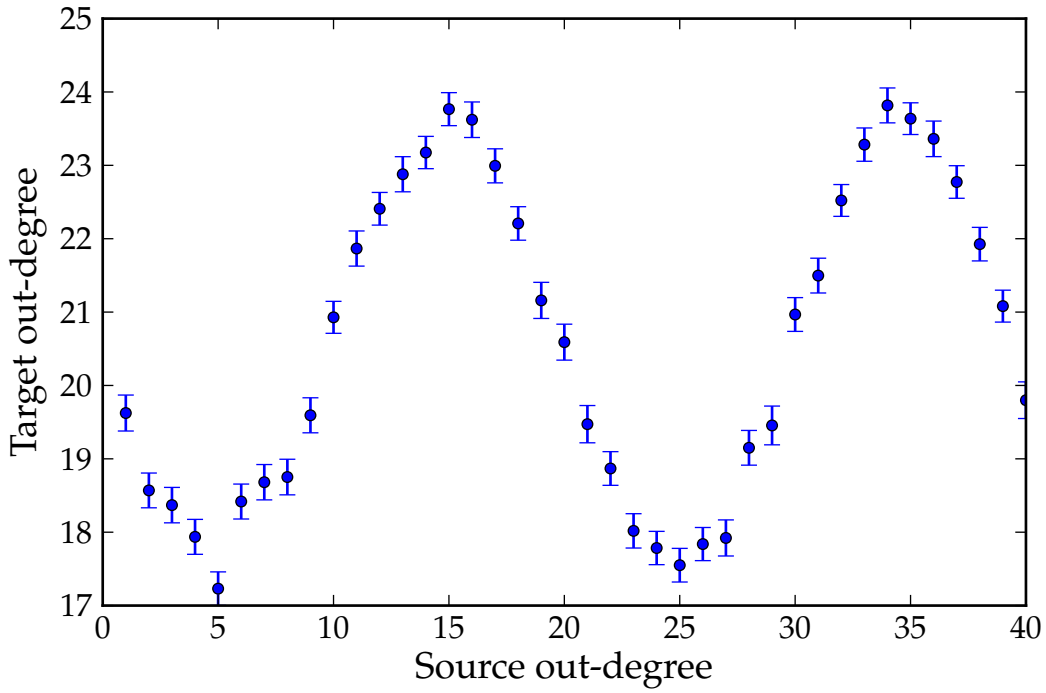


Figure 3.19: Average out/out degree correlation.

graph_tool.correlations.**avg_combined_corr**(*g, deg1, deg2, bins=[0, 1]*)

Obtain the single-vertex combined correlation histogram for the given graph.

Parameters **g** : Graph (page 28)

Graph to be used.

deg1 : string or PropertyMap (page 35)

first degree type ("in", "out" or "total") or vertex property map.

deg2 : string or PropertyMap (page 35)

second degree type ("in", "out" or "total") or vertex property map.

bins : list (optional, default: [0, 1])

Bins to be used for the first degrees. If the list has size 2, it is used as the constant width of an automatically generated bin range, starting from the first value.

Returns **bin_avg** : ndarray

Array with the deg2 average for the deg1 bins.

bin_dev : ndarray

Array with the standard deviation of the deg2 average for the deg1 bins.

bins : ndarray

The deg1 bins.

See Also:

assortativity (page 93) assortativity coefficient

scalar_assortativity (page 94) scalar assortativity coefficient

corr_hist (page 95) vertex-vertex correlation histogram

combined_corr_hist (page 97) combined single-vertex correlation histogram

avg_neighbour_corr (page 98) average nearest-neighbour correlation

avg_combined_corr (page 100) average combined single-vertex correlation

Notes

If enabled during compilation, this algorithm runs in parallel.

Examples

```
>>> def sample_k(max):
...     accept = False
...     while not accept:
...         i = randint(1,max+1)
...         j = randint(1,max+1)
...         accept = random() < (sin(i/pi)*sin(j/pi)+1)/2
...     return i,j
...
>>> g = gt.random_graph(10000, lambda: sample_k(40))
>>> h = gt.avg_combined_corr(g, "in", "out")
>>> clf()
>>> xlabel("In-degree")
<...>
```

```
>>> ylabel("Out-degree")
<...>
>>> errorbar(h[2][:-1], h[0], yerr=h[1], fmt="o")
<...>
>>> savefig("combined_avg_corr.pdf")
```

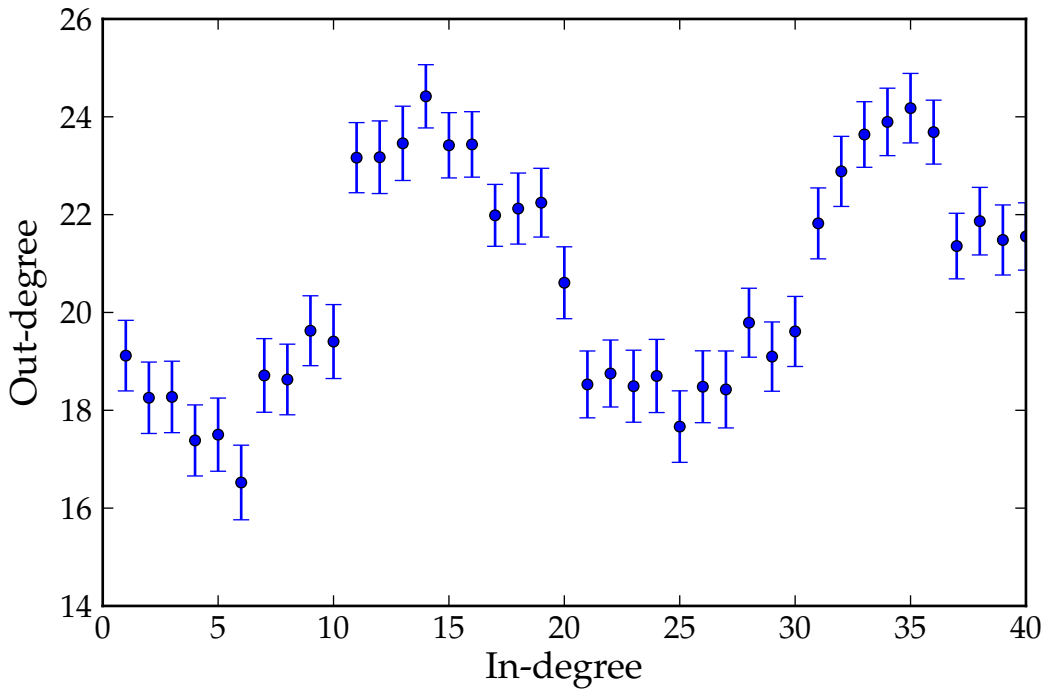


Figure 3.20: Average combined in/out-degree correlation.

3.2.6 `graph_tool.draw` - Graph drawing and layout

Summary

Layout algorithms

`sfdp_layout` (page 103)	Obtain the SFDP spring-block layout of the graph.
`fruchterman_reingold_layout` (page 105)	Calculate the Fruchterman-Reingold spring-block layout of
`arf_layout` (page 107)	Calculate the ARF spring-block layout of the graph.
`random_layout` (page 109)	Performs a random layout of the graph.

Graph drawing

`graph_draw` (page 111)	Draw a graph to screen or to a file using `cairo`.
`graphviz_draw` (page 116)	Draw a graph using graphviz.
`prop_to_size` (page 120)	Convert property map values to be more useful as a vertex size, or edge width

`cairo_draw` (page 121)	Draw a graph to a `cairo` context.

Continued on next page

Table 3.9 – continued from previous page

interactive_window (page 122)	Display an interactive GTK+ window containing the given graph.
GraphWidget (page 123)	Interactive GTK+ widget displaying a given graph.
GraphWindow (page 125)	Interactive GTK+ window containing a GraphWidget (page 123).

Low-level graph drawing

Contents

Layout algorithms

graph_tool.draw.**sfdp_layout**(*g*, *vweight=None*, *eweight=None*, *pin=None*, *groups=None*, *C=0.2*, *K=None*, *p=2.0*, *theta=0.6*, *max_level=11*, *gamma=1.0*, *mu=0.0*, *mu_p=1.0*, *init_step=None*, *cooling_step=0.9*, *adaptive_cooling=True*, *epsilon=0.1*, *max_iter=0*, *pos=None*, *multilevel=None*, *coarse_method='hybrid'*, *mivs_thres=0.9*, *ec_thres=0.75*, *weighted_coarse=False*, *verbose=False*)

Obtain the SFDP spring-block layout of the graph.

Parameters **g** : Graph (page 28)

Graph to be used.

vweight : PropertyMap (page 35) (optional, default: None)

A vertex property map with the respective weights.

eweight : PropertyMap (page 35) (optional, default: None)

An edge property map with the respective weights.

pin : PropertyMap (page 35) (optional, default: None)

A vertex property map with boolean values, which, if given, specify the vertices which will not have their positions modified.

groups : PropertyMap (page 35) (optional, default: None)

A vertex property map with group assignments. Vertices belonging to the same group will be put close together.

C : float (optional, default: 0.2)

Relative strength of repulsive forces.

K : float (optional, default: None)

Optimal edge length. If not provided, it will be taken to be the average edge distance in the initial layout.

p : float (optional, default: 2)

Repulsive force exponent.

theta : float (optional, default: 0.6)

Quadtree opening parameter, a.k.a. Barnes-Hut opening criterion.

max_level : int (optional, default: 11)

Maximum quadtree level.

gamma : float (optional, default: 1.0)

Strength of the attractive force between connected components, or group assignments.

mu : float (optional, default: `0.0`)

Strength of the attractive force between vertices of the same connected component, or group assignment.

mu_p : float (optional, default: `1.0`)

Scaling exponent of the attractive force between vertices of the same connected component, or group assignment.

init_step : float (optional, default: `None`)

Initial update step. If not provided, it will be chosen automatically.

cooling_step : float (optional, default: `0.9`)

Cooling update step.

adaptive_cooling : bool (optional, default: `True`)

Use an adaptive cooling scheme.

epsilon : float (optional, default: `0.1`)

Relative convergence criterion.

max_iter : int (optional, default: `0`)

Maximum number of iterations. If this value is `0`, it runs until convergence.

pos : `PropertyMap` (page 35) (optional, default: `None`)

Initial vertex layout. If not provided, it will be randomly chosen.

multilevel : bool (optional, default: `None`)

Use a multilevel layout algorithm. If `None` is given, it will be activated based on the size of the graph.

coarse_method : str (optional, default: `"hybrid"`)

Coarsening method used if `multilevel == True`. Allowed methods are `"hybrid"`, `"mivs"` and `"ec"`.

mivs_thres : float (optional, default: `0.9`)

If the relative size of the MIVS coarse graph is above this value, the coarsening stops.

ec_thres : float (optional, default: `0.75`)

If the relative size of the EC coarse graph is above this value, the coarsening stops.

weighted_coarse : bool (optional, default: `False`)

Use weighted coarse graphs.

verbose : bool (optional, default: `False`)

Provide verbose information.

Returns **pos** : `PropertyMap` (page 35)

A vector-valued vertex property map with the coordinates of the vertices.

Notes

This algorithm is defined in [hu-multilevel-2005] (page 229), and has complexity $O(V \log V)$.

References

[hu-multilevel-2005] (page 229)

Examples

```
>>> g = gt.price_network(3000)
>>> pos = gt.sfdp_layout(g)
>>> gt.graph_draw(g, pos=pos, output="graph-draw-sfdp.pdf")
<...>
```

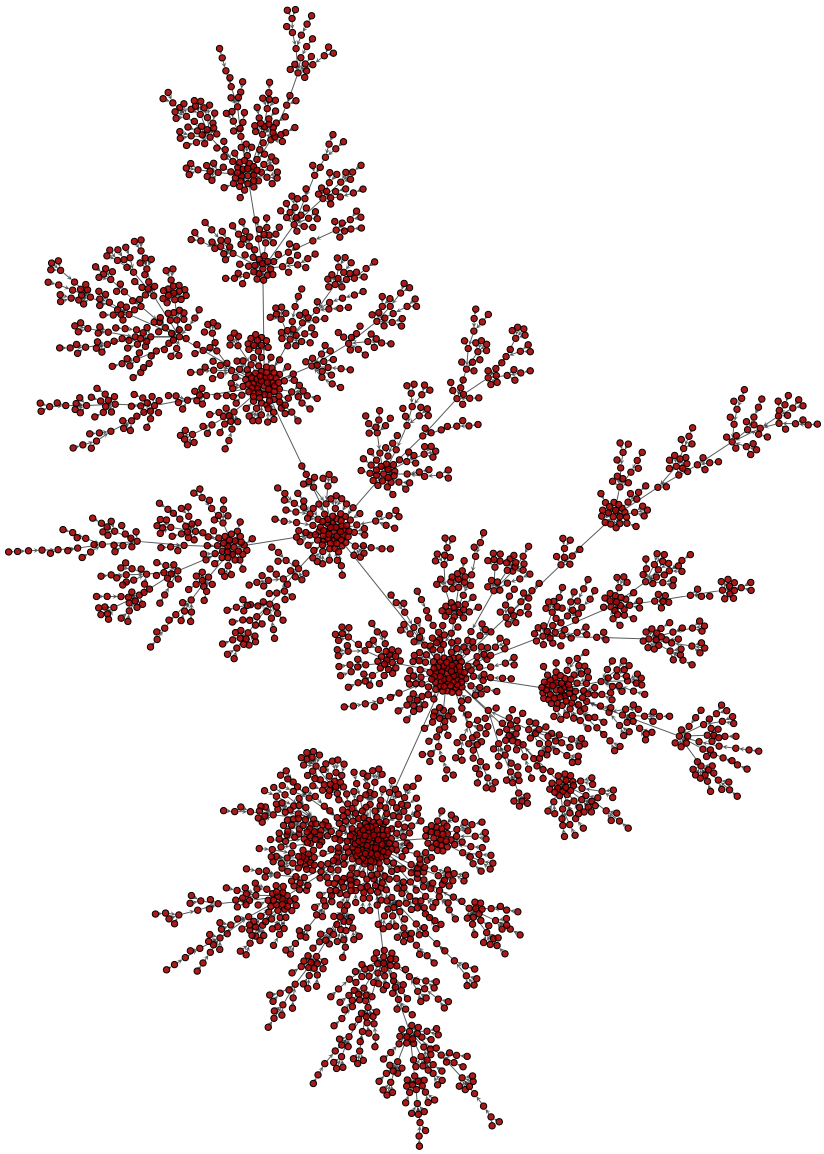


Figure 3.21: SFDP layout of a Price network.

graph_tool.draw.**fruchterman_reingold_layout**(*g*, *weight=None*, *a=None*, *r=1.0*, *scale=None*, *circular=False*, *grid=True*, *t_range=None*, *n_iter=100*, *pos=None*)

Calculate the Fruchterman-Reingold spring-block layout of the graph.

Parameters **g** : Graph (page 28)

Graph to be used.

weight : PropertyMap (optional, default: None)

An edge property map with the respective weights.

a : float (optional, default: V)

Attracting force between adjacent vertices.

r : float (optional, default: 1.0)

Repulsive force between vertices.

scale : float (optional, default: $\sqrt{V}$)

Total scale of the layout (either square side or radius).

circular : bool (optional, default: False)

If True, the layout will have a circular shape. Otherwise the shape will be a square.

grid : bool (optional, default: True)

If True, the repulsive forces will only act on vertices which are on the same site on a grid. Otherwise they will act on all vertex pairs.

t_range : tuple of floats (optional, default: (scale / 10, scale / 1000))

Temperature range used in annealing. The temperature limits the displacement at each iteration.

n_iter : int (optional, default: 100)

Total number of iterations.

pos : PropertyMap (optional, default: None)

Vector vertex property maps where the coordinates should be stored. If provided, this will also be used as the initial position of the vertices.

Returns **pos** : PropertyMap (page 35)

A vector-valued vertex property map with the coordinates of the vertices.

Notes

This algorithm is defined in [fruchterman-reingold] (page 229), and has complexity $O(\text{n-iter} \times V^2)$ if *grid=False* or $O(\text{n-iter} \times (V + E))$ otherwise.

References

[fruchterman-reingold] (page 229)

Examples

```
>>> g = gt.price_network(300)
>>> pos = gt.fruchterman_reingold_layout(g, n_iter=1000)
>>> gt.graph_draw(g, pos=pos, output="graph-draw-fr.pdf")
<...>
```

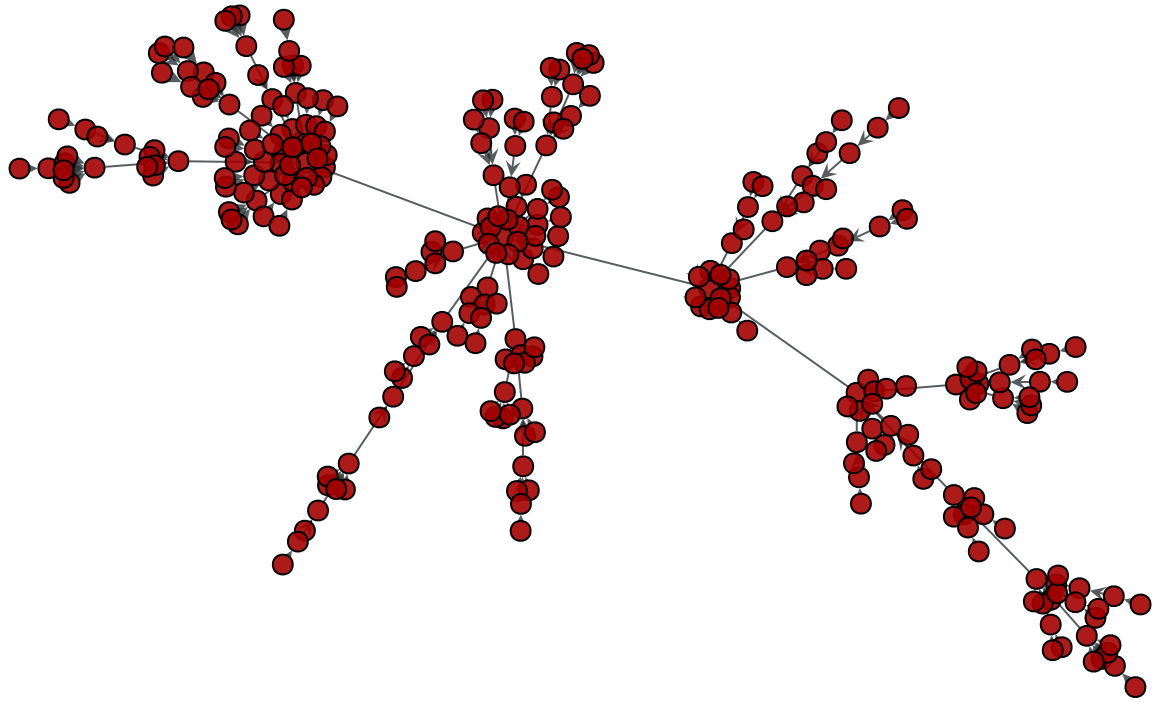


Figure 3.22: Fruchterman-Reingold layout of a Price network.

graph_tool.draw.**arf_layout**(*g*, *weight=None*, *d=0.5*, *a=10*, *dt=0.001*, *epsilon=1e-06*, *max_iter=1000*, *pos=None*, *dim=2*)

Calculate the ARF spring-block layout of the graph.

Parameters **g** : Graph (page 28)

Graph to be used.

weight : PropertyMap (page 35) (optional, default: None)

An edge property map with the respective weights.

d : float (optional, default: 0.5)

Opposing force between vertices.

a : float (optional, default: `10`)

Attracting force between adjacent vertices.

dt : float (optional, default: `0.001`)

Iteration step size.

epsilon : float (optional, default: `1e-6`)

Convergence criterion.

max_iter : int (optional, default: `1000`)

Maximum number of iterations. If this value is `0`, it runs until convergence.

pos : `PropertyMap` (page 35) (optional, default: `None`)

Vector vertex property maps where the coordinates should be stored.

dim : int (optional, default: `2`)

Number of coordinates per vertex.

Returns **pos** : `PropertyMap` (page 35)

A vector-valued vertex property map with the coordinates of the vertices.

Notes

This algorithm is defined in [geipel-self-organization-2007] (page 229), and has complexity $O(V^2)$.

References

[geipel-self-organization-2007] (page 229)

Examples

```
>>> g = gt.price_network(300)
>>> pos = gt.arf_layout(g, max_iter=0)
>>> gt.graph_draw(g, pos=pos, output="graph-draw-arf.pdf")
<...>
```

graph_tool.draw.**random_layout**(*g*, *shape=None*, *pos=None*, *dim=2*)

Performs a random layout of the graph.

Parameters **g** : `Graph` (page 28)

Graph to be used.

shape : tuple or list (optional, default: `None`)

Rectangular shape of the bounding area. The size of this parameter must match *dim*, and each element can be either a pair specifying a range, or a single value specifying a range starting from zero. If None is passed, a square of linear size $\sqrt{N}$ is used.

pos : `PropertyMap` (page 35) (optional, default: `None`)

Vector vertex property maps where the coordinates should be stored.

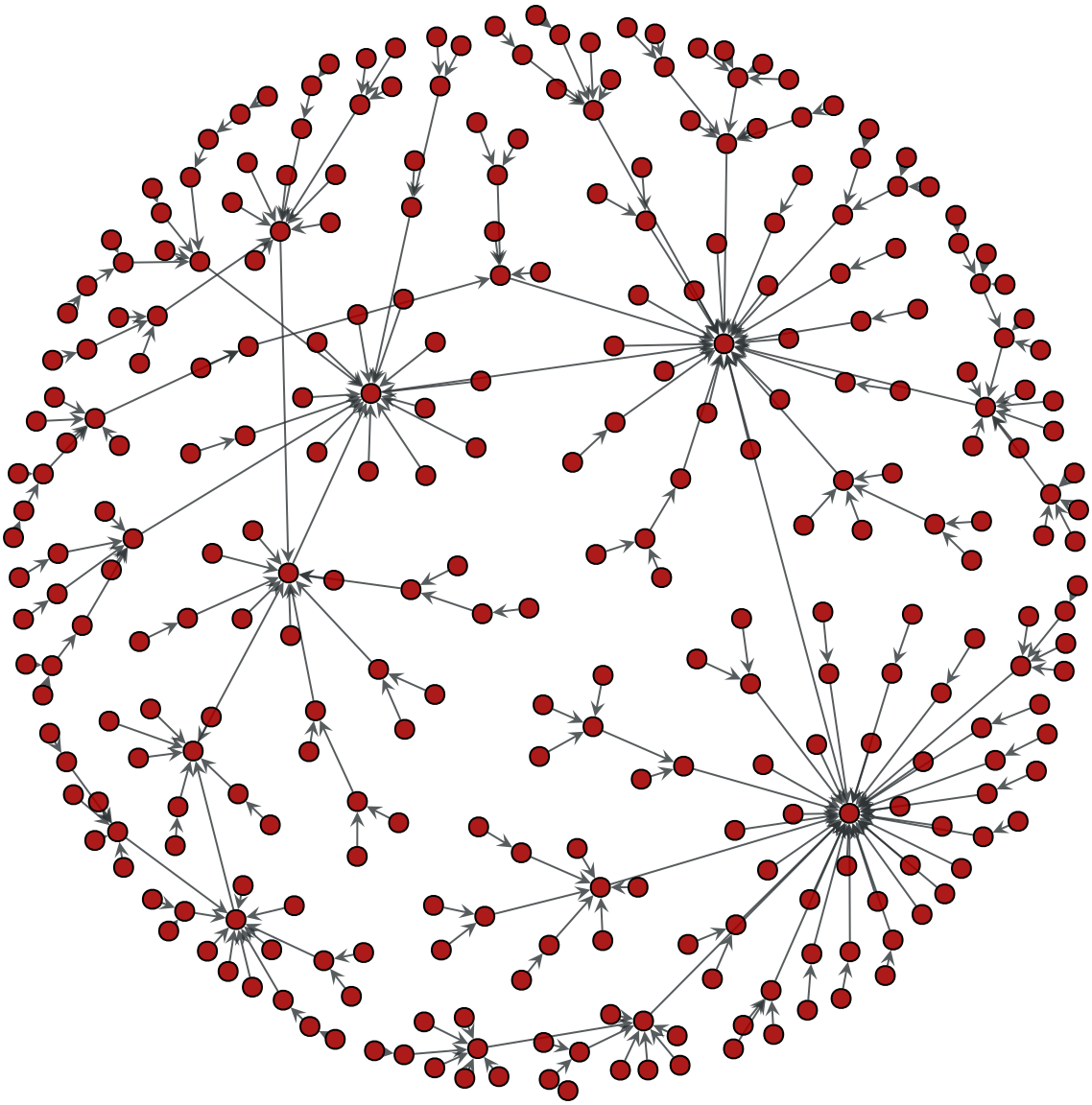


Figure 3.23: ARF layout of a Price network.

dim : int (optional, default: `2`)

Number of coordinates per vertex.

Returns **pos** : `PropertyMap` (page 35)

A vector-valued vertex property map with the coordinates of the vertices.

Notes

This algorithm has complexity $O(V)$.

Examples

```
>>> g = gt.random_graph(100, lambda: (3, 3))
>>> shape = [[50, 100], [1, 2], 4]
>>> pos = gt.random_layout(g, shape=shape, dim=3)
>>> pos[g.vertex(0)].a
array([ 68.72700594,   1.03142919,   2.56812658])
```

Graph drawing

graph_tool.draw.**graph_draw**(*g*, *pos=None*, *vprops=None*, *eprops=None*, *vorder=None*, *eorder=None*, *nodesfirst=False*, *output_size=(600, 600)*, *fit_view=True*, *output=None*, *fmt='auto'*, ***kwargs*)

Draw a graph to screen or to a file using `cairo`.

Parameters **g** : `Graph` (page 28)

Graph to be drawn.

pos : `PropertyMap` (page 35) (optional, default: `None`)

Vector-valued vertex property map containing the x and y coordinates of the vertices. If not given, it will be computed using `sfdp_layout()` (page 103).

vprops : dict (optional, default: `None`)

Dictionary with the vertex properties. Individual properties may also be given via the `vertex_<prop-name>` parameters, where `<prop-name>` is the name of the property.

eprops : dict (optional, default: `None`)

Dictionary with the vertex properties. Individual properties may also be given via the `edge_<prop-name>` parameters, where `<prop-name>` is the name of the property.

vorder : `PropertyMap` (page 35) (optional, default: `None`)

If provided, defines the relative order in which the vertices are drawn.

eorder : `PropertyMap` (page 35) (optional, default: `None`)

If provided, defines the relative order in which the edges are drawn.

nodesfirst : bool (optional, default: `False`)

If `True`, the vertices are drawn first, otherwise the edges are.

output_size : tuple of scalars (optional, default: `(600,600)`)

Size of the drawing canvas. The units will depend on the output format (pixels for the screen, points for PDF, etc).

fit_view : bool (optional, default: `True`)

If `True`, the layout will be scaled to fit the entire display area.

output : string (optional, default: `None`)

Output file name. If not given, the graph will be displayed via `interactive_window()` (page 122).

fmt : string (default: `"auto"`)

Output file format. Possible values are `"auto"`, `"ps"`, `"pdf"`, `"svg"`, and `"png"`. If the value is `"auto"`, the format is guessed from the `output` parameter.

vertex_* : `PropertyMap` (page 35) or arbitrary types (optional, default: `None`)

Parameters following the pattern `vertex_<prop-name>` specify the vertex property with name `<prop-name>`, as an alternative to the `vprops` parameter.

edge_* : `PropertyMap` (page 35) or arbitrary types (optional, default: `None`)

Parameters following the pattern `edge_<prop-name>` specify the edge property with name `<prop-name>`, as an alternative to the `eprops` parameter.

****kwargs** :

Any extra parameters are passed to `interactive_window()` (page 122), `GraphWindow` (page 125), `GraphWidget` (page 123) and `cairo_draw()` (page 121).

Returns **pos** : `PropertyMap` (page 35)

Vector vertex property map with the x and y coordinates of the vertices.

selected : `PropertyMap` (page 35) (optional, only if `output is None`)

Boolean-valued vertex property map marking the vertices which were selected interactively.

Notes

Table 3.10: List of vertex properties

Name	Description	Accepted types	Default Value
shape	The vertex shape. Can be one of the following strings: “circle”, “triangle”, “square”, “pentagon”, “hexagon”, “heptagon”, “octagon” “double_circle”, “double_triangle”, “double_square”, “double_pentagon”, “double_hexagon”, “double_heptagon”, “double_octagon”, “pie”. Optionally, this might take a numeric value corresponding to position in the list above.	`str` or `int`	`"circle"`
color	Color used to stroke the lines of the vertex.	`str` or list of `floats`	`[0., 0., 0., 1]`
fill_color	Color used to fill the interior of the vertex.	`str` or list of `floats`	`[0.640625, 0, 0, 0.9]`
size	The size of the vertex, in the default units of the output format (normally either pixels or points).	`float` or `int`	`5`
aspect	The aspect ratio of the vertex.	`float` or `int`	`1.0`
anchor	Specifies how the edges anchor to the vertices. If *0*, the anchor is at the center of the vertex, otherwise it is at the border.	`int`	`1`
pen_width	Width of the lines used to draw the vertex, in the default units of the output format (normally either pixels or points).	`float` or `int`	`0.8`
halo	Whether to draw a circular halo around the vertex.	`bool`	`False`
halo_color	Color used to draw the halo.	`str` or list of `floats`	`[0., 0., 1., 0.5]`
halo_size	Relative size of the halo.	`float`	`1.5`
text	Text to draw together with the vertex.	`str`	`""`
text_color	Color used to draw the text. If the value is `"auto"`, it will be computed based on fill_color to maximize contrast.	`str` or list of `floats`	`"auto"`
text_position	Position of the text relative to the vertex. If the passed value is positive, it will correspond to an angle in radians, which will determine where the text will be placed outside the vertex. If the value is negative, the text will be placed inside the vertex. If the value is `-1`, the vertex size will be automatically increased to accommodate the text.	`float` or `int`	`-1`
font_family	Font family used to draw the text.	`str`	`"serif"`
font_slant	Font slant used to draw the text.	`cairo.FONT_SLANT_*`	`cairo.FONT_SLANT_NORMAL`
font_weight	Font weight used to draw the text.	`cairo.FONT_WEIGHT_*`	`cairo.FONT_WEIGHT_NORMAL`
font_size	Font size used to draw the text.	`float` or `int`	`12`
surface	The cairo surface used to draw the vertex. If the value passed is a string, it is interpreted as an image file name to be loaded.	`cairo.Surface` or `str`	`None`
pie_fractions	Fractions of the pie sections for the vertices if `shape=="pie"`.	list of `int` or `float`	`[0.75, 0.25]`
pie_colors	Colors used in the pie sections if `shape=="pie"`.	list of strings or `float`.	`('b','g','r','c','m','y'`

Table 3.11: List of edge properties

Name	Description	Accepted types	Default Value
color	Color used to stroke the edge lines.	`str` or list of floats	`[0.179, 0.203,0.210, 0.8]`
pen_width	Width of the line used to draw the edge, in the default units of the output format (normally either pixels or points).	`float` or `int`	`1.0`
start_marker, mid_marker, end_marker	Edge markers. Can be one of "none", "arrow", "circle", "square", "diamond", or "bar". Optionally, this might take a numeric value corresponding to position in the list above.	`str` or `int`	`-1`
marker_size	Size of edge markers, in units appropriate to the output format (normally either pixels or points).	`float` or `int`	`4`
control_points	Control points of a Bézier spline used to draw the edge.	sequence of `floats`	`[]`
dash_style	Dash pattern is specified by an array of positive values. Each value provides the length of alternate "on" and "off" portions of the stroke. The last value specifies an offset into the pattern at which the stroke begins.	sequence of `floats`	`[]`
text	Text to draw next to the edges.	`str`	`""`
text_color	Color used to draw the text.	`str` or list of `floats`	`[0., 0., 0., 1.]`
text_distance	Distance from the edge and its text.	`float` or `int`	`4`
text_parallel	If `True` the text will be drawn parallel to the edges.	`bool`	`True`
font_family	Font family used to draw the text.	`str`	`"serif"`
font_slant	Font slant used to draw the text.	`cairo.FONT_SLANT_NORMAL`	`cairo.FONT_SLANT_NORMA`
font_weight	Font weight used to draw the text.	`cairo.FONT_WEIGHT_NORMAL`	`cairo.FONT_WEIGHT_NORM`
font_size	Font size used to draw the text.	`float` or `int`	`12`

Examples

```
>>> g = gt.price_network(1500)
>>> deg = g.degree_property_map("in")
>>> deg.a = 4 * (sqrt(deg.a) * 0.5 + 0.4)
>>> ebet = gt.betweenness(g)[1]
>>> ebet.a /= ebet.a.max() / 10.
>>> eorder = ebet.copy()
>>> eorder.a *= -1
>>> pos = gt.sfdp_layout(g)
>>> control = g.new_edge_property("vector<double>")
>>> for e in g.edges():
...     d = sqrt(sum((pos[e.source()].a - pos[e.target()].a) ** 2)) / 5
...     control[e] = [0.3, d, 0.7, d]
>>> gt.graph_draw(g, pos=pos, vertex_size=deg, vertex_fill_color=deg, vorder=deg,
...               edge_color=ebet, eorder=eorder, edge_pen_width=ebet,
```

```
...                 edge_control_points=control, # some curvy edges
...                 output="graph-draw.pdf")
<...>
```

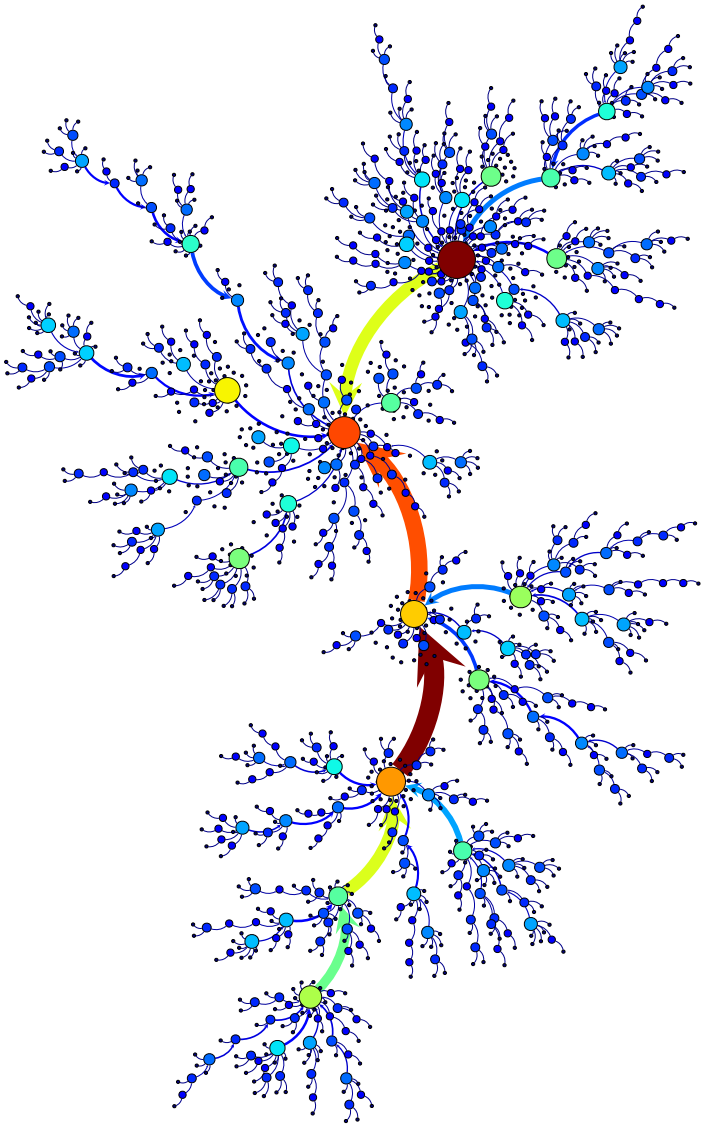


Figure 3.24: SFDP force-directed layout of a Price network with 1500 nodes. The vertex size and color indicate the degree, and the edge color and width the edge betweeness centrality.

graph_tool.draw.**graphviz_draw**(*g*, *pos=None*, *size=(15, 15)*, *pin=False*, *layout=None*, *maxiter=None*, *ratio='fill'*, *overlap=True*, *sep=None*, *splines=False*, *vsize=0.105*, *penwidth=1.0*, *elen=None*, *gprops={}*, *vprops={}*, *eprops={}*, *vcolor='#a40000'*, *ecolor='#2e3436'*, *vcmap=None*, *vnorm=True*, *ecmap=None*, *enorm=True*, *vorder=None*, *eorder=None*, *output=''*, *output_format='auto'*, *fork=False*, *return_string=False*)

Draw a graph using graphviz.

Parameters **g** : Graph (page 28)

Graph to be drawn.

pos : `PropertyMap` (page 35) or tuple of `PropertyMap` (page 35) (optional, default: `None`)

Vertex property maps containing the x and y coordinates of the vertices.

size : tuple of scalars (optional, default: `(15,15)`)

Size (in centimeters) of the canvas.

pin : bool or `PropertyMap` (page 35) (default: `False`)

If `True`, the vertices are not moved from their initial position. If a `PropertyMap` (page 35) is passed, it is used to pin nodes individually.

layout : string (default: `"neato" if g.num_vertices() <= 1000 else "sfdp"`)

Layout engine to be used. Possible values are `"neato"`, `"fdp"`, `"dot"`, `"circo"`, `"twopi"` and `"arf"`.

maxiter : int (default: `None`)

If specified, limits the maximum number of iterations.

ratio : string or float (default: `"fill"`)

Sets the aspect ratio (drawing height/drawing width) for the drawing. Note that this is adjusted before the `size` attribute constraints are enforced.

If `ratio` is numeric, it is taken as the desired aspect ratio. Then, if the actual aspect ratio is less than the desired ratio, the drawing height is scaled up to achieve the desired ratio; if the actual ratio is greater than that desired ratio, the drawing width is scaled up.

If `ratio == "fill"` and the size attribute is set, node positions are scaled, separately in both x and y, so that the final drawing exactly fills the specified size.

If `ratio == "compress"` and the size attribute is set, dot attempts to compress the initial layout to fit in the given size. This achieves a tighter packing of nodes but reduces the balance and symmetry. This feature only works in dot.

If `ratio == "expand"`, the size attribute is set, and both the width and the height of the graph are less than the value in size, node positions are scaled uniformly until at least one dimension fits size exactly. Note that this is distinct from using size as the desired size, as here the drawing is expanded before edges are generated and all node and text sizes remain unchanged.

If `ratio == "auto"`, the page attribute is set and the graph cannot be drawn on a single page, then size is set to an “ideal” value. In particular, the size in a given dimension will be the smallest integral multiple of the page size in that dimension which is at least half the current size. The two dimensions are then scaled independently to the new size. This feature only works in dot.

overlap : bool or string (default: `"prism"`)

Determines if and how node overlaps should be removed. Nodes are first enlarged using the sep attribute. If `True`, overlaps are retained. If the value is `"scale"`, overlaps are removed by uniformly scaling in x and y. If the value is `False`, node overlaps are removed by a

Voronoi-based technique. If the value is `"scalexy"`, x and y are separately scaled to remove overlaps.

If sfdp is available, one can set overlap to `"prism"` to use a proximity graph-based algorithm for overlap removal. This is the preferred technique, though `"scale"` and `False` can work well with small graphs. This technique starts with a small scaling up, controlled by the overlap_scaling attribute, which can remove a significant portion of the overlap. The prism option also accepts an optional nonnegative integer suffix. This can be used to control the number of attempts made at overlap removal. By default, `overlap == "prism"` is equivalent to `overlap == "prism1000"`. Setting `overlap == "prism0"` causes only the scaling phase to be run.

If the value is `"compress"`, the layout will be scaled down as much as possible without introducing any overlaps, obviously assuming there are none to begin with.

sep : float (default: `None`)

Specifies margin to leave around nodes when removing node overlap. This guarantees a minimal non-zero distance between nodes.

splines : bool (default: `False`)

If `True`, the edges are drawn as splines and routed around the vertices.

vsize : float, `PropertyMap` (page 35), or tuple (default: `0.105`)

Default vertex size (width and height). If a tuple is specified, the first value should be a property map, and the second is a scale factor.

penwidth : float, `PropertyMap` (page 35) or tuple (default: `1.0`)

Specifies the width of the pen, in points, used to draw lines and curves, including the boundaries of edges and clusters. It has no effect on text. If a tuple is specified, the first value should be a property map, and the second is a scale factor.

elen : float or `PropertyMap` (page 35) (default: `None`)

Preferred edge length, in inches.

gprops : dict (default: `{}`)

Additional graph properties, as a dictionary. The keys are the property names, and the values must be convertible to string.

vprops : dict (default: `{}`)

Additional vertex properties, as a dictionary. The keys are the property names, and the values must be convertible to string, or vertex property maps, with values convertible to strings.

eprops : dict (default: `{}`)

Additional edge properties, as a dictionary. The keys are the property names, and the values must be convertible to string, or edge property maps, with values convertible to strings.

vcolor : string or `PropertyMap` (page 35) (default: `"#a40000"`)

Drawing color for vertices. If the valued supplied is a property map, the values must be scalar types, whose color values are obtained from the `vcmap` argument.

ecolor : string or `PropertyMap` (page 35) (default: `"#2e3436"`)

Drawing color for edges. If the valued supplied is a property map, the values must be scalar types, whose color values are obtained from the `ecmap` argument.

vcmap : `matplotlib.colors.Colormap` (default: `matplotlib.cm.jet`)

Vertex color map.

vnorm : bool (default: `True`)

Normalize vertex color values to the [0,1] range.

ecmap : `matplotlib.colors.Colormap` (default: `matplotlib.cm.jet`)

Edge color map.

enorm : bool (default: `True`)

Normalize edge color values to the [0,1] range.

vorder : `PropertyMap` (page 35) (default: `None`)

Scalar vertex property map which specifies the order with which vertices are drawn.

eorder : `PropertyMap` (page 35) (default: `None`)

Scalar edge property map which specifies the order with which edges are drawn.

output : string (default: `""`)

Output file name.

output_format : string (default: `"auto"`)

Output file format. Possible values are `"auto"`, `"xlib"`, `"ps"`, `"svg"`, `"svgz"`, `"fig"`, `"mif"`, `"hpgl"`, `"pcl"`, `"png"`, `"gif"`, `"dia"`, `"imap"`, `"cmapx"`. If the value is `"auto"`, the format is guessed from the `output` parameter, or `xlib` if it is empty. If the value is `None`, no output is produced.

fork : bool (default: `False`)

If `True`, the program is forked before drawing. This is used as a work-around for a bug in graphviz, where the `exit()` function is called, which would cause the calling program to end. This is always assumed `True`, if `output_format == 'xlib'`.

return_string : bool (default: `False`)

If `True`, a string containing the rendered graph as binary data is returned (defaults to png format).

Returns **pos** : `PropertyMap` (page 35)

Vector vertex property map with the x and y coordinates of the vertices.

gv : gv.digraph or gv.graph (optional, only if `returngv == True`)

Internally used graphviz graph.

Notes

This function is a wrapper for the [graphviz] (page 229) routines. Extensive additional documentation for the graph, vertex and edge properties is available at: http://www.graphviz.org/doc/info/attrs.html.

References

[graphviz] (page 229)

Examples

```
>>> g = gt.price_network(1500)
>>> deg = g.degree_property_map("in")
>>> deg.a = 2 * (sqrt(deg.a) * 0.5 + 0.4)
>>> ebet = gt.betweenness(g)[1]
>>> gt.graphviz_draw(g, vcolor=deg, vorder=deg, elen=10,
...                  ecolor=ebet, eorder=ebet, output="graphviz-draw.pdf")
<...>
```

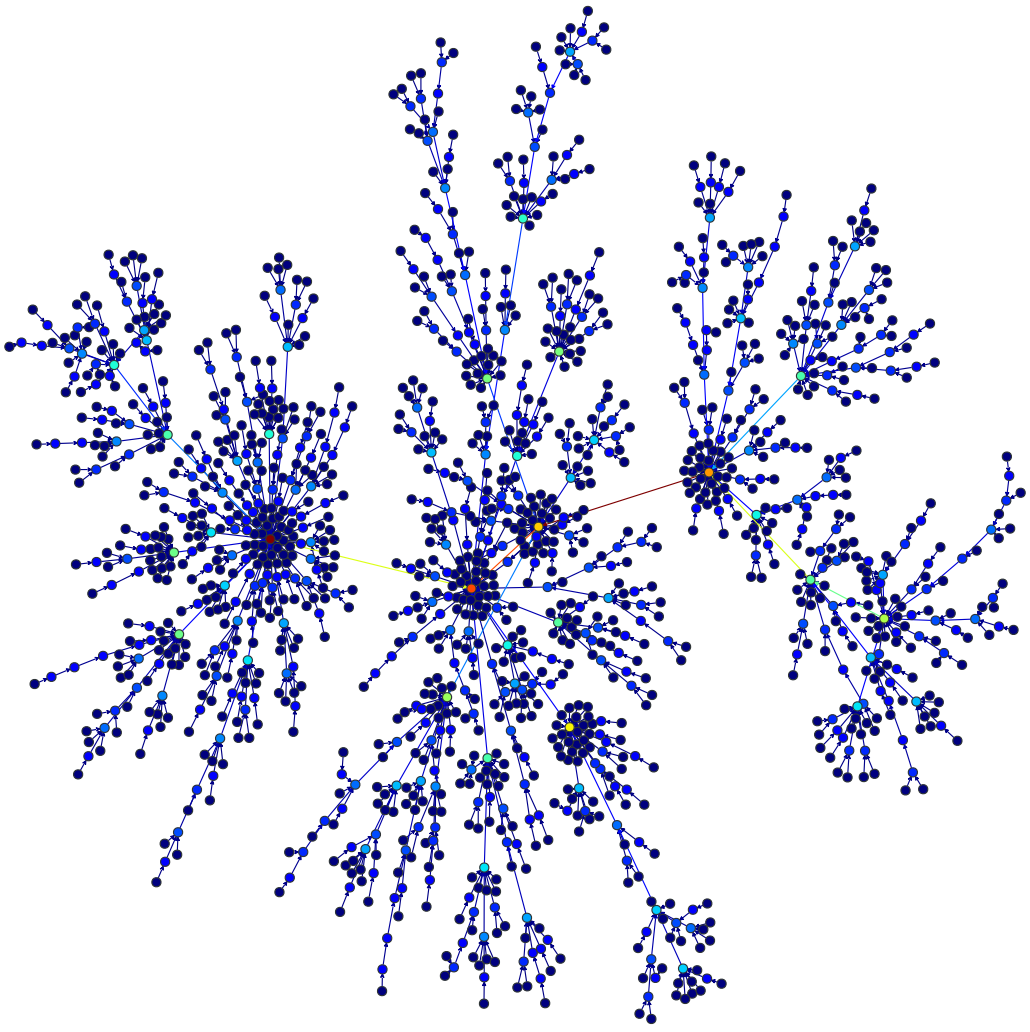


Figure 3.25: Kamada-Kawai force-directed layout of a Price network with 1500 nodes. The vertex size and color indicate the degree, and the edge color corresponds to the edge betweenness centrality

graph_tool.draw.**prop_to_size**(*prop*, *mi=0*, *ma=5*, *log=False*, *power=0.5*)

Convert property map values to be more useful as a vertex size, or edge width. The new values are taken to be

$$y = mi + (ma - mi)\left(\frac{x_i - min(x)}{max(x) - min(x)}\right)^{\text{power}}$$

If *log=True*, the natural logarithm of the property values are used instead.

Low-level graph drawing

graph_tool.draw.**cairo_draw**(*g*, *pos*, *cr*, *vprops=None*, *eprops=None*, *vorder=None*, *eorder=None*, *nodesfirst=False*, *vcmap=<matplotlib.colors.LinearSegmentedColormap object at 0x7fc148313650>*, *ecmap=<matplotlib.colors.LinearSegmentedColormap object at 0x7fc148313650>*, *loop_angle=nan*, *parallel_distance=None*, *fit_view=False*, ***kwargs*)

Draw a graph to a `cairo` context.

Parameters **g** : `Graph` (page 28)

Graph to be drawn.

pos : `PropertyMap` (page 35)

Vector-valued vertex property map containing the x and y coordinates of the vertices.

cr : `Context`

A `Context` instance.

vprops : dict (optional, default: `None`)

Dictionary with the vertex properties. Individual properties may also be given via the `vertex_<prop-name>` parameters, where `<prop-name>` is the name of the property.

eprops : dict (optional, default: `None`)

Dictionary with the vertex properties. Individual properties may also be given via the `edge_<prop-name>` parameters, where `<prop-name>` is the name of the property.

vorder : `PropertyMap` (page 35) (optional, default: `None`)

If provided, defines the relative order in which the vertices are drawn.

eorder : `PropertyMap` (page 35) (optional, default: `None`)

If provided, defines the relative order in which the edges are drawn.

nodesfirst : bool (optional, default: `False`)

If `True`, the vertices are drawn first, otherwise the edges are.

vcmap : `matplotlib.colors.Colormap` (default: `matplotlib.cm.jet`)

Vertex color map.

ecmap : `matplotlib.colors.Colormap` (default: `matplotlib.cm.jet`)

Edge color map.

loop_angle : float (optional, default: `nan`)

Angle used to draw self-loops. If `nan` is given, they will be placed radially from the center of the layout.

parallel_distance : float (optional, default: `None`)

Distance used between parallel edges. If not provided, it will be determined automatically.

fit_view : bool (optional, default: `True`)

If `True`, the layout will be scaled to fit the entire clip region.

vertex_* : `PropertyMap` (page 35) or arbitrary types (optional, default: `None`)

Parameters following the pattern `vertex_<prop-name>` specify the vertex property with name `<prop-name>`, as an alternative to the `vprops` parameter.

edge_* : `PropertyMap` (page 35) or arbitrary types (optional, default: `None`)

Parameters following the pattern `edge_<prop-name>` specify the edge property with name `<prop-name>`, as an alternative to the `eprops` parameter.

graph_tool.draw.**interactive_window**(*g*, *pos=None*, *vprops=None*, *eprops=None*, *vorder=None*, *eorder=None*, *nodesfirst=False*, *geometry=(500, 400)*, *update_layout=True*, *async=False*, ***kwargs*)

Display an interactive GTK+ window containing the given graph.

Parameters **g** : `Graph` (page 28)

Graph to be drawn.

pos : `PropertyMap` (page 35) (optional, default: `None`)

Vector-valued vertex property map containing the x and y coordinates of the vertices. If not given, it will be computed using `sfdp_layout()` (page 103).

vprops : dict (optional, default: `None`)

Dictionary with the vertex properties. Individual properties may also be given via the `vertex_<prop-name>` parameters, where `<prop-name>` is the name of the property.

eprops : dict (optional, default: `None`)

Dictionary with the vertex properties. Individual properties may also be given via the `edge_<prop-name>` parameters, where `<prop-name>` is the name of the property.

vorder : `PropertyMap` (page 35) (optional, default: `None`)

If provided, defines the relative order in which the vertices are drawn.

eorder : `PropertyMap` (page 35) (optional, default: `None`)

If provided, defines the relative order in which the edges are drawn.

nodesfirst : bool (optional, default: `False`)

If `True`, the vertices are drawn first, otherwise the edges are.

geometry : tuple (optional, default: `(500, 400)`)

Window geometry.

update_layout : bool (optional, default: `True`)

If `True`, the layout will be updated dynamically.

async : bool (optional, default: `False`)

If `True`, run asynchronously. (Requires `IPython`)

****kwargs** :

Any extra parameters are passed to `GraphWindow` (page 125), `GraphWidget` (page 123) and `cairo_draw()` (page 121).

Returns **pos** : `PropertyMap` (page 35)

Vector vertex property map with the x and y coordinates of the vertices.

selected : `PropertyMap` (page 35) (optional, only if `output is None`)

Boolean-valued vertex property map marking the vertices which were selected interactively.

Notes

See documentation of `GraphWidget` (page 123) for key bindings information.

class `graph_tool.draw.`**`GraphWidget`**(*g*, *pos*, *vprops=None*, *eprops=None*, *vorder=None*, *eorder=None*, *nodesfirst=False*, *update_layout=False*, *layout_K=1.0*, *multilevel=False*, *display_props=None*, *display_props_size=11*, *bg_color=None*, ***kwargs*)

Interactive GTK+ widget displaying a given graph.

Parameters **g** : `Graph` (page 28)

Graph to be drawn.

pos : `PropertyMap` (page 35) (optional, default: `None`)

Vector-valued vertex property map containing the x and y coordinates of the vertices. If not given, it will be computed using `sfdp_layout()` (page 103).

vprops : dict (optional, default: `None`)

Dictionary with the vertex properties. Individual properties may also be given via the `vertex_<prop-name>` parameters, where `<prop-name>` is the name of the property.

eprops : dict (optional, default: `None`)

Dictionary with the vertex properties. Individual properties may also be given via the `edge_<prop-name>` parameters, where `<prop-name>` is the name of the property.

vorder : `PropertyMap` (page 35) (optional, default: `None`)

If provided, defines the relative order in which the vertices are drawn.

eorder : `PropertyMap` (page 35) (optional, default: `None`)

If provided, defines the relative order in which the edges are drawn.

nodesfirst : bool (optional, default: `False`)

If `True`, the vertices are drawn first, otherwise the edges are.

update_layout : bool (optional, default: `True`)

If `True`, the layout will be updated dynamically.

layout_K : float (optional, default: `1.0`)

Parameter `K` passed to `sfdp_layout()` (page 103).

multilevel : bool (optional, default: `False`)

Parameter `multilevel` passed to `sfdp_layout()` (page 103).

display_props : list of `PropertyMap` (page 35) instances (optional, default: `None`)

List of properties to be displayed when the mouse passes over a vertex.

display_props_size : float (optional, default: `11`)

Font size used to display the vertex properties.

bg_color : str or sequence (optional, default: `None`)

Background color. The default is white.

vertex_* : `PropertyMap` (page 35) or arbitrary types (optional, default: `None`)

Parameters following the pattern `vertex_<prop-name>` specify the vertex property with name `<prop-name>`, as an alternative to the `vprops` parameter.

edge_* : `PropertyMap` (page 35) or arbitrary types (optional, default: `None`)

Parameters following the pattern `edge_<prop-name>` specify the edge property with name `<prop-name>`, as an alternative to the `eprops` parameter.

****kwargs** :

Any extra parameters are passed to `cairo_draw()` (page 121).

Notes

The graph drawing can be panned by dragging with the middle mouse button pressed. The graph may be zoomed by scrolling with the mouse wheel, or equivalent (if the "shift" key is held, the vertex/edge sizes are scaled accordingly). The layout may be rotated by dragging while holding the "control" key. Pressing the "r" key centers and zooms the layout around the graph. By pressing the "a" key, the current translation, scaling and rotation transformations are applied to the vertex positions themselves, and the transformation matrix is reset (if this is never done, the given position properties are never modified).

Individual vertices may be selected by pressing the left mouse button. The currently selected vertex follows the mouse pointer. To stop the selection, the right mouse button must be pressed. Alternatively, a group of vertices may be selected by holding the "shift" button while the pointer is dragged while pressing the left button. The selected vertices may be moved by dragging the pointer with the left button pressed. They may be rotated by holding the "control" key and scrolling with the mouse. If the key "z" is pressed, the layout is zoomed to fit the selected vertices only.

If the key "s" is pressed, the dynamic spring-block layout is activated. Vertices which are currently selected are not updated.

cleanup(*self*)
Cleanup callbacks.

reset_layout(*self*)
Reset the layout algorithm.

layout_callback(*self*)
Perform one step of the layout algorithm.

regenerate_surface(*self*, *lazy=True*, *timeout=350*)
Redraw the graph surface. If lazy is True, the actual redrawing will be performed after the specified timeout.

draw(*self*, *da*, *cr*)
Redraw the widget.

pos_to_device(*self*, *pos*, *dist=False*, *surface=False*, *cr=None*)
Convert a position from the graph space to the widget space.

pos_from_device(*self*, *pos*, *dist=False*, *surface=False*, *cr=None*)
Convert a position from the widget space to the device space.

apply_transform(*self*)
Apply current transform matrix to vertex coordinates.

fit_to_window(*self*, *ink=False*, *g=None*)
Fit graph to window.

init_picked(*self*)
Init picked vertices.

button_press_event(*self*, *widget*, *event*)
Handle button press.

button_release_event(*self*, *widget*, *event*)
Handle button release.

motion_notify_event(*self*, *widget*, *event*)
Handle pointer motion.

scroll_event(*self*, *widget*, *event*)
Handle scrolling.

key_press_event(*self*, *widget*, *event*)
Handle key press.

key_release_event(*self*, *widget*, *event*)
Handle release event.

class graph_tool.draw.**GraphWindow**(*g*, *pos*, *geometry*, *vprops=None*, *eprops=None*, *vorder=None*, *eorder=None*, *nodesfirst=False*, *update_layout=False*, ***kwargs*)
Interactive GTK+ window containing a GraphWidget (page 123).

Parameters **g** : Graph (page 28)

Graph to be drawn.

pos : PropertyMap (page 35) (optional, default: None)

Vector-valued vertex property map containing the x and y coordinates of the vertices. If not given, it will be computed using sfdp_layout() (page 103).

geometry : tuple

Widget geometry.

vprops : dict (optional, default: None)

Dictionary with the vertex properties. Individual properties may also be given via the vertex_<prop-name> parameters, where <prop-name> is the name of the property.

eprops : dict (optional, default: None)

Dictionary with the vertex properties. Individual properties may also be given via the edge_<prop-name> parameters, where <prop-name> is the name of the property.

vorder : PropertyMap (page 35) (optional, default: None)

If provided, defines the relative order in which the vertices are drawn.

eorder : PropertyMap (page 35) (optional, default: None)

If provided, defines the relative order in which the edges are drawn.

nodesfirst : bool (optional, default: False)

If True, the vertices are drawn first, otherwise the edges are.

update_layout : bool (optional, default: True)

If True, the layout will be updated dynamically.

****kwargs** :

Any extra parameters are passed to GraphWidget (page 123) and cairo_draw() (page 121).

3.2.7 graph_tool.flow - Maximum flow algorithms

Summary

edmonds_karp_max_flow (page 126)	Calculate maximum flow on the graph with the Edmonds-Ka
push_relabel_max_flow (page 128)	Calculate maximum flow on the graph with the push-relabel
boykov_kolmogorov_max_flow (page 130)	Calculate maximum flow on the graph with the Boykov-Kolm
min_st_cut (page 132)	Get the minimum source-target cut, given the residual capac
min_cut (page 134)	Get the minimum cut of an undirected graph, given the weig

Contents

The following network will be used as an example throughout the documentation.

```
from numpy.random import seed, random
from scipy.linalg import norm
gt.seed_rng(42)
seed(42)
points = random((400, 2))
points[0] = [0, 0]
points[1] = [1, 1]
g, pos = gt.triangulation(points, type="delaunay")
g.set_directed(True)
edges = list(g.edges())
# reciprocate edges
for e in edges:
   g.add_edge(e.target(), e.source())
# The capacity will be defined as the inverse euclidean distance
cap = g.new_edge_property("double")
for e in g.edges():
    cap[e] = min(1.0 / norm(pos[e.target()].a - pos[e.source()].a), 10)
g.edge_properties["cap"] = cap
g.vertex_properties["pos"] = pos
g.save("flow-example.xml.gz")
gt.graph_draw(g, pos=pos, edge_pen_width=gt.prop_to_size(cap, mi=0, ma=3, power=1),
              output="flow-example.pdf")
```

graph_tool.flow.**edmonds_karp_max_flow**(*g*, *source*, *target*, *capacity*, *residual=None*)

Calculate maximum flow on the graph with the Edmonds-Karp algorithm.

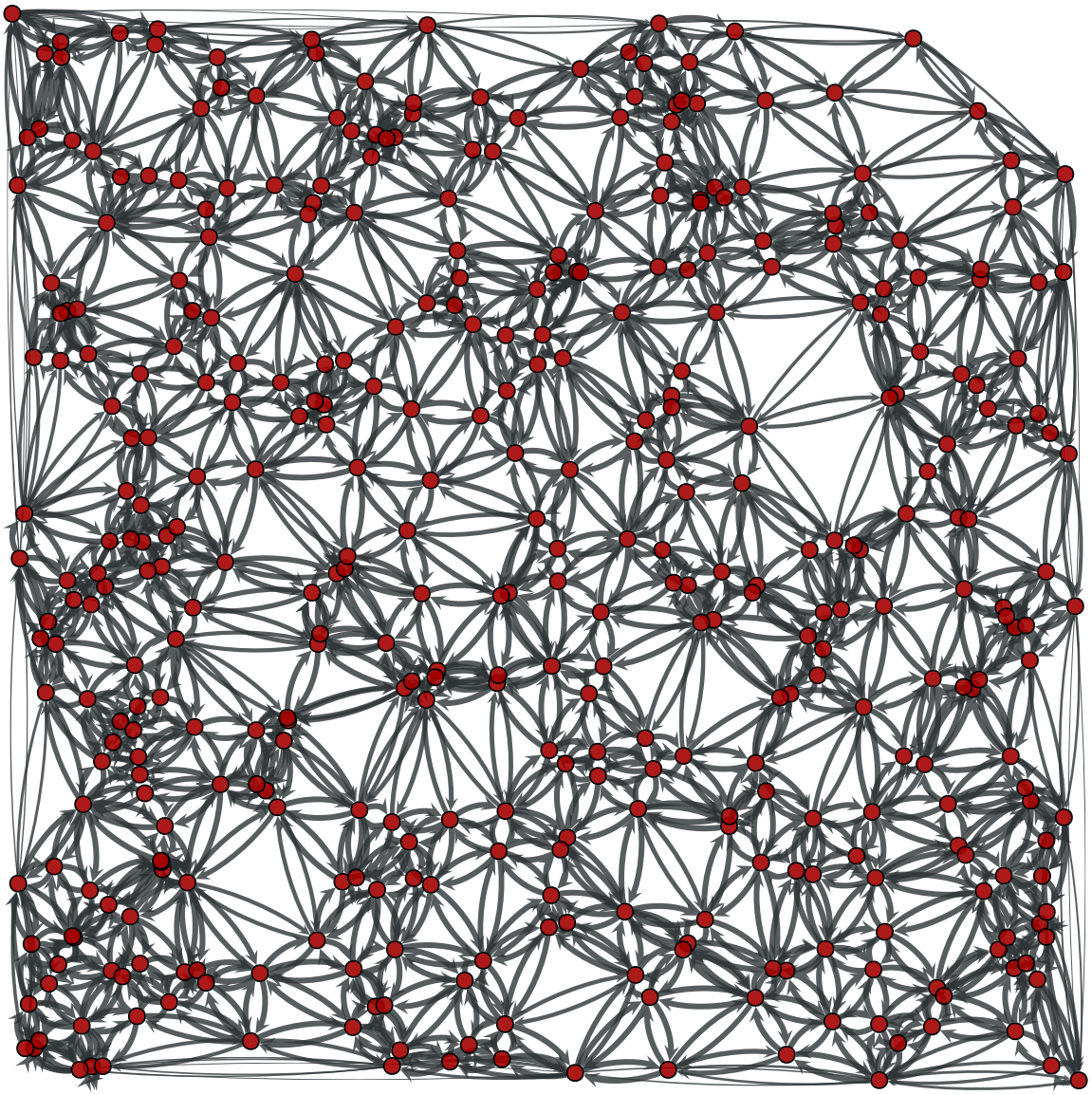


Figure 3.26: Example network used in the documentation below. The edge capacities are represented by the edge width.

Parameters **g** : Graph (page 28)

Graph to be used.

source : Vertex

The source vertex.

target : Vertex

The target (or "sink") vertex.

capacity : PropertyMap (page 35)

Edge property map with the edge capacities.

residual : PropertyMap (page 35) (optional, default: none)

Edge property map where the residuals should be stored.

Returns **residual** : PropertyMap (page 35)

Edge property map with the residual capacities (capacity - flow).

Notes

The algorithm is due to [edmonds-theoretical-1972] (page 229), though we are using the variation called the "labeling algorithm" described in [ravindra-network-1993] (page 230).

This algorithm provides a very simple and easy to implement solution to the maximum flow problem. However, there are several reasons why this algorithm is not as good as the push_relabel_max_flow() or the boykov_kolmogorov_max_flow() algorithm.

- In the non-integer capacity case, the time complexity is $O(VE^2)$ which is worse than the time complexity of the push-relabel algorithm $O(V^2E^{1/2})$ for all but the sparsest of graphs.
- In the integer capacity case, if the capacity bound U is very large then the algorithm will take a long time.

The time complexity is $O(VE^2)$ in the general case or $O(VEU)$ if capacity values are integers bounded by some constant U.

References

[boost-edmonds-karp] (page 229), [edmonds-theoretical-1972] (page 229), [ravindra-network-1993] (page 230)

Examples

```
>>> g = gt.load_graph("flow-example.xml.gz")
>>> cap = g.edge_properties["cap"]
>>> src, tgt = g.vertex(0), g.vertex(1)
>>> res = gt.edmonds_karp_max_flow(g, src, tgt, cap)
>>> res.a = cap.a - res.a  # the actual flow
>>> max_flow = sum(res[e] for e in tgt.in_edges())
>>> print(max_flow)
44.89059578411614
>>> pos = g.vertex_properties["pos"]
>>> gt.graph_draw(g, pos=pos, edge_pen_width=gt.prop_to_size(res, mi=0, ma=5, power=1), outpu
<...>
```

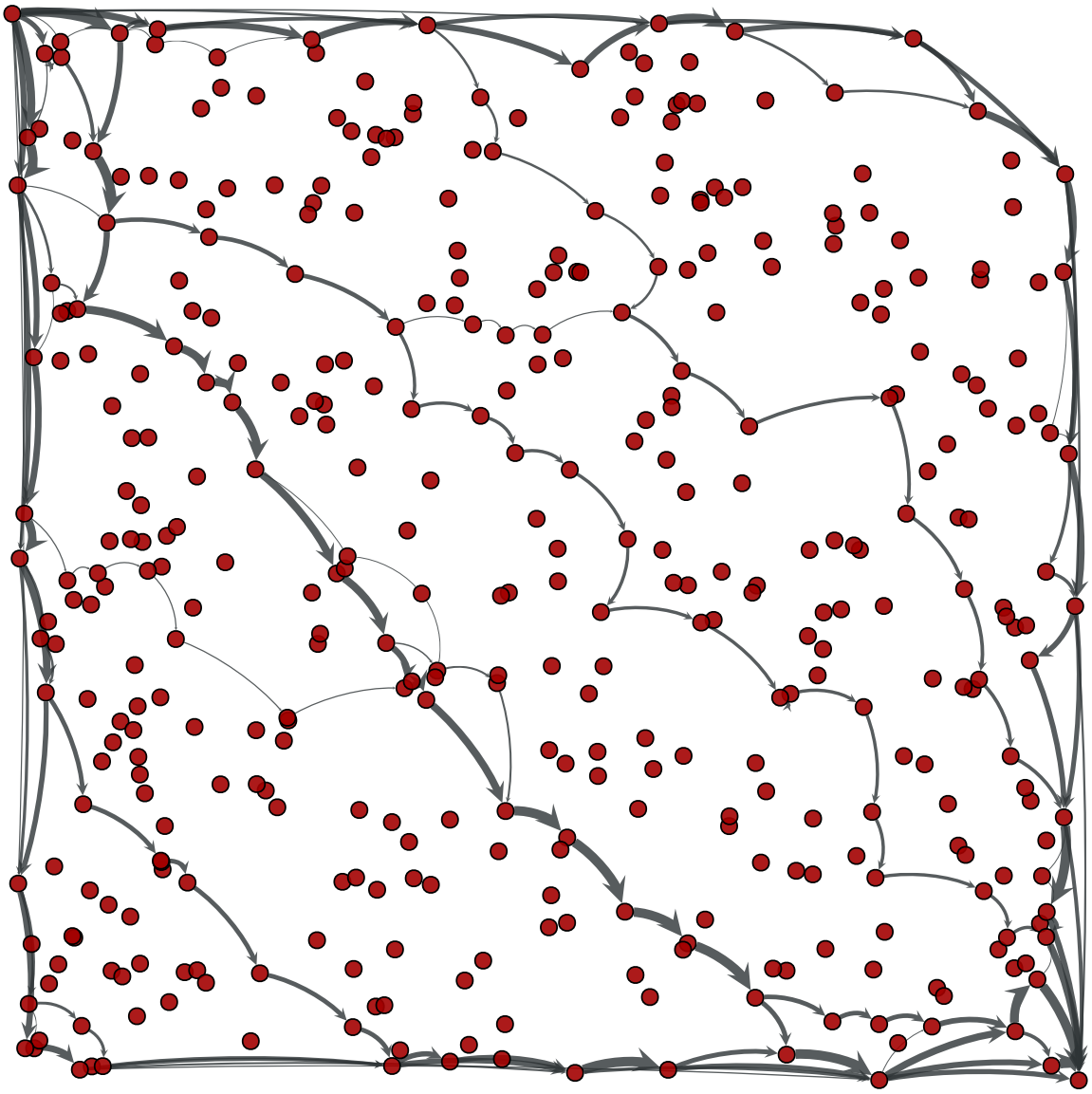


Figure 3.27: Edge flows obtained with the Edmonds-Karp algorithm. The source and target are on the lower left and upper right corners, respectively. The edge flows are represented by the edge width.

graph_tool.flow.**push_relabel_max_flow**(*g*, *source*, *target*, *capacity*, *residual=None*)

Calculate maximum flow on the graph with the push-relabel algorithm.

Parameters **g** : Graph (page 28)

Graph to be used.

source : Vertex

The source vertex.

target : Vertex

The target (or "sink") vertex.

capacity : PropertyMap (page 35)

Edge property map with the edge capacities.

residual : PropertyMap (page 35) (optional, default: none)

Edge property map where the residuals should be stored.

Returns **residual** : PropertyMap (page 35)

Edge property map with the residual capacities (capacity - flow).

Notes

The algorithm is defined in [goldberg-new-1985] (page 230). The complexity is $O(V^3)$.

References

[boost-push-relabel] (page 230), [goldberg-new-1985] (page 230)

Examples

```
>>> g = gt.load_graph("flow-example.xml.gz")
>>> cap = g.edge_properties["cap"]
>>> src, tgt = g.vertex(0), g.vertex(1)
>>> res = gt.push_relabel_max_flow(g, src, tgt, cap)
>>> res.a = cap.a - res.a  # the actual flow
>>> max_flow = sum(res[e] for e in tgt.in_edges())
>>> print(max_flow)
44.89059578411614
>>> pos = g.vertex_properties["pos"]
>>> gt.graph_draw(g, pos=pos, edge_pen_width=gt.prop_to_size(res, mi=0, ma=5, power=1), outpu
<...>
```

graph_tool.flow.**boykov_kolmogorov_max_flow**(*g*, *source*, *target*, *capacity*, *residual=None*)

Calculate maximum flow on the graph with the Boykov-Kolmogorov algorithm.

Parameters **g** : Graph (page 28)

Graph to be used.

source : Vertex

The source vertex.

target : Vertex

The target (or "sink") vertex.

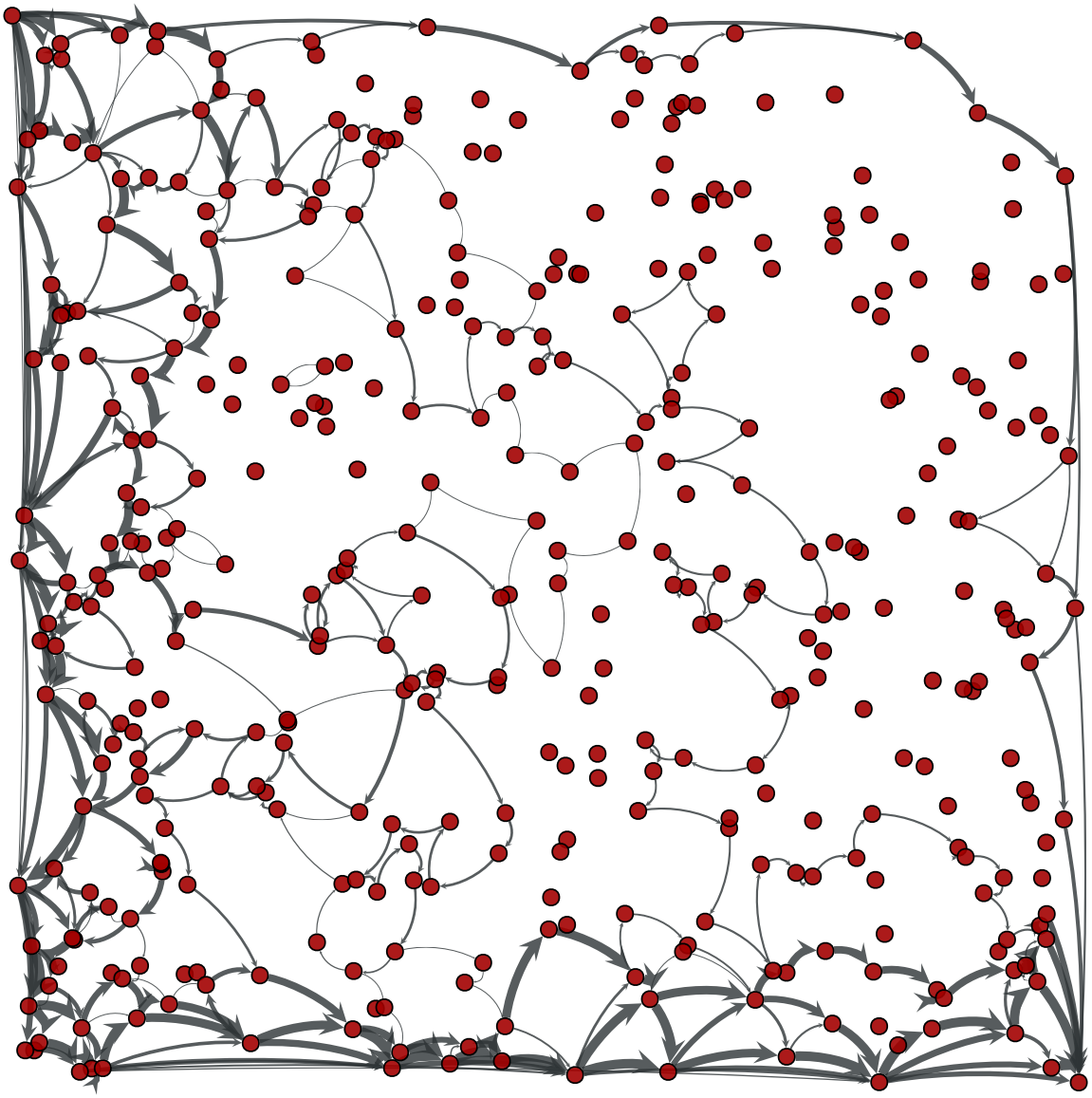


Figure 3.28: Edge flows obtained with the push-relabel algorithm. The source and target are on the lower left and upper right corners, respectively. The edge flows are represented by the edge width.

capacity : PropertyMap (page 35)

Edge property map with the edge capacities.

residual : PropertyMap (page 35) (optional, default: none)

Edge property map where the residuals should be stored.

Returns **residual** : PropertyMap (page 35)

Edge property map with the residual capacities (capacity - flow).

Notes

The algorithm is defined in [kolmogorov-graph-2003] (page 230) and [boykov-experimental-2004] (page 230). The worst case complexity is $O(EV^2|C|)$, where $|C|$ is the minimum cut (but typically performs much better).

For a more detailed description, see [boost-kolmogorov] (page 230).

References

[boost-kolmogorov] (page 230), [kolmogorov-graph-2003] (page 230), [boykov-experimental-2004] (page 230)

Examples

```
>>> g = gt.load_graph("flow-example.xml.gz")
>>> cap = g.edge_properties["cap"]
>>> src, tgt = g.vertex(0), g.vertex(1)
>>> res = gt.boykov_kolmogorov_max_flow(g, src, tgt, cap)
>>> res.a = cap.a - res.a  # the actual flow
>>> max_flow = sum(res[e] for e in tgt.in_edges())
>>> print(max_flow)
44.89059578411614
>>> pos = g.vertex_properties["pos"]
>>> gt.graph_draw(g, pos=pos, edge_pen_width=gt.prop_to_size(res, mi=0, ma=3, power=1), outpu
<...>
```

graph_tool.flow.**min_st_cut**(*g*, *source*, *residual*)

Get the minimum source-target cut, given the residual capacity of the edges.

Parameters **g** : Graph (page 28)

Graph to be used.

source : Vertex

The source vertex.

residual : PropertyMap (page 35)

Edge property map where the residual capacity is stored.

Returns **min_cut** : float

The value of the minimum cut.

partition : PropertyMap (page 35)

Boolean-valued vertex property map with the cut partition. Vertices with value *True* belong to the source side of the cut.

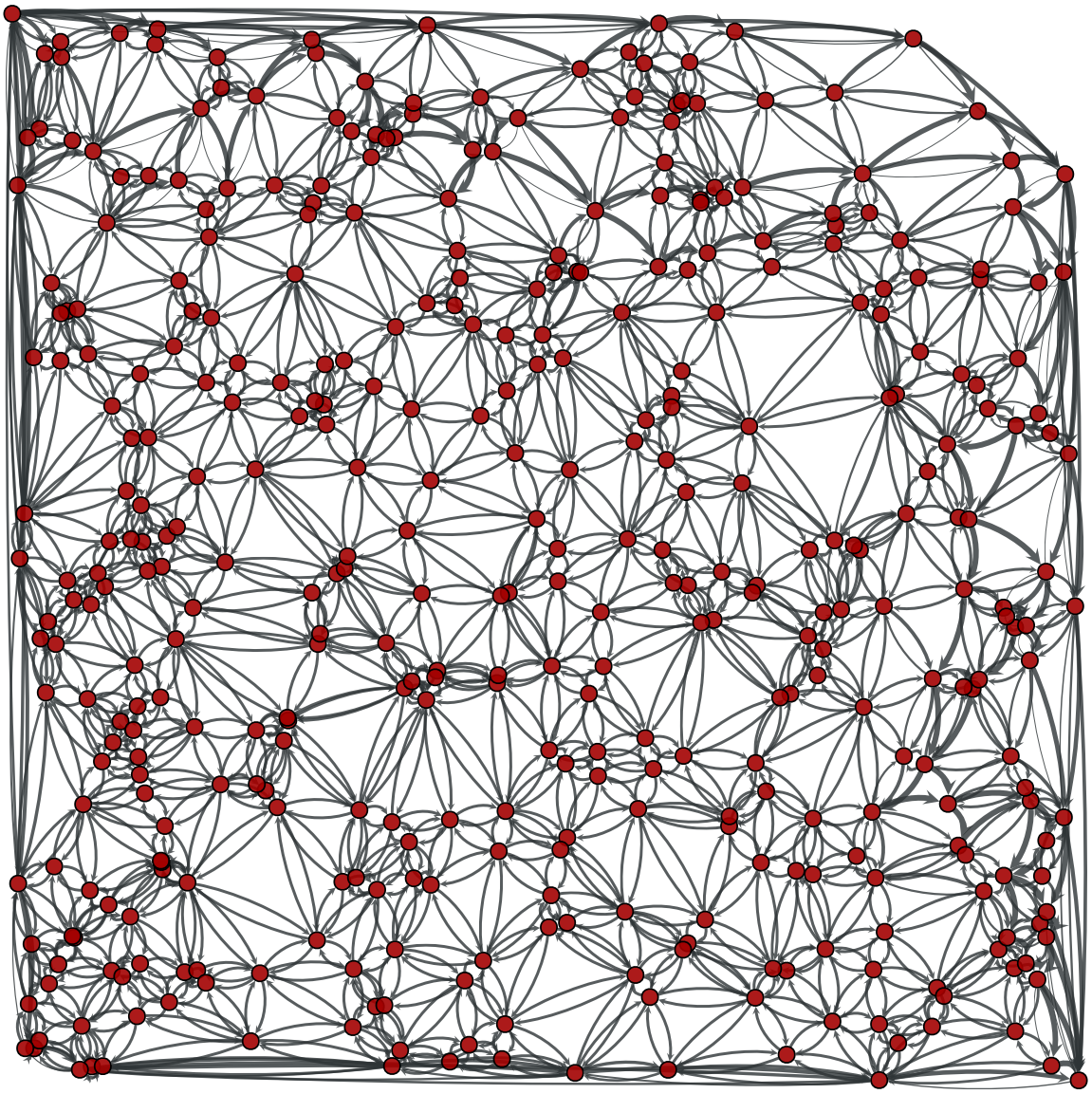


Figure 3.29: Edge flows obtained with the Boykov-Kolmogorov algorithm. The source and target are on the lower left and upper right corners, respectively. The edge flows are represented by the edge width.

Notes

The source-side of the cut set is obtained by following all vertices which are reachable from the source via edges with nonzero residual capacity.

This algorithm runs in $O(V + E)$ time.

References

[max-flow-min-cut] (page 230)

Examples

```
>>> g = gt.load_graph("flow-example.xml.gz")
>>> cap = g.edge_properties["cap"]
>>> src, tgt = g.vertex(0), g.vertex(1)
>>> res = gt.boykov_kolmogorov_max_flow(g, src, tgt, cap)
>>> mc, part = gt.min_st_cut(g, src, res)
>>> print(mc)
14.331937627198545
>>> pos = g.vertex_properties["pos"]
>>> res.a = cap.a - res.a  # the actual flow
>>> gt.graph_draw(g, pos=pos, edge_pen_width=gt.prop_to_size(res, mi=0, ma=3, power=1),
...               vertex_fill_color=part, output="example-min-st-cut.pdf")
<...>
```

graph_tool.flow.**min_cut**(*g*, *weight*)

Get the minimum cut of an undirected graph, given the weight of the edges.

Parameters **g** : Graph (page 28)

Graph to be used.

weight : PropertyMap (page 35)

Edge property map with the edge weights.

Returns **min_cut** : float

The value of the minimum cut.

partition : PropertyMap (page 35)

Boolean-valued vertex property map with the cut partition.

Notes

The algorithm is defined in [stoer_simple_1997] (page **??**).

The time complexity is $O(VE + V^2 \log V)$.

References

[stoer_simple_1997] (page **??**)

Examples

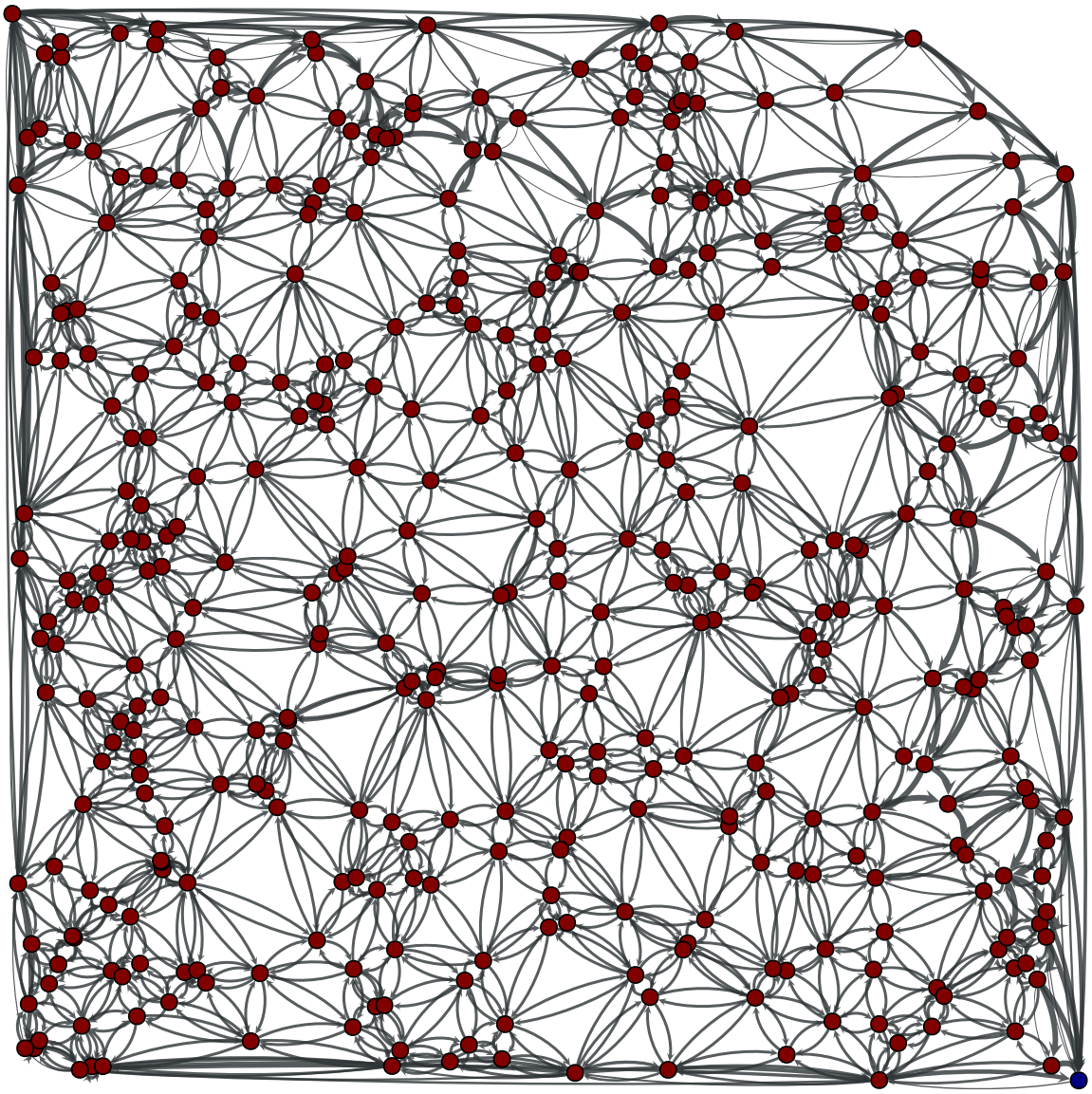


Figure 3.30: Edge flows obtained with the Boykov-Kolmogorov algorithm. The source and target are on the lower left and upper right corners, respectively. The edge flows are represented by the edge width. Vertices of the same color are on the same side of a minimum cut.

```
>>> g = gt.load_graph("mincut-example.xml.gz")
>>> weight = g.edge_properties["weight"]
>>> mc, part = gt.min_cut(g, weight)
>>> print(mc)
4.0
>>> pos = g.vertex_properties["pos"]
>>> gt.graph_draw(g, pos=pos, edge_pen_width=weight, vertex_fill_color=part,
...               output="example-min-cut.pdf")
<...>
```

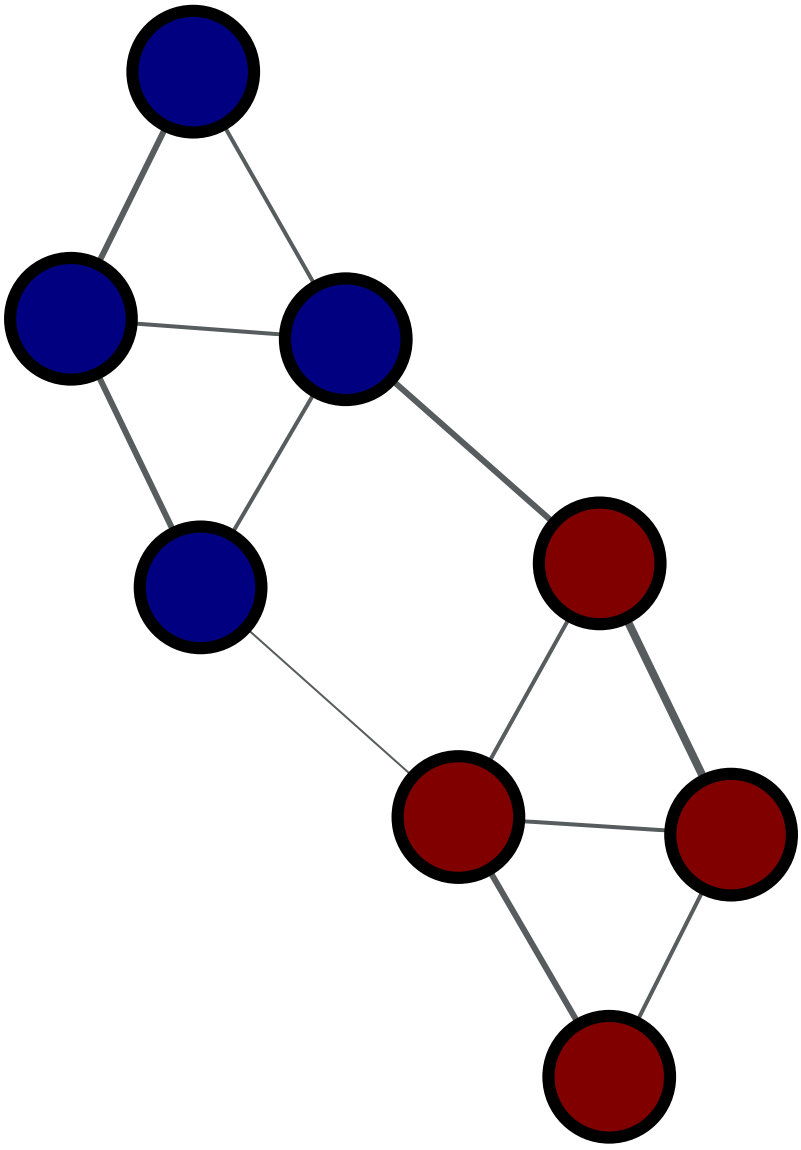


Figure 3.31: Vertices of the same color are on the same side of a minimum cut. The edge weights are represented by the edge width.

3.2.8 `graph_tool.generation` - Random graph generation

Summary

random_graph (page 137)	Generate a random graph, with a given degree distribution and (optionall
random_rewire (page 142)	Shuffle the graph in-place, following a variety of possible statistical mode
predecessor_tree (page 149)	Return a graph from a list of predecessors given by the pred_map vertex
line_graph (page 149)	Return the line graph of the given graph *g*.
graph_union (page 151)	Return the union of graphs g1 and g2, composed of all edges and vertices
triangulation (page 155)	Generate a 2D or 3D triangulation graph from a given point set.
lattice (page 157)	Generate a N-dimensional square lattice.
geometric_graph (page 160)	Generate a geometric network form a set of N-dimensional points.
price_network (page 162)	A generalized version of Price's – or Barabási-Albert if undirected – prefer
complete_graph (page **??**)	Generate complete graph.
circular_graph (page **??**)	Generate a circular graph.

Contents

graph_tool.generation.**random_graph**(*N*, *deg_sampler*, *directed=True*, *parallel_edges=False*, *self_loops=False*, *block_membership=None*, *block_type='int'*, *degree_block=False*, *random=True*, *verbose=False*, ***kwargs*)

Generate a random graph, with a given degree distribution and (optionally) vertex-vertex correlation.

The graph will be randomized via the random_rewire() (page 142) function, and any remaining parameters will be passed to that function. Please read its documentation for all the options regarding the different statistical models which can be chosen.

Parameters **N** : int

Number of vertices in the graph.

deg_sampler : function

A degree sampler function which is called without arguments, and returns a tuple of ints representing the in and out-degree of a given vertex (or a single int for undirected graphs, representing the out-degree). This function is called once per vertex, but may be called more times, if the degree sequence cannot be used to build a graph.

Optionally, you can also pass a function which receives one or two arguments. If block_membership == None, the single argument passed will be the index of the vertex which will receive the degree. If block_membership != None, the first value passed will be the vertex index, and the second will be the block value of the vertex.

directed : bool (optional, default: True)

Whether the generated graph should be directed.

parallel_edges : bool (optional, default: False)

If True, parallel edges are allowed.

self_loops : bool (optional, default: False)

If True, self-loops are allowed.

block_membership : list or ndarray or function (optional, default: None)

If supplied, the graph will be sampled from a stochastic blockmodel ensemble, and this parameter specifies the block membership of the vertices, which will be passed to the random_rewire() (page 142) function.

If the value is a list or a ndarray, it must have len(block_membership) == N, and the values will define to which block each vertex belongs.

If this value is a function, it will be used to sample the block types. It must be callable either with no arguments or with a single argument which will be the vertex index. In either case it must return a type compatible with the block_type parameter.

block_type : string (optional, default: "int")

Value type of block labels. Valid only if block_membership != None.

degree_block : bool (optional, default: False)

If True, the degree of each vertex will be appended to block labels when constructing the blockmodel, such that the resulting block type will be a pair (r, k), where r is the original block label.

random : bool (optional, default: True)

If True, the returned graph is randomized. Otherwise a deterministic placement of the edges will be used.

verbose : bool (optional, default: False)

If True, verbose information is displayed.

Returns **random_graph** : Graph (page 28)

The generated graph.

blocks : PropertyMap (page 35)

A vertex property map with the block values. This is only returned if block_membership != None.

See Also:

random_rewire (page 142) in-place graph shuffling

Notes

The algorithm makes sure the degree sequence is graphical (i.e. realizable) and keeps re-sampling the degrees if is not. With a valid degree sequence, the edges are placed deterministically, and later the graph is shuffled with the random_rewire() (page 142) function, with all remaining parameters passed to it.

The complexity is $O(V + E)$ if parallel edges are allowed, and $O(V + E \times \text{n-iter})$ if parallel edges are not allowed.

Note: If parallel_edges == False this algorithm only guarantees that the returned graph will be a random sample from the desired ensemble if n_iter is sufficiently large. The algorithm implements an efficient Markov chain based on edge swaps, with a mixing time which depends on the degree distribution and correlations desired. If degree correlations are provided, the mixing time tends to be larger.

References

[metropolis-equations-1953] (page **??**), [hastings-monte-carlo-1970] (page **??**), [holland-stochastic-1983] (page **??**), [karrer-stochastic-2011] (page **??**)

Examples

This is a degree sampler which uses rejection sampling to sample from the distribution $P(k) \propto 1/k$, up to a maximum.

```
>>> def sample_k(max):
...     accept = False
...     while not accept:
...         k = randint(1,max+1)
...         accept = random() < 1.0/k
...     return k
...
```

The following generates a random undirected graph with degree distribution $P(k) \propto 1/k$ (with k_max=40) and an *assortative* degree correlation of the form:

$$P(i,k) \propto \frac{1}{1+|i-k|}$$

```
>>> g = gt.random_graph(1000, lambda: sample_k(40), model="probabilistic",
...                     vertex_corr=lambda i, k: 1.0 / (1 + abs(i - k)), directed=False,
...                     n_iter=100)
>>> gt.scalar_assortativity(g, "out")
(0.6285094791115295, 0.010745128857935755)
```

The following samples an in,out-degree pair from the joint distribution:

$$p(j,k) = \frac{1}{2}\frac{e^{-m_1}m_1^j}{j!}\frac{e^{-m_1}m_1^k}{k!} + \frac{1}{2}\frac{e^{-m_2}m_2^j}{j!}\frac{e^{-m_2}m_2^k}{k!}$$

with $m_1 = 4$ and $m_2 = 20$.

```
>>> def deg_sample():
...     if random() > 0.5:
...         return poisson(4), poisson(4)
...     else:
...         return poisson(20), poisson(20)
...
```

The following generates a random directed graph with this distribution, and plots the combined degree correlation.

```
>>> g = gt.random_graph(20000, deg_sample)
>>>
>>> hist = gt.combined_corr_hist(g, "in", "out")
>>>
>>> clf()
>>> imshow(hist[0].T, interpolation="nearest", origin="lower")
<...>
>>> colorbar()
<...>
>>> xlabel("in-degree")
<...>
>>> ylabel("out-degree")
<...>
>>> savefig("combined-deg-hist.pdf")
```

A correlated directed graph can be build as follows. Consider the following degree correlation:

$$P(j',k'|j,k) = \frac{e^{-k}k^{j'}}{j'!}\frac{e^{-(20-j)}(20-j)^{k'}}{k'!}$$

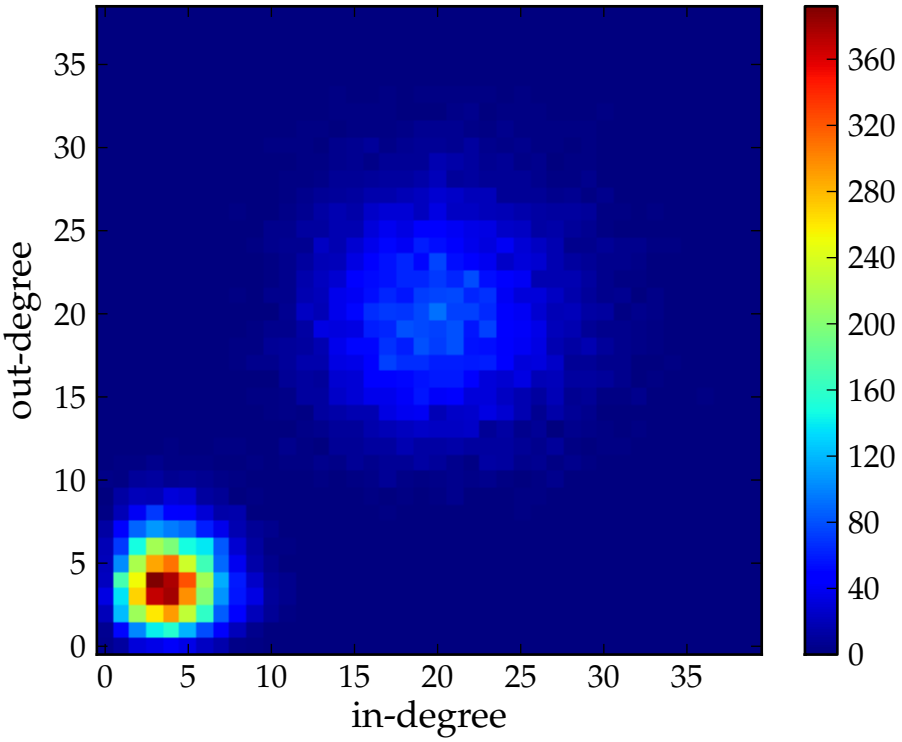


Figure 3.32: Combined degree histogram.

i.e., the in->out correlation is "disassortative", the out->in correlation is "assortative", and everything else is uncorrelated. We will use a flat degree distribution in the range [1,20).

```
>>> p = scipy.stats.poisson
>>> g = gt.random_graph(20000, lambda: (sample_k(19), sample_k(19)),
...                     model="probabilistic",
...                     vertex_corr=lambda a,b: (p.pmf(a[0], b[1]) *
...                                              p.pmf(a[1], 20 - b[0])),
...                     n_iter=100)
```

Lets plot the average degree correlations to check.

```
>>> clf()
>>> axes([0.1,0.15,0.63,0.8])
<...>
>>> corr = gt.avg_neighbour_corr(g, "in", "in")
>>> errorbar(corr[2][:-1], corr[0], yerr=corr[1], fmt="o-",
...          label=r"$\left<\text{in}\right>$ vs in")
<...>
>>> corr = gt.avg_neighbour_corr(g, "in", "out")
>>> errorbar(corr[2][:-1], corr[0], yerr=corr[1], fmt="o-",
...          label=r"$\left<\text{out}\right>$ vs in")
<...>
>>> corr = gt.avg_neighbour_corr(g, "out", "in")
>>> errorbar(corr[2][:-1], corr[0], yerr=corr[1], fmt="o-",
...           label=r"$\left<\text{in}\right>$ vs out")
<...>
>>> corr = gt.avg_neighbour_corr(g, "out", "out")
>>> errorbar(corr[2][:-1], corr[0], yerr=corr[1], fmt="o-",
```

```
...           label=r"$\left<\text{out}\right>$ vs out")
<...>
>>> legend(bbox_to_anchor=(1.01, 0.5), loc="center left", borderaxespad=0.)
<...>
>>> xlabel("Source degree")
<...>
>>> ylabel("Average target degree")
<...>
>>> savefig("deg-corr-dir.pdf")
```

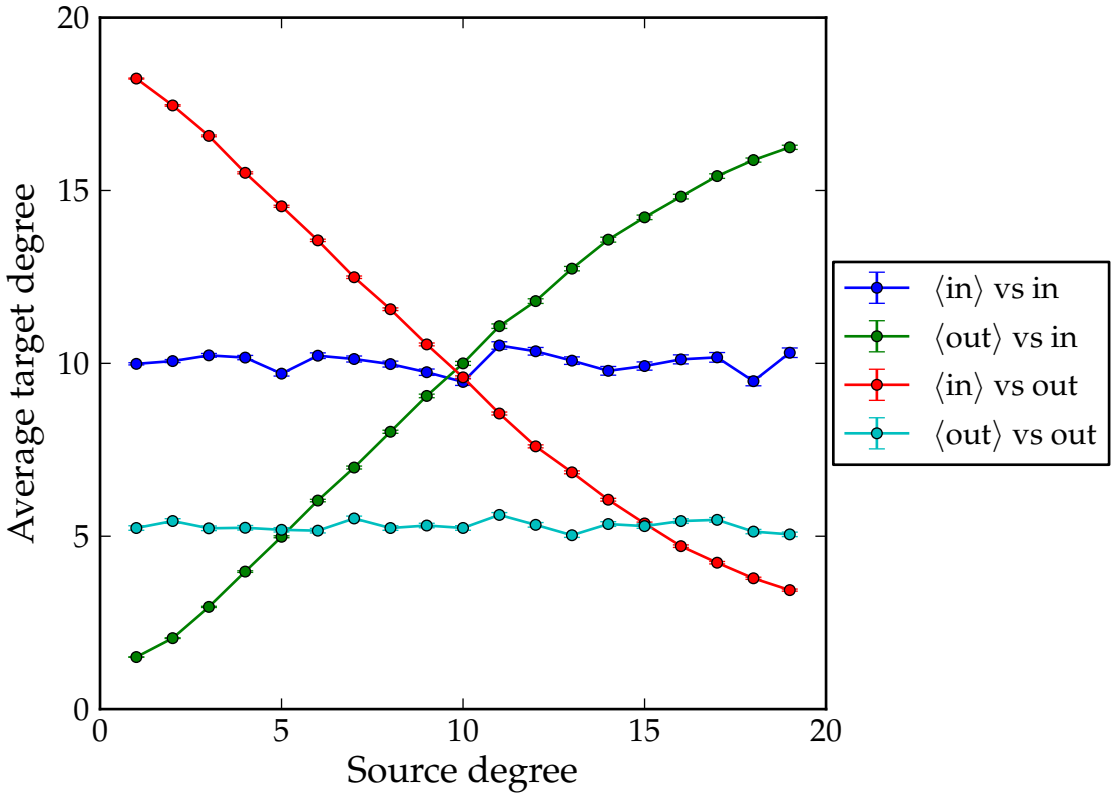


Figure 3.33: Average nearest neighbour correlations.

Stochastic blockmodels

The following example shows how a stochastic blockmodel [holland-stochastic-1983] (page **??**) [karrer-stochastic-2011] (page **??**) can be generated. We will consider a system of 10 blocks, which form communities. The connection probability will be given by

```
>>> def corr(a, b):
...     if a == b:
...         return 0.999
...     else:
...         return 0.001
```

The blockmodel can be generated as follows.

```
>>> g, bm = gt.random_graph(5000, lambda: poisson(10), directed=False,
...                         model="blockmodel-traditional",
...                         block_membership=lambda: randint(10),
...                         vertex_corr=corr)
>>> gt.graph_draw(g, vertex_fill_color=bm, edge_color="black", output="blockmodel.pdf")
<...>
```

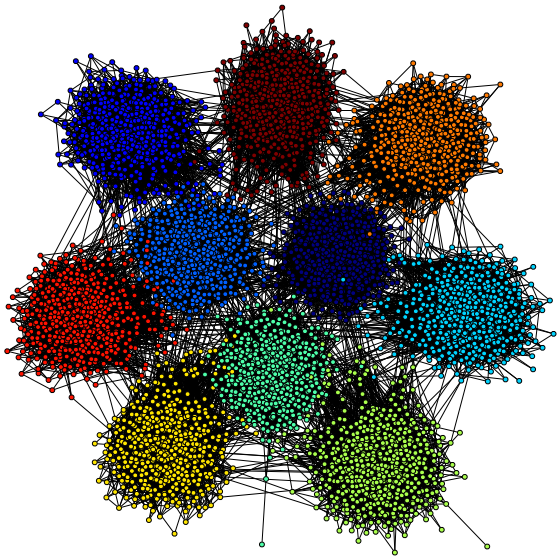


Figure 3.34: Simple blockmodel with 10 blocks.

graph_tool.generation.**random_rewire**(*g*, *model='uncorrelated'*, *n_iter=1*, *edge_sweep=True*, *parallel_edges=False*, *self_loops=False*, *vertex_corr=None*, *block_membership=None*, *alias=True*, *cache_probs=True*, *persist=False*, *ret_fail=False*, *verbose=False*)

Shuffle the graph in-place, following a variety of possible statistical models, chosen via the parameter `model`.

Parameters **g** : `Graph` (page 28)

Graph to be shuffled. The graph will be modified.

model : string (optional, default: `"uncorrelated"`)

The following statistical models can be chosen, which determine how the edges are rewired.

`erdos` The edges will be rewired entirely randomly, and the resulting graph will correspond to the Erdős–Rényi model.

`uncorrelated` The edges will be rewired randomly, but the degree sequence of the graph will remain unmodified.

`correlated` The edges will be rewired randomly, but both the degree sequence of the graph and the *vertex-vertex degree correlations* will remain unmodified.

`probabilistic` This is similar to the `correlated` option, but the vertex-vertex correlations are not kept unmodified, but instead are sampled from an arbitrary degree-based probabilistic model specified via the `vertex_corr` parameter.

`blockmodel` This is just like `probabilistic`, but the values passed to the `vertex_corr` function will correspond to the block membership values specified by the `block_membership` parameter.

`blockmodel-traditional` This is just like `blockmodel`, but the degree sequence *is not* preserved during rewiring.

n_iter : int (optional, default: `1`)

Number of iterations. If `edge_sweep == True`, each iteration corresponds to an entire "sweep" over all edges. Otherwise this corresponds to the total number of edges which are randomly chosen for a swap attempt (which may repeat).

edge_sweep : bool (optional, default: `True`)

If `True`, each iteration will perform an entire "sweep" over the edges, where each edge is visited once in random order, and a edge swap is attempted.

parallel : bool (optional, default: `False`)

If `True`, parallel edges are allowed.

self_loops : bool (optional, default: `False`)

If `True`, self-loops are allowed.

vertex_corr : function (optional, default: `None`)

A function which gives the vertex-vertex correlation of the graph.

If `model == probabilistic` it should be callable with two parameters: the (in, out)-degree pair of the source vertex an edge, and the (in,out)-degree pair of the target of the same edge (for undirected graphs, both parameters are single values). The function should

return a number proportional to the probability of such an edge existing in the generated graph.

If `model == blockmodel` or `model == blockmodel-traditional`, the values passed to the function will be the block value of the respective vertices, as specified via the `block_membership`. The function should also return a number proportional to the probability of such an edge existing in the generated graph.

block_membership : `PropertyMap` (page 35) (optional, default: `None`)

If supplied, the graph will be rewired to conform to a blockmodel ensemble. The value must be a vertex property map which defines the block of each vertex.

alias : bool (optional, default: `True`)

If `True`, and `model` is any of `probabilistic`, `blockmodel`, or `blockmodel-traditional`, the alias method will be used to sample the candidate edges. In the case of `blockmodel-traditional`, if `parallel_edges == True` and `self_loops == True` this makes the sampling of the edges direct (not rejection based), so that `n_iter == 1` is enough to get an uncorrelated sample.

cache_probs : bool (optional, default: `True`)

If `True`, the probabilities returned by the `vertex_corr` parameter will be cached internally. This is crucial for good performance, since in this case the supplied python function is called only a few times, and not at every attempted edge rewire move. However, in the case were the different parameter combinations to the probability function is very large, the memory and time requirements to keep the cache may not be worthwhile.

persist : bool (optional, default: `False`)

If `True`, an edge swap which is rejected will be attempted again until it succeeds. This may improve the quality of the shuffling for some probabilistic models, and should be sufficiently fast for sparse graphs, but otherwise it may result in many repeated attempts for certain corner-cases in which edges are difficult to swap.

verbose : bool (optional, default: `False`)

If `True`, verbose information is displayed.

Returns **rejection_count** : int

Number of rejected edge moves (due to parallel edges or self-loops, or the probabilistic model used).

See Also:

`random_graph` **(page 137)** random graph generation

Notes

This algorithm iterates through all the edges in the network and tries to swap its target or source with the target or source of another edge. The selected canditate swaps are chosen according to the `model` parameter.

Note: If `parallel_edges = False`, parallel edges are not placed during rewiring. In this case, the returned graph will be a uncorrelated sample from the desired ensemble only if `n_iter` is sufficiently large. The algorithm implements an efficient Markov chain

based on edge swaps, with a mixing time which depends on the degree distribution and correlations desired. If degree probabilistic correlations are provided, the mixing time tends to be larger.

If `model` is either "probabilistic" or "blockmodel", the Markov chain still needs to be mixed, even if parallel edges and self-loops are allowed. In this case the Markov chain is implemented using the Metropolis-Hastings [metropolis-equations-1953] (page **??**) [hastings-monte-carlo-1970] (page **??**) acceptance/rejection algorithm. It will eventually converge to the desired probabilities for sufficiently large values of `n_iter`.

Each edge is tentatively swapped once per iteration, so the overall complexity is $O(V + E \times \text{n-iter})$. If `edge_sweep == False`, the complexity becomes $O(V + E + \text{n-iter})$.

References

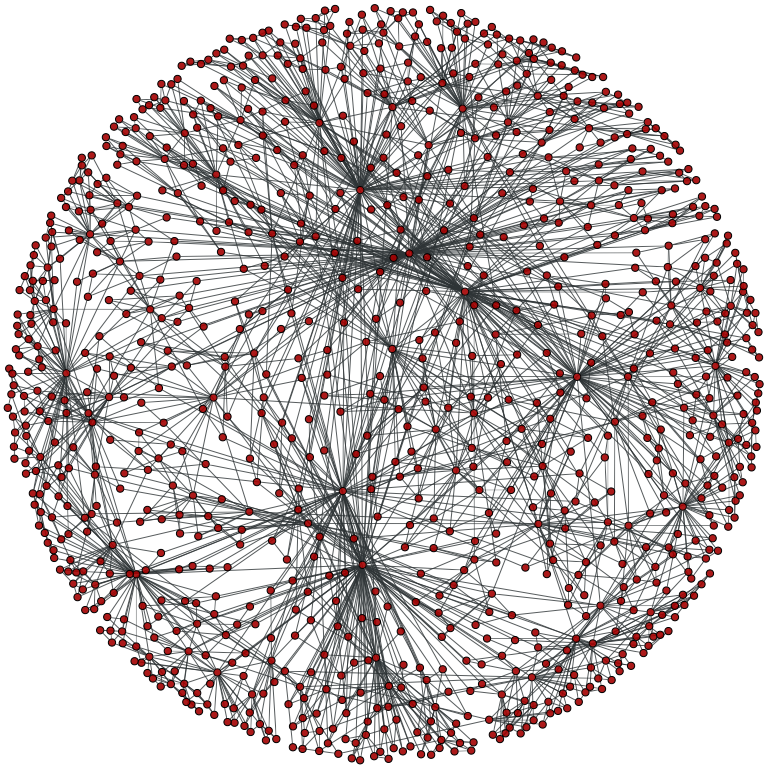
[metropolis-equations-1953] (page **??**), [hastings-monte-carlo-1970] (page **??**), [holland-stochastic-1983] (page **??**), [karrer-stochastic-2011] (page **??**)

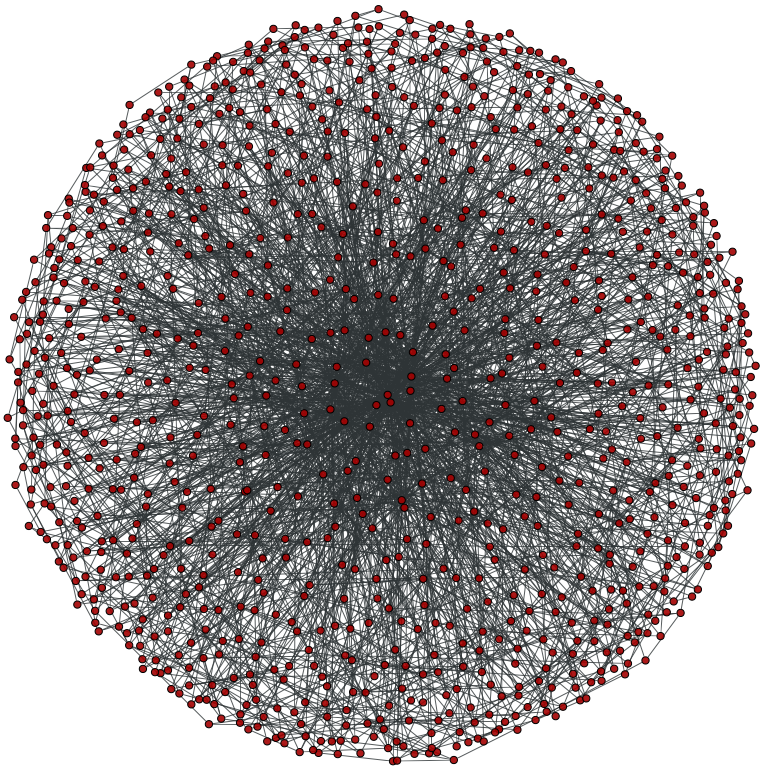
Examples

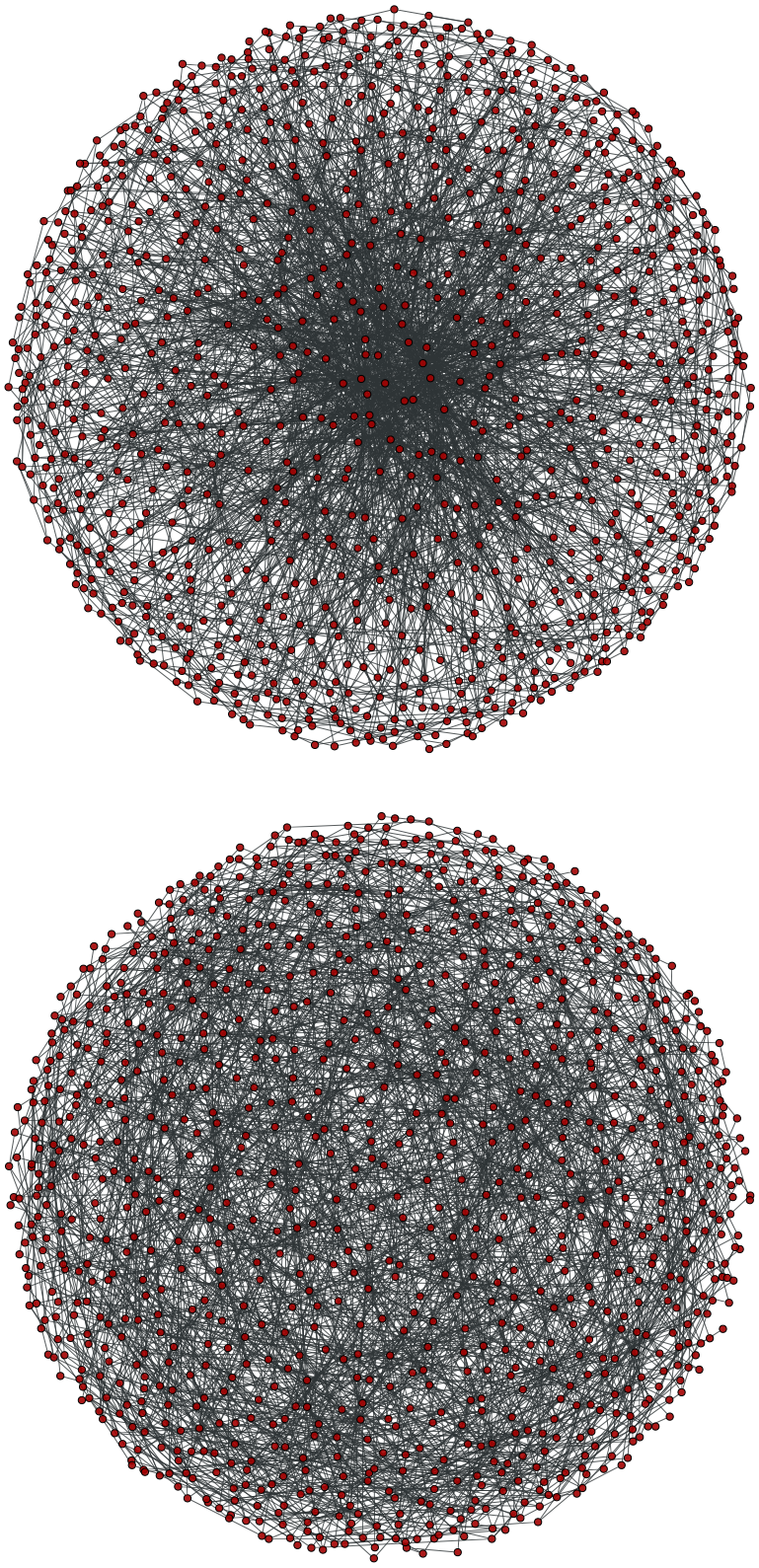
Some small graphs for visualization.

```
>>> g, pos = gt.triangulation(random((1000,2)))
>>> pos = gt.arf_layout(g)
>>> gt.graph_draw(g, pos=pos, output="rewire_orig.pdf", output_size=(300, 300))
<...>

>>> gt.random_rewire(g, "correlated")
189
>>> pos = gt.arf_layout(g)
>>> gt.graph_draw(g, pos=pos, output="rewire_corr.pdf", output_size=(300, 300))
<...>

>>> gt.random_rewire(g)
197
>>> pos = gt.arf_layout(g)
>>> gt.graph_draw(g, pos=pos, output="rewire_uncorr.pdf", output_size=(300, 300))
<...>

>>> gt.random_rewire(g, "erdos")
26
>>> pos = gt.arf_layout(g)
>>> gt.graph_draw(g, pos=pos, output="rewire_erdos.pdf", output_size=(300, 300))
<...>
```

Some ridiculograms :







From left to right: Original graph; Shuffled graph, with degree correlations; Shuffled graph, without degree correlations; Shuffled graph, with random degrees.

We can try with larger graphs to get better statistics, as follows.

```
>>> figure()
<...>
>>> g = gt.random_graph(30000, lambda: sample_k(20), model="probabilistic",
```

```
...                       vertex_corr=lambda i, j: exp(abs(i-j)), directed=False,
...                       n_iter=100)
>>> corr = gt.avg_neighbour_corr(g, "out", "out")
>>> errorbar(corr[2][:-1], corr[0], yerr=corr[1], fmt="o-", label="Original")
<...>
>>> gt.random_rewire(g, "correlated")
206
>>> corr = gt.avg_neighbour_corr(g, "out", "out")
>>> errorbar(corr[2][:-1], corr[0], yerr=corr[1], fmt="*", label="Correlated")
<...>
>>> gt.random_rewire(g)
109
>>> corr = gt.avg_neighbour_corr(g, "out", "out")
>>> errorbar(corr[2][:-1], corr[0], yerr=corr[1], fmt="o-", label="Uncorrelated")
<...>
>>> gt.random_rewire(g, "erdos")
13
>>> corr = gt.avg_neighbour_corr(g, "out", "out")
>>> errorbar(corr[2][:-1], corr[0], yerr=corr[1], fmt="o-", label=r"Erd\H{o}s")
<...>
>>> xlabel("$k$")
<...>
>>> ylabel(r"$\left<k_{nn}\right>$")
<...>
>>> legend(loc="best")
<...>
>>> savefig("shuffled-stats.pdf")
```

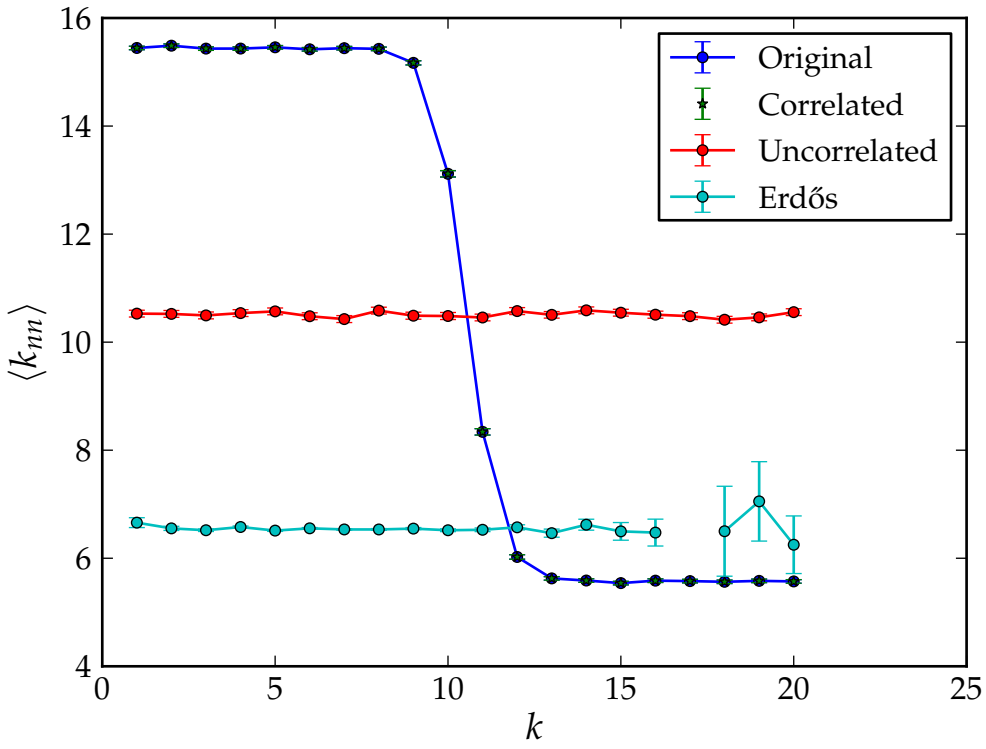


Figure 3.35: Average degree correlations for the different shuffled and non-shuffled graphs. The shuffled graph with correlations displays exactly the same correlation as the original graph.

Now let's do it for a directed graph. See random_graph() (page 137) for more details.

```
>>> p = scipy.stats.poisson
>>> g = gt.random_graph(20000, lambda: (sample_k(19), sample_k(19)),
...                     model="probabilistic",
...                     vertex_corr=lambda a, b: (p.pmf(a[0], b[1]) * p.pmf(a[1], 20 - b[0])),
...                     n_iter=100)
>>> figure()
<...>
>>> axes([0.1,0.15,0.6,0.8])
<...>
>>> corr = gt.avg_neighbour_corr(g, "in", "out")
>>> errorbar(corr[2][:-1], corr[0], yerr=corr[1], fmt="o-",
...          label=r"$\left<\text{o}\right>$ vs i")
<...>
>>> corr = gt.avg_neighbour_corr(g, "out", "in")
>>> errorbar(corr[2][:-1], corr[0], yerr=corr[1], fmt="o-",
...          label=r"$\left<\text{i}\right>$ vs o")
<...>
>>> gt.random_rewire(g, "correlated")
4323
>>> corr = gt.avg_neighbour_corr(g, "in", "out")
>>> errorbar(corr[2][:-1], corr[0], yerr=corr[1], fmt="o-",
...          label=r"$\left<\text{o}\right>$ vs i, corr.")
<...>
>>> corr = gt.avg_neighbour_corr(g, "out", "in")
>>> errorbar(corr[2][:-1], corr[0], yerr=corr[1], fmt="o-",
...          label=r"$\left<\text{i}\right>$ vs o, corr.")
<...>
>>> gt.random_rewire(g, "uncorrelated")
153
>>> corr = gt.avg_neighbour_corr(g, "in", "out")
>>> errorbar(corr[2][:-1], corr[0], yerr=corr[1], fmt="o-",
...          label=r"$\left<\text{o}\right>$ vs i, uncorr.")
<...>
>>> corr = gt.avg_neighbour_corr(g, "out", "in")
>>> errorbar(corr[2][:-1], corr[0], yerr=corr[1], fmt="o-",
...          label=r"$\left<\text{i}\right>$ vs o, uncorr.")
<...>
>>> legend(bbox_to_anchor=(1.01, 0.5), loc="center left", borderaxespad=0.)
<...>
>>> xlabel("Source degree")
<...>
>>> ylabel("Average target degree")
<...>
>>> savefig("shuffled-deg-corr-dir.pdf")
```

graph_tool.generation.**predecessor_tree**(*g*, *pred_map*)

Return a graph from a list of predecessors given by the pred_map vertex property.

graph_tool.generation.**line_graph**(*g*)

Return the line graph of the given graph *g*.

Notes

Given an undirected graph G, its line graph L(G) is a graph such that

- each vertex of L(G) represents an edge of G; and
- two vertices of L(G) are adjacent if and only if their corresponding edges share a common endpoint ("are adjacent") in G.

For a directed graph, the second criterion becomes:

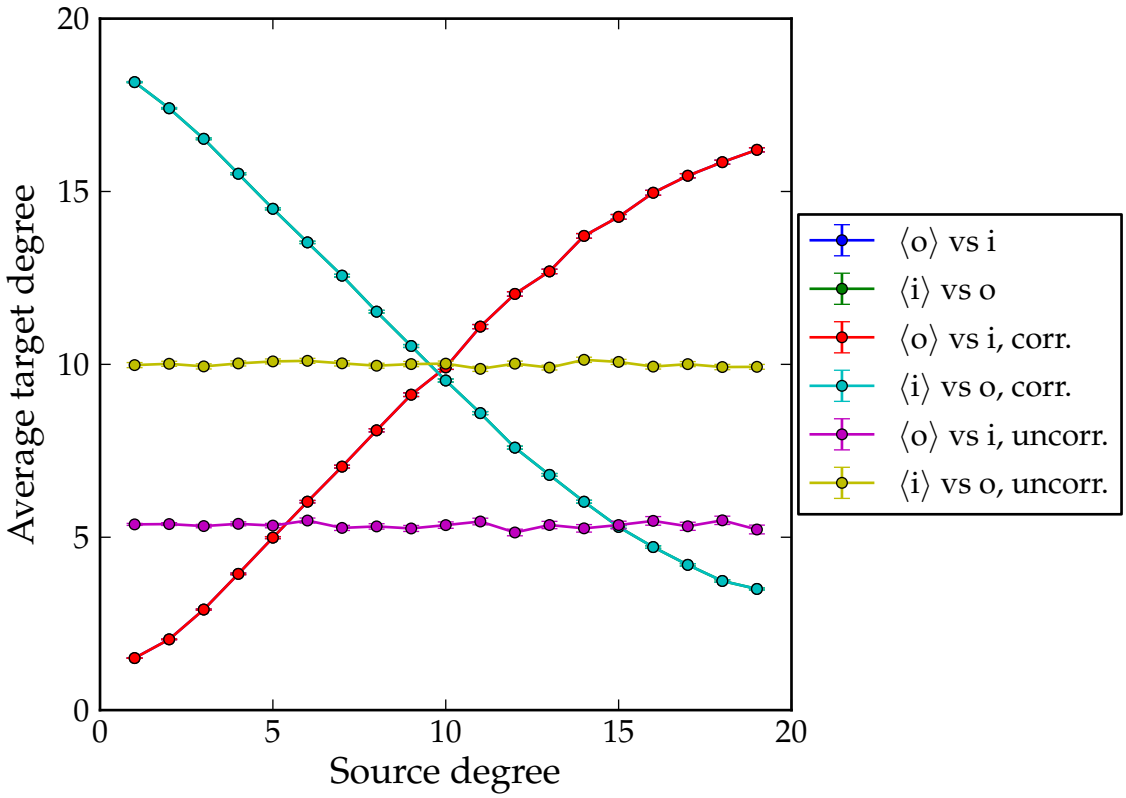


Figure 3.36: Average degree correlations for the different shuffled and non-shuffled directed graphs. The shuffled graph with correlations displays exactly the same correlation as the original graph.

- Two vertices representing directed edges from u to v and from w to x in G are connected by an edge from uv to wx in the line digraph when v = w.

References

[line-wiki] (page 230)

Examples

```
>>> g = gt.collection.data["lesmis"]
>>> lg, vmap = gt.line_graph(g)
>>> gt.graph_draw(g, pos=g.vp["pos"], output="lesmis.pdf")
<...>
>>> pos = gt.graph_draw(lg, output="lesmis-lg.pdf")
```

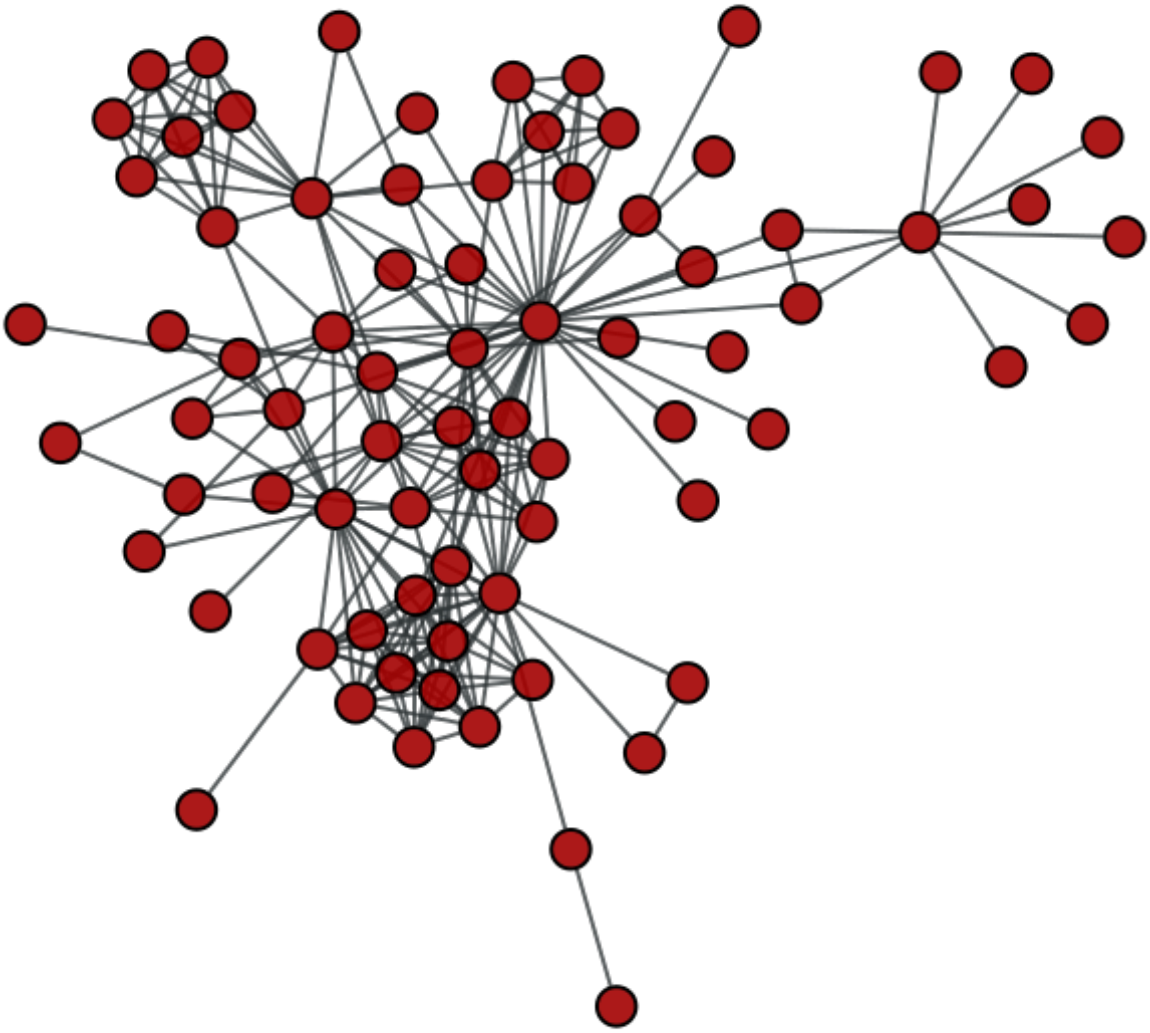


Figure 3.37: Coappearances of characters in Victor Hugo's novel "Les Miserables".

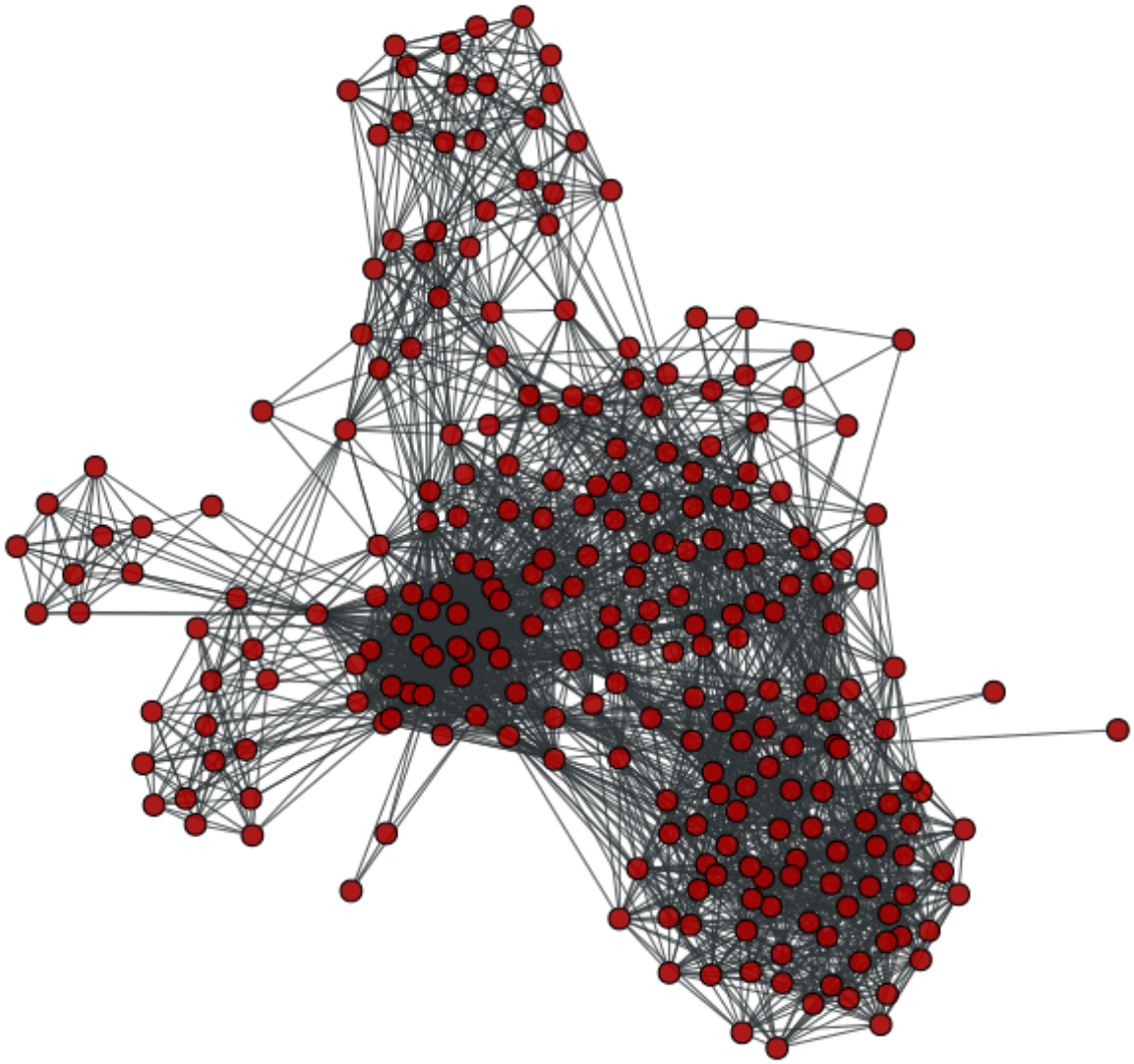


Figure 3.38: Line graph of the coappearance network on the left.

graph_tool.generation.**graph_union**(*g1*, *g2*, *intersection=None*, *props=None*, *include=False*)

Return the union of graphs g1 and g2, composed of all edges and vertices of g1 and g2, without overlap.

Parameters **g1** : Graph (page 28)

First graph in the union.

g2 : Graph (page 28)

Second graph in the union.

intersection : PropertyMap (page 35) (optional, default: None)

Vertex property map owned by *g1* which maps each of each of its vertices to vertex indexes belonging to *g2*. Negative values mean no mapping exists, and thus both vertices in *g1* and *g2* will be present in the union graph.

props : list of tuples of PropertyMap (page 35) (optional, default: [])

Each element in this list must be a tuple of two PropertyMap objects. The first element must be a property of *g1*, and the second of *g2*. The values of the property maps are propagated into the union graph, and returned.

include : bool (optional, default: False)

If true, graph *g2* is inserted into *g1* which is modified. If false, a new graph is created, and both graphs remain unmodified.

Returns **ug** : Graph (page 28)

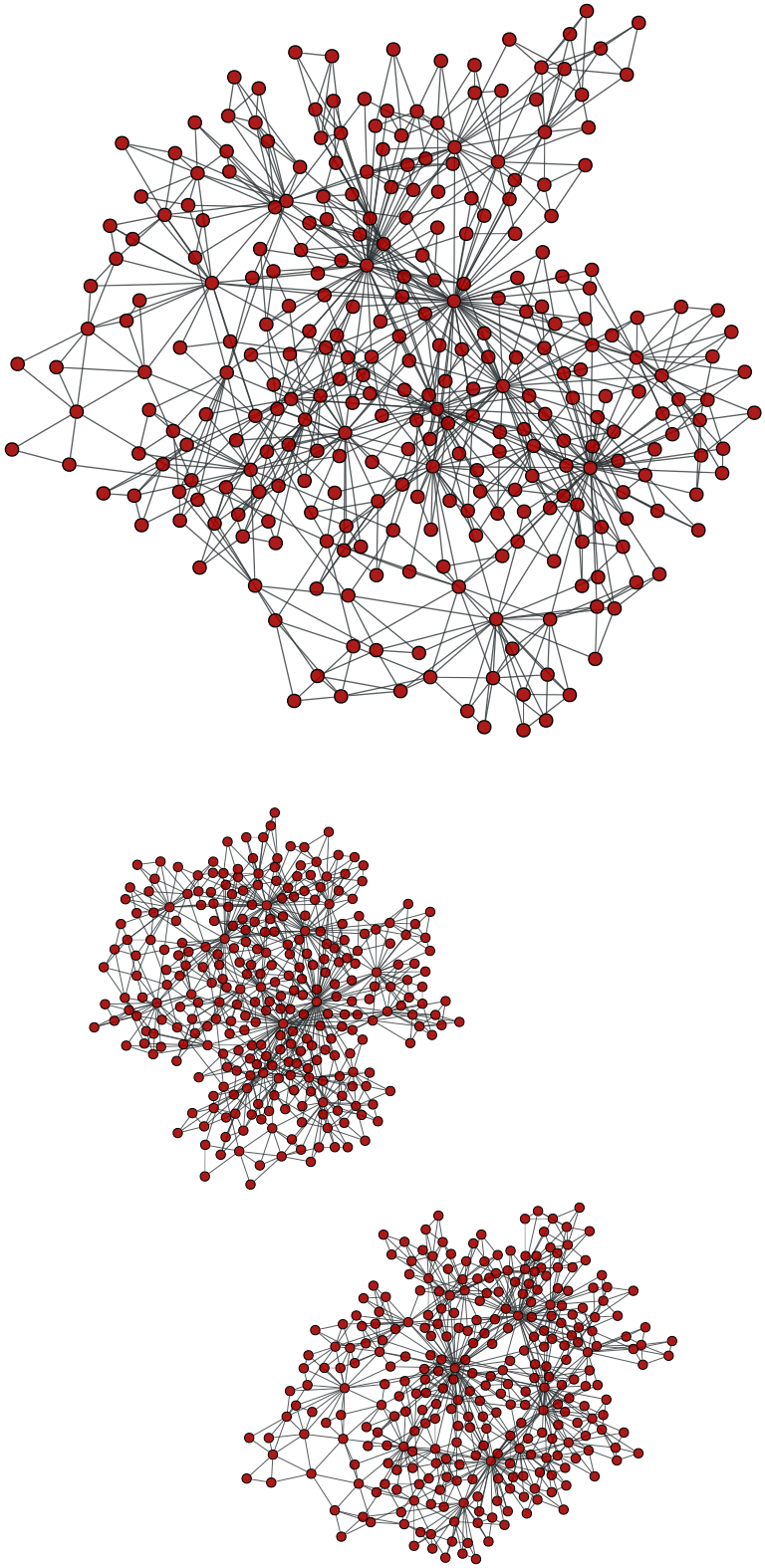
The union graph

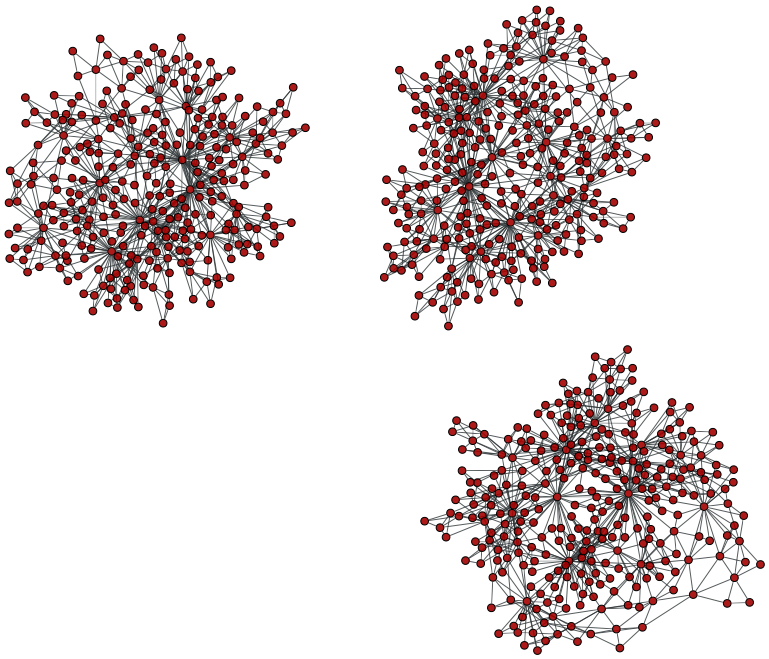
props : list of PropertyMap (page 35) objects

List of propagated properties. This is only returned if *props* is not empty.

Examples

```
>>> g = gt.triangulation(random((300,2)))[0]
>>> ug = gt.graph_union(g, g)
>>> uug = gt.graph_union(g, ug)
>>> pos = gt.sfdp_layout(g)
>>> gt.graph_draw(g, pos=pos, output_size=(300,300), output="graph_original.pdf")
<...>

>>> pos = gt.sfdp_layout(ug)
>>> gt.graph_draw(ug, pos=pos, output_size=(300,300), output="graph_union.pdf")
<...>

>>> pos = gt.sfdp_layout(uug)
>>> gt.graph_draw(uug, pos=pos, output_size=(300,300), output="graph_union2.pdf")
<...>
```





graph_tool.generation.**triangulation**(*points*, *type='simple'*, *periodic=False*)

Generate a 2D or 3D triangulation graph from a given point set.

Parameters **points** : ndarray

Point set for the triangulation. It may be either a N x d array, where N is the number of points, and d is the space dimension (either 2 or 3).

type : string (optional, default: 'simple')

Type of triangulation. May be either 'simple' or 'delaunay'.

periodic : bool (optional, default: False)

If True, periodic boundary conditions will be used. This is parameter is valid only for type="delaunay", and is otherwise ignored.

Returns **triangulation_graph** : Graph (page 28)

The generated graph.

pos : PropertyMap (page 35)

Vertex property map with the Cartesian coordinates.

See Also:

random_graph (page 137) random graph generation

Notes

A triangulation [cgal-triang] (page 230) is a division of the convex hull of a point set into triangles, using only that set as triangle vertices.

In simple triangulations (*type="simple"*), the insertion of a point is done by locating a face that contains the point, and splitting this face into three new faces (the order of insertion

is therefore important). If the point falls outside the convex hull, the triangulation is restored by flips. Apart from the location, insertion takes a time O(1). This bound is only an amortized bound for points located outside the convex hull.

Delaunay triangulations (*type="delaunay"*) have the specific empty sphere property, that is, the circumscribing sphere of each cell of such a triangulation does not contain any other vertex of the triangulation in its interior. These triangulations are uniquely defined except in degenerate cases where five points are co-spherical. Note however that the CGAL implementation computes a unique triangulation even in these cases.

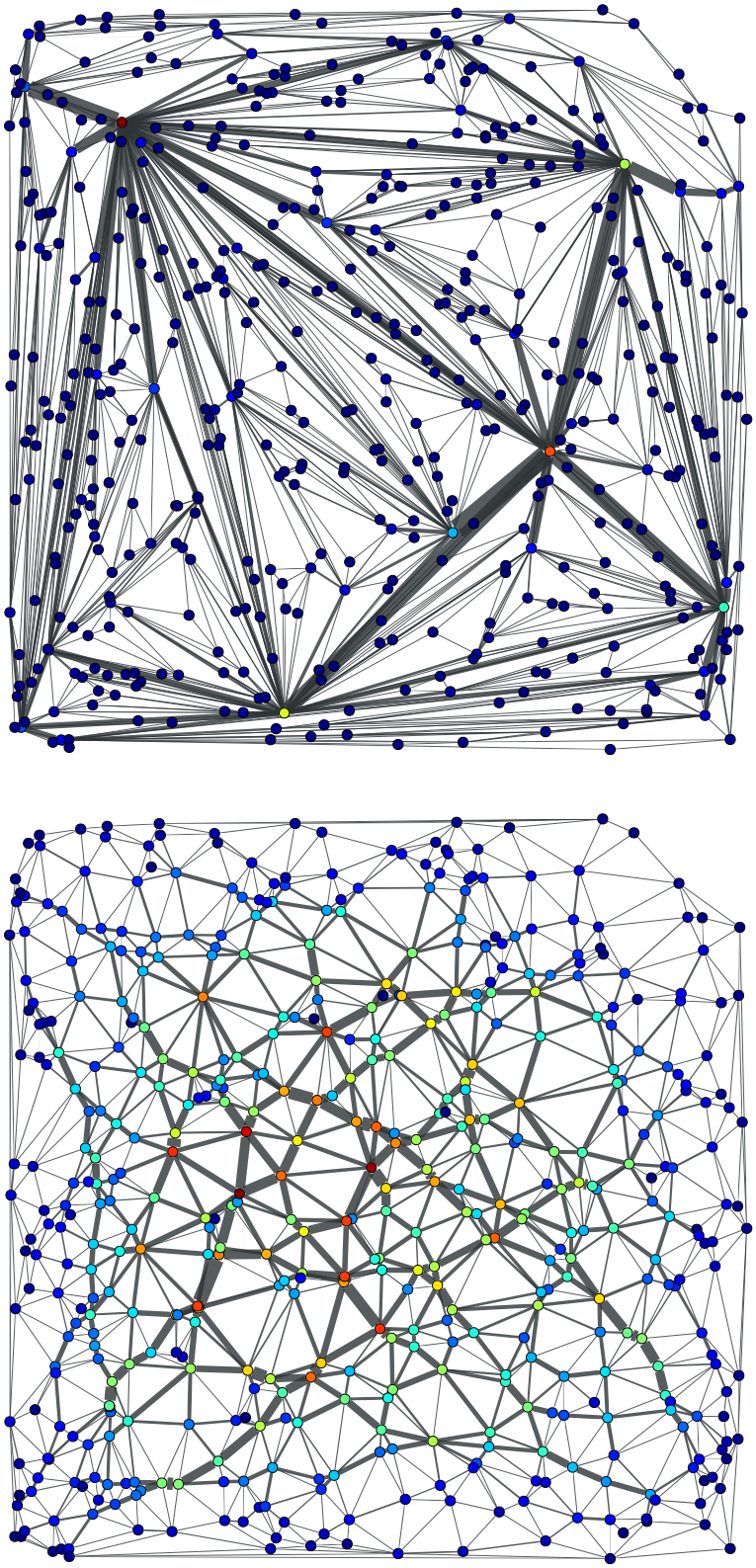
References

[cgal-triang] (page 230)

Examples

```
>>> points = random((500, 2)) * 4
>>> g, pos = gt.triangulation(points)
>>> weight = g.new_edge_property("double") # Edge weights corresponding to
...                                        # Euclidean distances
>>> for e in g.edges():
...     weight[e] = sqrt(sum((array(pos[e.source()]) -
...                           array(pos[e.target()]))**2))
>>> b = gt.betweenness(g, weight=weight)
>>> b[1].a *= 100
>>> gt.graph_draw(g, pos=pos, output_size=(300,300), vertex_fill_color=b[0],
...               edge_pen_width=b[1], output="triang.pdf")
<...>

>>> g, pos = gt.triangulation(points, type="delaunay")
>>> weight = g.new_edge_property("double")
>>> for e in g.edges():
...     weight[e] = sqrt(sum((array(pos[e.source()]) -
...                           array(pos[e.target()]))**2))
>>> b = gt.betweenness(g, weight=weight)
>>> b[1].a *= 120
>>> gt.graph_draw(g, pos=pos, output_size=(300,300), vertex_fill_color=b[0],
...               edge_pen_width=b[1], output="triang-delaunay.pdf")
<...>
```

2D triangulation of random points:



Left: Simple triangulation. *Right:* Delaunay triangulation. The vertex colors and the edge thickness correspond to the weighted betweenness centrality.

graph_tool.generation.**lattice**(*shape*, *periodic=False*)

Generate a N-dimensional square lattice.

Parameters **shape** : list or ndarray

List of sizes in each dimension.

periodic : bool (optional, default: `False`)

If `True`, periodic boundary conditions will be used.

Returns **lattice_graph** : `Graph` (page 28)

The generated graph.

See Also:

`triangulation` (page 155) 2D or 3D triangulation

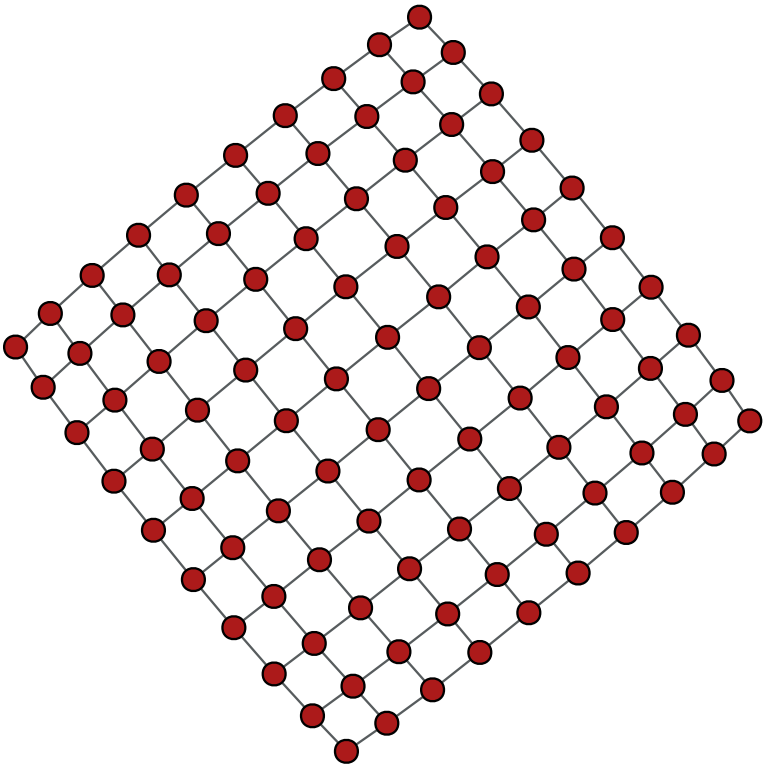
`random_graph` (page 137) random graph generation

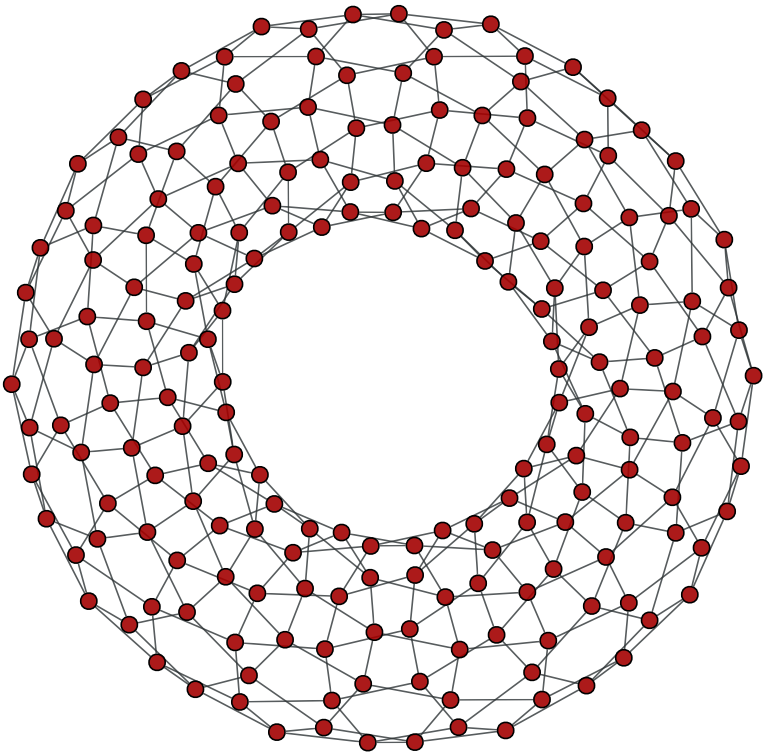
References

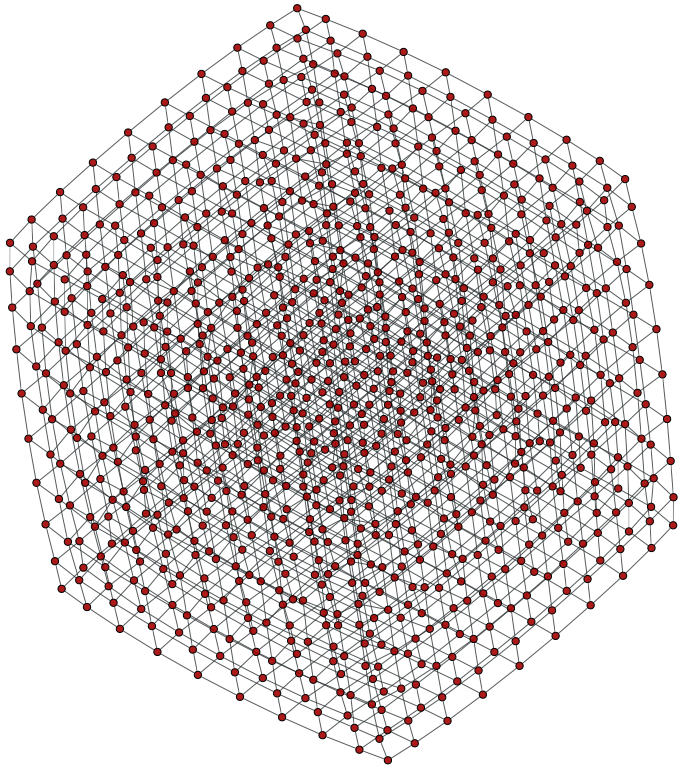
[lattice] (page 230)

Examples

```
>>> g = gt.lattice([10,10])
>>> pos = gt.sfdp_layout(g, cooling_step=0.95, epsilon=1e-2)
>>> gt.graph_draw(g, pos=pos, output_size=(300,300), output="lattice.pdf")
<...>

>>> g = gt.lattice([10,20], periodic=True)
>>> pos = gt.sfdp_layout(g, cooling_step=0.95, epsilon=1e-2)
>>> gt.graph_draw(g, pos=pos, output_size=(300,300), output="lattice_periodic.pdf")
<...>

>>> g = gt.lattice([10,10,10])
>>> pos = gt.sfdp_layout(g, cooling_step=0.95, epsilon=1e-2)
>>> gt.graph_draw(g, pos=pos, output_size=(300,300), output="lattice_3d.pdf")
<...>
```







Left: 10x10 2D lattice. *Middle:* 10x20 2D periodic lattice (torus). *Right:* 10x10x10 3D lattice.

graph_tool.generation.**complete_graph**(*N*, *self_loops=False*, *directed=False*)

Generate complete graph.

Parameters **N** : int

Number of vertices.

self_loops : bool (optional, default: False)

If True, self-loops are included.

directed : bool (optional, default: False)

If True, a directed graph is generated.

Returns **complete_graph** : Graph (page 28)

A complete graph.

References

[complete] (page **??**)

Examples

```
>>> g = gt.complete_graph(30)
>>> pos = gt.sfdp_layout(g, cooling_step=0.95, epsilon=1e-2)
>>> gt.graph_draw(g, pos=pos, output_size=(300,300), output="complete.pdf")
<...>
```

graph_tool.generation.**circular_graph**(*N*, *k=1*, *self_loops=False*, *directed=False*)

Generate a circular graph.

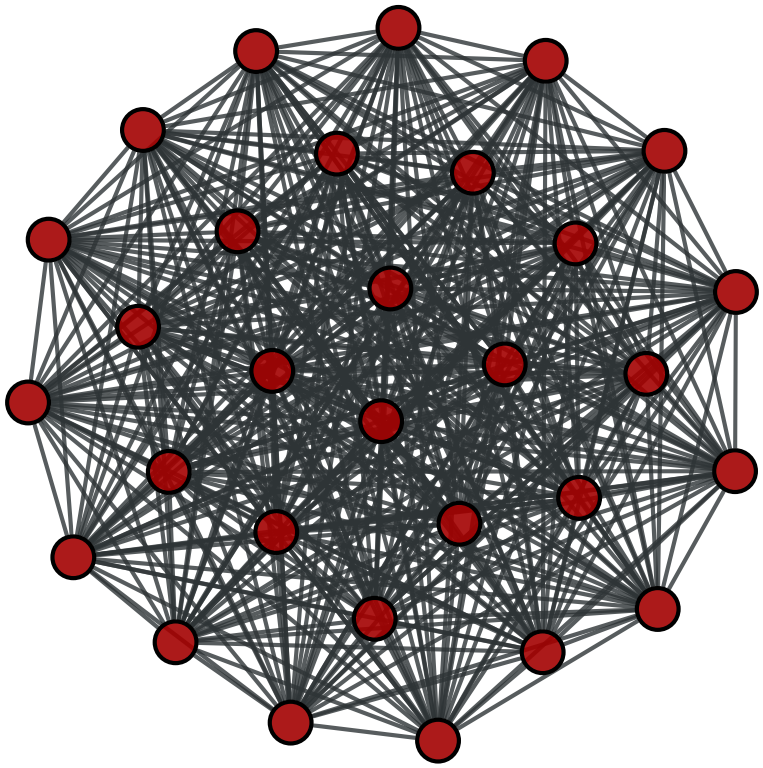


Figure 3.39: A complete graph with $N = 30$ vertices.

Parameters **N** : `int`

Number of vertices.

k : `int` (optional, default: `True`)

Number of nearest neighbours to be connected.

self_loops : bool (optional, default: `False`)

If `True`, self-loops are included.

directed : bool (optional, default: `False`)

If `True`, a directed graph is generated.

Returns **circular_graph** : `Graph` (page 28)

A circular graph.

Examples

```
>>> g = gt.circular_graph(30, 2)
>>> pos = gt.sfdp_layout(g, cooling_step=0.95)
>>> gt.graph_draw(g, pos=pos, output_size=(300,300), output="circular.pdf")
<...>
```

graph_tool.generation.**geometric_graph**(*points*, *radius*, *ranges=None*)

Generate a geometric network form a set of N-dimensional points.

Parameters **points** : list or `ndarray`

List of points. This must be a two-dimensional array, where the rows are coordinates in a N-dimensional space.

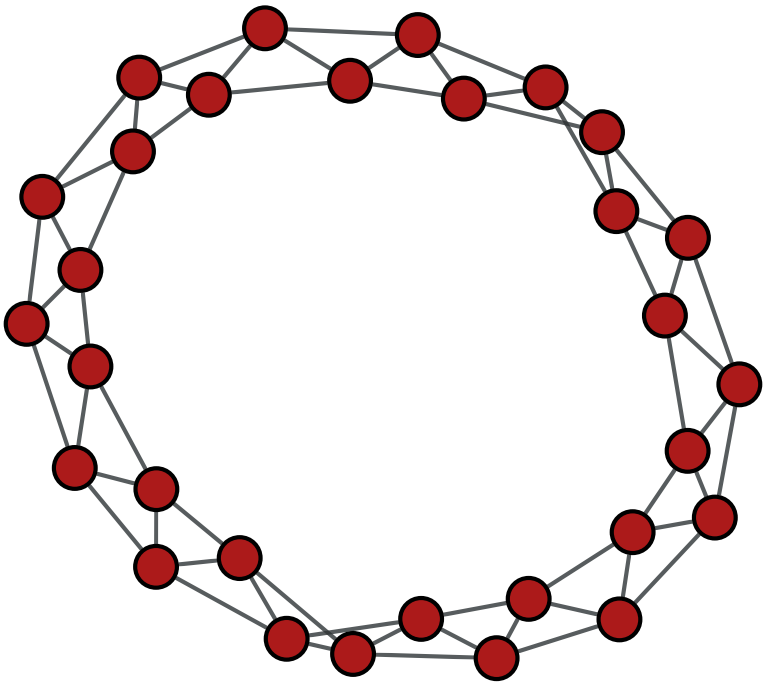


Figure 3.40: A circular graph with $N = 30$ vertices, and $k = 2$.

radius : float

Pairs of points with an euclidean distance lower than this parameters will be connected.

ranges : list or `ndarray` (optional, default: `None`)

If provided, periodic boundary conditions will be assumed, and the values of this parameter it will be used as the ranges in all dimensions. It must be a two-dimensional array, where each row will cointain the lower and upper bound of each dimension.

Returns **geometric_graph** : `Graph` (page 28)

The generated graph.

pos : `PropertyMap` (page 35)

A vertex property map with the position of each vertex.

See Also:

`triangulation` (page 155) 2D or 3D triangulation

`random_graph` (page 137) random graph generation

`lattice` (page 157) N-dimensional square lattice

Notes

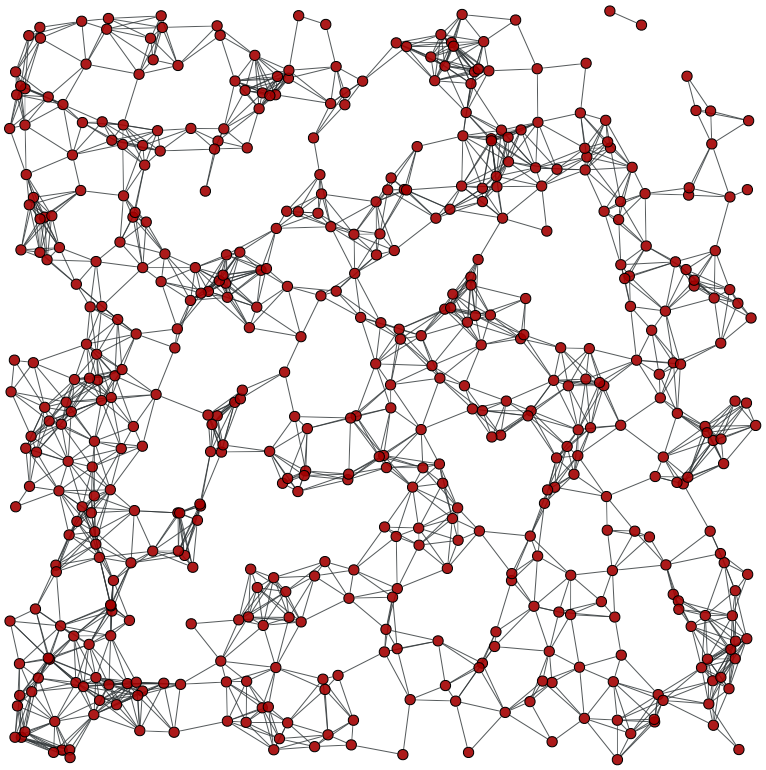
A geometric graph [geometric-graph] (page 230) is generated by connecting points embedded in a N-dimensional euclidean space which are at a distance equal to or smaller than a given radius.

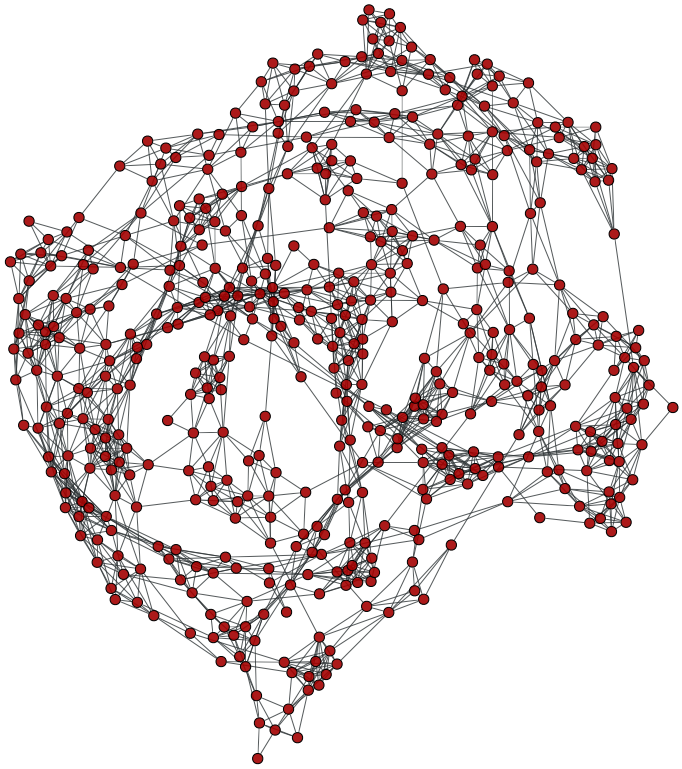
References

[geometric-graph] (page 230)

Examples

```
>>> points = random((500, 2)) * 4
>>> g, pos = gt.geometric_graph(points, 0.3)
>>> gt.graph_draw(g, pos=pos, output_size=(300,300), output="geometric.pdf")
<...>

>>> g, pos = gt.geometric_graph(points, 0.3, [(0,4), (0,4)])
>>> pos = gt.graph_draw(g, output_size=(300,300), output="geometric_periodic.pdf")
```





***Left:* Geometric network with random points. *Right:* Same network, but** with periodic boundary conditions.

graph_tool.generation.**price_network**(*N*, *m=1*, *c=None*, *gamma=1*, *directed=True*, *seed_graph=None*)

A generalized version of Price's – or Barabási-Albert if undirected – preferential attachment network model.

Parameters **N** : int

Size of the network.

m : int (optional, default: `1`)

Out-degree of newly added vertices.

c : float (optional, default: `1 if directed == True else 0`)

Constant factor added to the probability of a vertex receiving an edge (see notes below).

gamma : float (optional, default: `1`)

Preferential attachment power (see notes below).

directed : bool (optional, default: `True`)

If `True`, a Price network is generated. If `False`, a Barabási-Albert network is generated.

seed_graph : `Graph` (page 28) (optional, default: `None`)

If provided, this graph will be used as the starting point of the algorithm.

Returns **price_graph** : `Graph` (page 28)

The generated graph.

See Also:

triangulation (page 155) 2D or 3D triangulation

random_graph (page 137) random graph generation

lattice (page 157) N-dimensional square lattice

geometric_graph (page 160) N-dimensional geometric network

Notes

The (generalized) [price] (page 231) network is either a directed or undirected graph (the latter is called a Barabási-Albert network), generated dynamically by at each step adding a new vertex, and connecting it to m other vertices, chosen with probability π defined as:

$$\pi \propto k^{\gamma} + c$$

where k is the in-degree of the vertex (or simply the degree in the undirected case). If $\gamma = 1$, the tail of resulting in-degree distribution of the directed case is given by

$$P_{k_{\text{in}}} \sim k_{\text{in}}^{-(2+c/m)},$$

or for the undirected case

$$P_k \sim k^{-(3+c/m)}.$$

However, if $\gamma \neq 1$, the in-degree distribution is not scale-free (see [dorogovtsev-evolution] (page 231) for details).

Note that if *seed_graph* is not given, the algorithm will *always* start with one node if $c > 0$, or with two nodes with a link between them otherwise. If $m > 1$, the degree of the newly added vertices will be vary dynamically as $m'(t) = \min(m, N(t))$, where $N(t)$ is the number of vertices added so far. If this behaviour is undesired, a proper seed graph with $N \geq m$ vertices must be provided.

This algorithm runs in $O(N \log N)$ time.

References

[yule] (page 230), [price] (page 231), [barabasi-albert] (page 231), [dorogovtsev-evolution] (page 231)

Examples

```
>>> g = gt.price_network(100000)
>>> gt.graph_draw(g, pos=gt.sfdp_layout(g, epsilon=1e-2, cooling_step=0.95),
...               vertex_fill_color=g.vertex_index, vertex_size=2,
...               edge_pen_width=1, output="price-network.png")
<...>
>>> g = gt.price_network(100000, c=0.1)
>>> gt.graph_draw(g, pos=gt.sfdp_layout(g, epsilon=1e-2, cooling_step=0.95),
...               vertex_fill_color=g.vertex_index, vertex_size=2,
...               edge_pen_width=1, output="price-network-broader.png")
<...>
```

3.2.9 `graph_tool.run_action` - Inline C++ code embedding

This module implements support for automatic ad-hoc code embedding into graph-tool.

Summary

Figure 3.41: Price network with $N = 10^5$ nodes and $c = 1$. The colors represent the order in which vertices were added.

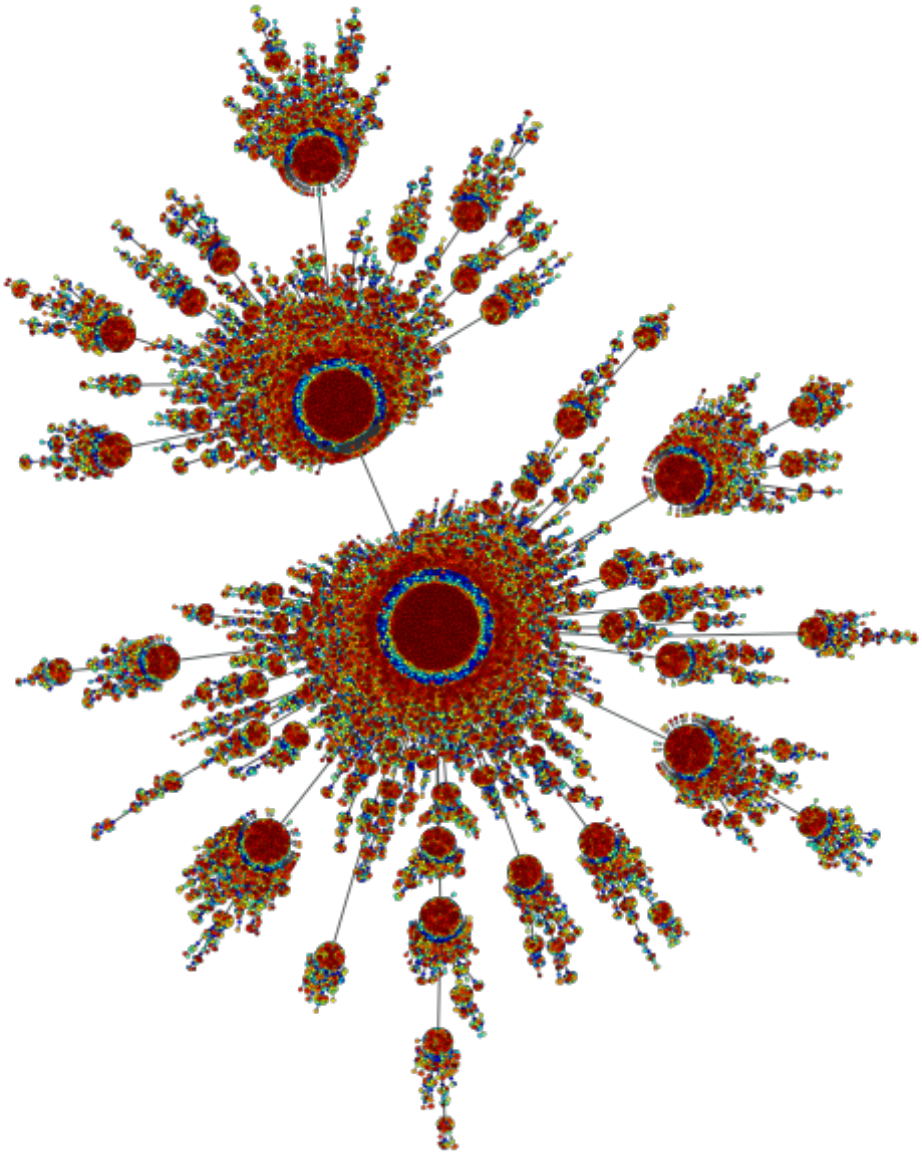


Figure 3.42: Price network with $N = 10^5$ nodes and $c = 0.1$. The colors represent the order in which vertices were added.

`inline` (page 166)	Compile (if necessary) and run the C++ code specified by 'code', using `weave`.

Contents

`graph_tool.run_action.`**`inline`**(*code*, *arg_names=None*, *local_dict=None*, *global_dict=None*, *force=False*, *compiler='gcc'*, *verbose=False*, *auto_downcast=1*, *support_code=''*, *libraries=None*, *library_dirs=None*, *extra_compile_args=None*, *runtime_library_dirs=None*, *extra_objects=None*, *extra_link_args=None*, *mask_ret=None*, *debug=False*)

Compile (if necessary) and run the C++ code specified by 'code', using `weave`. The (possibly modified) variables in 'arg_names' are returned.

See `scipy.weave.inline()` for detailed parameter documentation.

Notes

Graphs and property maps are automatically converted to appropriate [Boost] (page 231) graph types. For convenience, the graph types are automatically typedef'd to *${name}_graph_t*, where ${name} is the graph's variable name passed to *arg_names*. Property map types are typedef'd to *vprop_${val_type}_t*, *eprop_${val_type}_t* or *gprop_${val_type}_t*, for vertex, edge or graph properties, where ${val_type} specifies the value type (e.g. int, bool, double, etc.). In the case of vector types, the "<" and ">" symbols are replaced by underscores ("_").

References

[Boost] (page 231), [Weave] (page 231)

Examples

```
>>> from numpy.random import seed
>>> seed(42)
>>> g = gt.random_graph(100, lambda: (3, 3))
>>> nv = 0
>>> ret = gt.inline("nv = num_vertices(g);", ['g', 'nv'])
>>> print(ret["nv"])
100
>>> prop = g.new_vertex_property("vector<double>")
>>> prop[g.vertex(0)] = [1.0, 4.2]
>>> val = 0.0
>>> ret = gt.inline("val = prop[vertex(0,g)][1];", ['g', 'prop', 'val'])
>>> print(ret["val"])
4.2
```

3.2.10 `graph_tool.search` - Search algorithms

This module includes several search algorithms, which are customizable to arbitrary purposes. It is mostly a wrapper around the Visitor interface of the Boost Graph Library, and the respective search functions.

Summary

bfs_search (page 169)	Breadth-first traversal of a directed or undirected graph.
dfs_search (page 171)	Depth-first traversal of a directed or undirected graph.
dijkstra_search (page 174)	Dijsktra traversal of a directed or undirected graph, with non-negative
bellman_ford_search (page 177)	Bellman-Ford traversal of a directed or undirected graph, with negati
astar_search (page 181)	Heuristic A^* search on a weighted, directed or undirected graph for th
BFSVisitor (page 167)	A visitor object that is invoked at the event-points inside the bfs_sea
DFSVisitor (page 170)	A visitor object that is invoked at the event-points inside the dfs_sea
DijkstraVisitor (page 173)	A visitor object that is invoked at the event-points inside the dijkstr
BellmanFordVisitor (page 177)	A visitor object that is invoked at the event-points inside the bellman
AStarVisitor (page 180)	A visitor object that is invoked at the event-points inside the astar_s
StopSearch (page 186)	If this exception is raised from inside any search visitor object, the se

Examples

In this module, most documentation examples will make use of the network `search_example.xml`, shown below.

```
>>> gt.seed_rng(42)
>>> g = gt.load_graph("search_example.xml")
>>> name = g.vertex_properties["name"]
>>> weight = g.edge_properties["weight"]
>>> pos = gt.graph_draw(g, vertex_text=name, vertex_font_size=12, vertex_shape="double_circle",
...                     vertex_fill_color="#729fcf", vertex_pen_width=3,
...                     edge_pen_width=weight, output="search_example.pdf")
```

Contents

class graph_tool.search.**BFSVisitor**

A visitor object that is invoked at the event-points inside the bfs_search() (page 169) algorithm. By default, it performs no action, and should be used as a base class in order to be useful.

initialize_vertex(*self*, *u*)

This is invoked on every vertex of the graph before the start of the graph search.

discover_vertex(*self*, *u*)

This is invoked when a vertex is encountered for the first time.

examine_vertex(*self*, *u*)

This is invoked on a vertex as it is popped from the queue. This happens immediately before examine_edge() is invoked on each of the out-edges of vertex u.

examine_edge(*self*, *e*)

This is invoked on every out-edge of each vertex after it is discovered.

tree_edge(*self*, *e*)

This is invoked on each edge as it becomes a member of the edges that form the search tree.

non_tree_edge(*self*, *e*)

This is invoked on back or cross edges for directed graphs and cross edges for undirected graphs.

gray_target(*self*, *e*)

This is invoked on the subset of non-tree edges whose target vertex is colored gray at the time of examination. The color gray indicates that the vertex is currently in the queue.

black_target(*self*, *e*)

This is invoked on the subset of non-tree edges whose target vertex is colored black

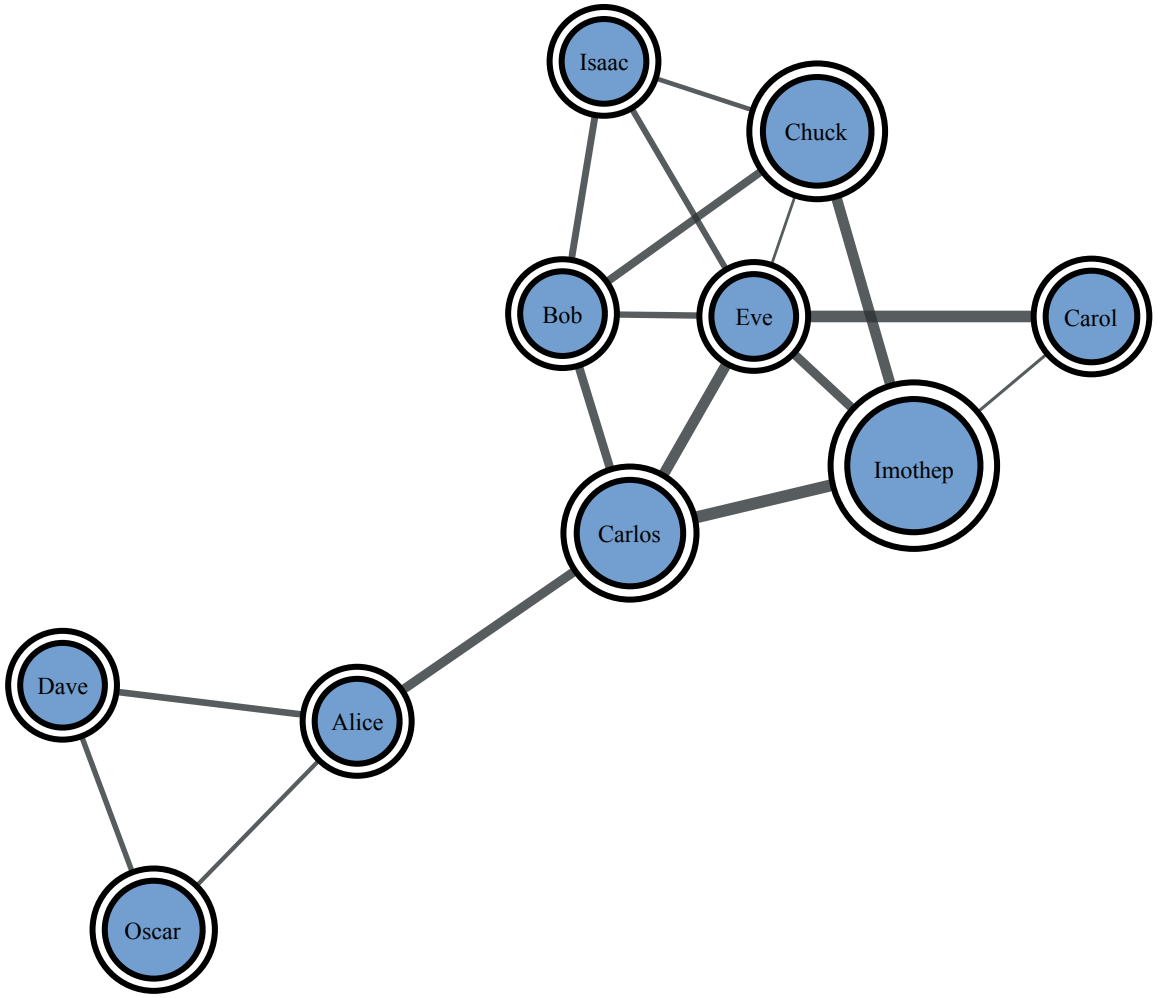


Figure 3.43: This is the network used in the examples below. The width of the edges correspond to the values of the "weight" property map.

at the time of examination. The color black indicates that the vertex has been removed from the queue.

finish_vertex(*self*, *u*)
This invoked on a vertex after all of its out edges have been added to the search tree and all of the adjacent vertices have been discovered (but before the out-edges of the adjacent vertices have been examined).

graph_tool.search.**bfs_search**(*g*, *source*, *visitor=<graph_tool.search.BFSVisitor object at 0x7fc155a02ad0>*)

Breadth-first traversal of a directed or undirected graph.

Parameters **g** : Graph (page 28)

Graph to be used.

source : Vertex (page 33)

Source vertex.

visitor : BFSVisitor (page 167) (optional, default: BFSVisitor())

A visitor object that is invoked at the event points inside the algorithm. This should be a subclass of BFSVisitor (page 167).

See Also:

dfs_search (page 171) Depth-first search

dijkstra_search (page 174) Dijkstra's search algorithm

astar_search (page 181) A^* heuristic search algorithm

Notes

A breadth-first traversal visits vertices that are closer to the source before visiting vertices that are further away. In this context "distance" is defined as the number of edges in the shortest path from the source vertex.

The time complexity is $O(V + E)$.

The pseudo-code for the BFS algorithm is listed below, with the annotated event points, for which the given visitor object will be called with the appropriate method.

```
BFS(G, source)
 for each vertex u in V[G]       initialize vertex u
   color[u] := WHITE
   d[u] := infinity
 end for
 color[source] := GRAY
 d[source] := 0
 ENQUEUE(Q, source)              discover vertex source
 while (Q != Ø)
   u := DEQUEUE(Q)               examine vertex u
   for each vertex v in Adj[u]   examine edge (u,v)
     if (color[v] = WHITE)       (u,v) is a tree edge
       color[v] := GRAY
       ENQUEUE(Q, v)             discover vertex v
     else                        (u,v) is a non-tree edge
       if (color[v] = GRAY)
         ...                     (u,v) has a gray target
       else
         ...                     (u,v) has a black target
   end for
   color[u] := BLACK             finish vertex u
 end while
```

References

[bfs] (page **??**), [bfs-bgl] (page 231), [bfs-wikipedia] (page 231)

Examples

We must define what should be done during the search by subclassing `BFSVisitor` (page 167), and specializing the appropriate methods. In the following we will keep track of the distance from the root, and the predecessor tree.

```
class VisitorExample(gt.BFSVisitor):

    def __init__(self, name, pred, dist):
        self.name = name
        self.pred = pred
        self.dist = dist

    def discover_vertex(self, u):
        print("-->", self.name[u], "has been discovered!")

    def examine_vertex(self, u):
        print(self.name[u], "has been examined...")

    def tree_edge(self, e):
        self.pred[e.target()] = int(e.source())
        self.dist[e.target()] = self.dist[e.source()] + 1
```

With the above class defined, we can perform the BFS search as follows.

```
>>> dist = g.new_vertex_property("int")
>>> pred = g.new_vertex_property("int")
>>> gt.bfs_search(g, g.vertex(0), VisitorExample(name, pred, dist))
--> Bob has been discovered!
Bob has been examined...
--> Eve has been discovered!
--> Chuck has been discovered!
--> Carlos has been discovered!
--> Isaac has been discovered!
Eve has been examined...
--> Imothep has been discovered!
--> Carol has been discovered!
Chuck has been examined...
Carlos has been examined...
--> Alice has been discovered!
Isaac has been examined...
Imothep has been examined...
Carol has been examined...
Alice has been examined...
--> Oscar has been discovered!
--> Dave has been discovered!
Oscar has been examined...
Dave has been examined...
>>> print(dist.a)
[0 2 2 1 1 3 1 1 3 2]
>>> print(pred.a)
[0 3 6 0 0 1 0 0 1 6]
```

class graph_tool.search.**DFSVisitor**
A visitor object that is invoked at the event-points inside the `dfs_search()` (page 171) algorithm. By default, it performs no action, and should be used as a base class in order to be useful.

initialize_vertex(*self*, *u*)
This is invoked on every vertex of the graph before the start of the graph search.

start_vertex(*self*, *u*)
This is invoked on the source vertex once before the start of the search.

discover_vertex(*self*, *u*)
This is invoked when a vertex is encountered for the first time.

examine_edge(*self*, *e*)
This is invoked on every out-edge of each vertex after it is discovered.

tree_edge(*self*, *e*)
This is invoked on each edge as it becomes a member of the edges that form the search tree.

back_edge(*self*, *e*)
This is invoked on the back edges in the graph. For an undirected graph there is some ambiguity between tree edges and back edges since the edge (u,v) and (v,u) are the same edge, but both the `tree_edge()` (page 171) and `back_edge()` functions will be invoked. One way to resolve this ambiguity is to record the tree edges, and then disregard the back-edges that are already marked as tree edges. An easy way to record tree edges is to record predecessors at the tree_edge event point.

forward_or_cross_edge(*self*, *e*)
This is invoked on forward or cross edges in the graph. In an undirected graph this method is never called.

finish_vertex(*self*, *e*)
This is invoked on vertex u after finish_vertex has been called for all the vertices in the DFS-tree rooted at vertex u. If vertex u is a leaf in the DFS-tree, then the `finish_vertex()` function is called on u after all the out-edges of u have been examined.

graph_tool.search.**dfs_search**(*g*, *source*, *visitor=<graph_tool.search.DFSVisitor object at 0x7fc145938550>*)

Depth-first traversal of a directed or undirected graph.

Parameters **g** : `Graph` (page 28)

Graph to be used.

source : `Vertex` (page 33)

Source vertex.

visitor : `DFSVisitor` (page 170) (optional, default: `DFSVisitor()`)

A visitor object that is invoked at the event points inside the algorithm. This should be a subclass of `DFSVisitor` (page 170).

See Also:

bfs_search **(page 169)** Breadth-first search

dijkstra_search **(page 174)** Dijkstra's search algorithm

astar_search **(page 181)** A^* heuristic search algorithm

Notes

When possible, a depth-first traversal chooses a vertex adjacent to the current vertex to visit next. If all adjacent vertices have already been discovered, or there are no adjacent vertices, then the algorithm backtracks to the last vertex that had undiscovered neighbors. Once all reachable vertices have been visited, the algorithm selects from any

remaining undiscovered vertices and continues the traversal. The algorithm finishes when all vertices have been visited.

The time complexity is $O(V + E)$.

The pseudo-code for the DFS algorithm is listed below, with the annotated event points, for which the given visitor object will be called with the appropriate method.

```
DFS(G)
  for each vertex u in V
    color[u] := WHITE                 initialize vertex u
  end for
  time := 0
  call DFS-VISIT(G, source)           start vertex s

DFS-VISIT(G, u)
  color[u] := GRAY                    discover vertex u
  for each v in Adj[u]                examine edge (u,v)
    if (color[v] = WHITE)             (u,v) is a tree edge
      call DFS-VISIT(G, v)
    else if (color[v] = GRAY)         (u,v) is a back edge
      ...
    else if (color[v] = BLACK)        (u,v) is a cross or forward edge
      ...
  end for
  color[u] := BLACK                   finish vertex u
```

References

[dfs-bgl] (page 231), [dfs-wikipedia] (page 231)

Examples

We must define what should be done during the search by subclassing `DFSVisitor` (page 170), and specializing the appropriate methods. In the following we will keep track of the discover time, and the predecessor tree.

```
class VisitorExample(gt.DFSVisitor):

    def __init__(self, name, pred, time):
        self.name = name
        self.pred = pred
        self.time = time
        self.last_time = 0

    def discover_vertex(self, u):
        print("-->", self.name[u], "has been discovered!")
        self.time[u] = self.last_time
        self.last_time += 1

    def examine_edge(self, e):
        print("edge (%s, %s) has been examined..." % \
              (self.name[e.source()], self.name[e.target()]))

    def tree_edge(self, e):
        self.pred[e.target()] = int(e.source())
```

With the above class defined, we can perform the DFS search as follows.

```
>>> time = g.new_vertex_property("int")
>>> pred = g.new_vertex_property("int")
```

```
>>> gt.dfs_search(g, g.vertex(0), VisitorExample(name, pred, time))
--> Bob has been discovered!
edge (Bob, Eve) has been examined...
--> Eve has been discovered!
edge (Eve, Isaac) has been examined...
--> Isaac has been discovered!
edge (Isaac, Bob) has been examined...
edge (Isaac, Chuck) has been examined...
--> Chuck has been discovered!
edge (Chuck, Eve) has been examined...
edge (Chuck, Isaac) has been examined...
edge (Chuck, Imothep) has been examined...
--> Imothep has been discovered!
edge (Imothep, Carol) has been examined...
--> Carol has been discovered!
edge (Carol, Eve) has been examined...
edge (Carol, Imothep) has been examined...
edge (Imothep, Carlos) has been examined...
--> Carlos has been discovered!
edge (Carlos, Eve) has been examined...
edge (Carlos, Imothep) has been examined...
edge (Carlos, Bob) has been examined...
edge (Carlos, Alice) has been examined...
--> Alice has been discovered!
edge (Alice, Oscar) has been examined...
--> Oscar has been discovered!
edge (Oscar, Alice) has been examined...
edge (Oscar, Dave) has been examined...
--> Dave has been discovered!
edge (Dave, Oscar) has been examined...
edge (Dave, Alice) has been examined...
edge (Alice, Dave) has been examined...
edge (Alice, Carlos) has been examined...
edge (Imothep, Chuck) has been examined...
edge (Imothep, Eve) has been examined...
edge (Chuck, Bob) has been examined...
edge (Isaac, Eve) has been examined...
edge (Eve, Imothep) has been examined...
edge (Eve, Bob) has been examined...
edge (Eve, Carol) has been examined...
edge (Eve, Carlos) has been examined...
edge (Eve, Chuck) has been examined...
edge (Bob, Chuck) has been examined...
edge (Bob, Carlos) has been examined...
edge (Bob, Isaac) has been examined...
>>> print(time.a)
[0 7 5 6 3 9 1 2 8 4]
>>> print(pred.a)
[0 3 9 9 7 8 0 6 1 4]
```

class graph_tool.search.**DijkstraVisitor**

A visitor object that is invoked at the event-points inside the dijkstra_search() (page 174) algorithm. By default, it performs no action, and should be used as a base class in order to be useful.

initialize_vertex(*self*, *u*)

This is invoked on every vertex of the graph before the start of the graph search.

examine_vertex(*self*, *u*)

This is invoked on a vertex as it is popped from the queue. This happens immediately before examine_edge() is invoked on each of the out-edges of vertex u.

examine_edge(*self*, *e*)

This is invoked on every out-edge of each vertex after it is discovered.

discover_vertex(*self*, *u*)
This is invoked when a vertex is encountered for the first time.

edge_relaxed(*self*, *e*)
Upon examination, if the following condition holds then the edge is relaxed (its distance is reduced), and this method is invoked.

```
(u, v) = tuple(e)
assert(compare(combine(d[u], weight[e]), d[v]))
```

edge_not_relaxed(*self*, *e*)
Upon examination, if the edge is not relaxed (see `edge_relaxed()`) then this method is invoked.

finish_vertex(*self*, *u*)
This invoked on a vertex after all of its out edges have been added to the search tree and all of the adjacent vertices have been discovered (but before their out-edges have been examined).

graph_tool.search.**dijkstra_search**(*g*, *source*, *weight*, *visitor=<graph_tool.search.DijkstraVisitor object at 0x7fc145938710>*, *dist_map=None*, *pred_map=None*, *combine=<function <lambda> at 0x7fc14594b9e0>*, *compare=<function <lambda> at 0x7fc14594be60>*, *zero=0*, *infinity=inf*)

Dijsktra traversal of a directed or undirected graph, with non-negative weights.

Parameters **g** : `Graph` (page 28)

Graph to be used.

source : `Vertex` (page 33)

Source vertex.

weight : `PropertyMap` (page 35)

Edge property map with weight values.

visitor : `DijkstraVisitor` (page 173) (optional, default: `DijkstraVisitor()`)

A visitor object that is invoked at the event points inside the algorithm. This should be a subclass of `DijkstraVisitor` (page 173).

dist_map : `PropertyMap` (page 35) (optional, default: `None`)

A vertex property map where the distances from the source will be stored.

pred_map : `PropertyMap` (page 35) (optional, default: `None`)

A vertex property map where the predecessor map will be stored (must have value type "int").

combine : binary function (optional, default: `lambda a, b: a + b`)

This function is used to combine distances to compute the distance of a path.

compare : binary function (optional, default: `lambda a, b: a < b`)

This function is use to compare distances to determine which vertex is closer to the source vertex.

zero : int or float (optional, default: `0`)

Value assumed to correspond to a distance of zero by the combine and compare functions.

infinity : int or float (optional, default: `float('inf')`)

Value assumed to correspond to a distance of infinity by the combine and compare functions.

Returns **dist_map** : `PropertyMap` (page 35)

A vertex property map with the computed distances from the source.

pred_map : `PropertyMap` (page 35)

A vertex property map with the predecessor tree.

See Also:

bfs_search (page 169) Breadth-first search

dfs_search (page 171) Depth-first search

astar_search (page 181) A^* heuristic search algorithm

Notes

Dijkstra's algorithm finds all the shortest paths from the source vertex to every other vertex by iteratively "growing" the set of vertices S to which it knows the shortest path. At each step of the algorithm, the next vertex added to S is determined by a priority queue. The queue contains the vertices in V - S prioritized by their distance label, which is the length of the shortest path seen so far for each vertex. The vertex u at the top of the priority queue is then added to S, and each of its out-edges is relaxed: if the distance to u plus the weight of the out-edge (u,v) is less than the distance label for v then the estimated distance for vertex v is reduced. The algorithm then loops back, processing the next vertex at the top of the priority queue. The algorithm finishes when the priority queue is empty.

The time complexity is $O(V \log V)$.

The pseudo-code for Dijkstra's algorithm is listed below, with the annotated event points, for which the given visitor object will be called with the appropriate method.

```
DIJKSTRA(G, source, weight)
  for each vertex u in V                    initialize vertex u
    d[u] := infinity
    p[u] := u
  end for
  d[source] := 0
  INSERT(Q, source)                         discover vertex s
  while (Q != Ø)
    u := EXTRACT-MIN(Q)                     examine vertex u
    for each vertex v in Adj[u]             examine edge (u,v)
      if (weight[(u,v)] + d[u] < d[v])      edge (u,v) relaxed
        d[v] := weight[(u,v)] + d[u]
        p[v] := u
        DECREASE-KEY(Q, v)
      else                                  edge (u,v) not relaxed
        ...
      if (d[v] was originally infinity)
        INSERT(Q, v)                        discover vertex v
    end for                                 finish vertex u
  end while
  return d
```

References

[dijkstra] (page **??**), [dijkstra-bgl] (page 231), [dijkstra-wikipedia] (page 231)

Examples

We must define what should be done during the search by subclassing `DijkstraVisitor` (page 173), and specializing the appropriate methods. In the following we will keep track of the discover time, and the predecessor tree.

```
class VisitorExample(gt.DijkstraVisitor):

    def __init__(self, name, time):
        self.name = name
        self.time = time
        self.last_time = 0

    def discover_vertex(self, u):
        print("-->", self.name[u], "has been discovered!")
        self.time[u] = self.last_time
        self.last_time += 1

    def examine_edge(self, e):
        print("edge (%s, %s) has been examined..." % \
              (self.name[e.source()], self.name[e.target()]))

    def edge_relaxed(self, e):
        print("edge (%s, %s) has been relaxed..." % \
              (self.name[e.source()], self.name[e.target()]))
```

With the above class defined, we can perform the Dijkstra search as follows.

```
>>> time = g.new_vertex_property("int")
>>> dist, pred = gt.dijkstra_search(g, g.vertex(0), weight, VisitorExample(name, time))
--> Bob has been discovered!
edge (Bob, Eve) has been examined...
edge (Bob, Eve) has been relaxed...
--> Eve has been discovered!
edge (Bob, Chuck) has been examined...
edge (Bob, Chuck) has been relaxed...
--> Chuck has been discovered!
edge (Bob, Carlos) has been examined...
edge (Bob, Carlos) has been relaxed...
--> Carlos has been discovered!
edge (Bob, Isaac) has been examined...
edge (Bob, Isaac) has been relaxed...
--> Isaac has been discovered!
edge (Eve, Isaac) has been examined...
edge (Eve, Imothep) has been examined...
edge (Eve, Imothep) has been relaxed...
--> Imothep has been discovered!
edge (Eve, Bob) has been examined...
edge (Eve, Carol) has been examined...
edge (Eve, Carol) has been relaxed...
--> Carol has been discovered!
edge (Eve, Carlos) has been examined...
edge (Eve, Chuck) has been examined...
edge (Isaac, Bob) has been examined...
edge (Isaac, Chuck) has been examined...
edge (Isaac, Eve) has been examined...
edge (Chuck, Eve) has been examined...
edge (Chuck, Isaac) has been examined...
```

```
edge (Chuck, Imothep) has been examined...
edge (Chuck, Bob) has been examined...
edge (Carlos, Eve) has been examined...
edge (Carlos, Imothep) has been examined...
edge (Carlos, Bob) has been examined...
edge (Carlos, Alice) has been examined...
edge (Carlos, Alice) has been relaxed...
--> Alice has been discovered!
edge (Imothep, Carol) has been examined...
edge (Imothep, Carlos) has been examined...
edge (Imothep, Chuck) has been examined...
edge (Imothep, Eve) has been examined...
edge (Alice, Oscar) has been examined...
edge (Alice, Oscar) has been relaxed...
--> Oscar has been discovered!
edge (Alice, Dave) has been examined...
edge (Alice, Dave) has been relaxed...
--> Dave has been discovered!
edge (Alice, Carlos) has been examined...
edge (Carol, Eve) has been examined...
edge (Carol, Imothep) has been examined...
edge (Oscar, Alice) has been examined...
edge (Oscar, Dave) has been examined...
edge (Dave, Oscar) has been examined...
edge (Dave, Alice) has been examined...
>>> print(time.a)
[0 7 6 3 2 9 1 4 8 5]
>>> print(pred.a)
[0 3 6 0 0 1 0 0 1 6]
>>> print(dist.a)
[  0.           8.91915887   9.27141329   4.29277116   4.02118246
  12.23513866   3.23790211   3.45487436  11.04391549   7.74858396]
```

class graph_tool.search.**BellmanFordVisitor**

A visitor object that is invoked at the event-points inside the bellman_ford_search() (page 177) algorithm. By default, it performs no action, and should be used as a base class in order to be useful.

examine_edge(*self*, *e*)

This is invoked on every edge in the graph |V| times.

edge_relaxed(*self*, *e*)

This is invoked when the distance label for the target vertex is decreased. The edge (u,v) that participated in the last relaxation for vertex v is an edge in the shortest paths tree.

edge_not_relaxed(*self*, *e*)

This is invoked if the distance label for the target vertex is not decreased.

edge_minimized(*self*, *e*)

This is invoked during the second stage of the algorithm, during the test of whether each edge was minimized. If the edge is minimized then this function is invoked.

edge_not_minimized(*self*, *e*)

This is invoked during the second stage of the algorithm, during the test of whether each edge was minimized. If the edge was not minimized, this function is invoked. This happens when there is a negative cycle in the graph.

graph_tool.search.**bellman_ford_search**(*g*, *source*, *weight*, *visitor=<graph_tool.search.BellmanFordVisitor object at 0x7fc145948610>*, *dist_map=None*, *pred_map=None*, *combine=<function <lambda> at 0x7fc14594bf80>*, *compare=<function <lambda> at 0x7fc14594c320>*, *zero=0*, *infinity=inf*)

Bellman-Ford traversal of a directed or undirected graph, with negative weights.

Parameters **g** : Graph (page 28)

Graph to be used.

source : Vertex (page 33)

Source vertex.

weight : PropertyMap (page 35)

Edge property map with weight values.

visitor : DijkstraVisitor (page 173) (optional, default: DijkstraVisitor())

A visitor object that is invoked at the event points inside the algorithm. This should be a subclass of DijkstraVisitor (page 173).

dist_map : PropertyMap (page 35) (optional, default: None)

A vertex property map where the distances from the source will be stored.

pred_map : PropertyMap (page 35) (optional, default: None)

A vertex property map where the predecessor map will be stored (must have value type "int").

combine : binary function (optional, default: lambda a, b: a + b)

This function is used to combine distances to compute the distance of a path.

compare : binary function (optional, default: lambda a, b: a < b)

This function is use to compare distances to determine which vertex is closer to the source vertex.

zero : int or float (optional, default: 0)

Value assumed to correspond to a distance of zero by the combine and compare functions.

infinity : int or float (optional, default: float('inf'))

Value assumed to correspond to a distance of infinity by the combine and compare functions.

Returns **minimized** : bool

True if all edges were successfully minimized, or False if there is a negative loop in the graph.

dist_map : PropertyMap (page 35)

A vertex property map with the computed distances from the source.

pred_map : PropertyMap (page 35)

A vertex property map with the predecessor tree.

See Also:

bfs_search (page 169) Breadth-first search

dfs_search (page 171) Depth-first search

dijsktra_search Dijkstra search

astar_search (page 181) A^* heuristic search

Notes

The Bellman-Ford algorithm [bellman-ford] (page 231) solves the single-source shortest paths problem for a graph with both positive and negative edge weights. If you only need to solve the shortest paths problem for positive edge weights, `dijkstra_search()` (page 174) provides a more efficient alternative. If all the edge weights are all equal, then `bfs_search()` (page 169) provides an even more efficient alternative.

The Bellman-Ford algorithm proceeds by looping through all of the edges in the graph, applying the relaxation operation to each edge. In the following pseudo-code, `v` is a vertex adjacent to `u`, `w` maps edges to their weight, and `d` is a distance map that records the length of the shortest path to each vertex seen so far. `p` is a predecessor map which records the parent of each vertex, which will ultimately be the parent in the shortest paths tree

```
RELAX(u, v, w, d, p)
  if (w(u,v) + d[u] < d[v])
    d[v] := w(u,v) + d[u]       relax edge (u,v)
    p[v] := u
  else
    ...                         edge (u,v) is not relaxed
```

The algorithm repeats this loop `|V|` times after which it is guaranteed that the distances to each vertex have been reduced to the minimum possible unless there is a negative cycle in the graph. If there is a negative cycle, then there will be edges in the graph that were not properly minimized. That is, there will be edges `(u,v)` such that `w(u,v) + d[u] < d[v]`. The algorithm loops over the edges in the graph one final time to check if all the edges were minimized, returning true if they were and returning false otherwise.

```
BELLMAN-FORD(G)
  for each vertex u in V
    d[u] := infinity
    p[u] := u
  end for
  for i := 1 to |V|-1
    for each edge (u,v) in E          examine edge (u,v)
      RELAX(u, v, w, d, p)
    end for
  end for
  for each edge (u,v) in E
    if (w(u,v) + d[u] < d[v])
      return (false, , )              edge (u,v) was not minimized
    else
      ...                             edge (u,v) was minimized
  end for
  return (true, p, d)
```

The time complexity is $O(VE)$.

References

[bellman-ford] (page 231), [bellman-ford-bgl] (page 231), [bellman-ford-wikipedia] (page 231)

Examples

We must define what should be done during the search by subclassing BellmanFordVisitor (page 177), and specializing the appropriate methods. In the following we will keep track of the edge minimizations.

```
class VisitorExample(gt.BellmanFordVisitor):

    def __init__(self, name):
        self.name = name

    def edge_minimized(self, e):
        print("edge (%s, %s) has been minimized..." % \
              (self.name[e.source()], self.name[e.target()]))

    def edge_not_minimized(self, e):
        print("edge (%s, %s) has not been minimized..." % \
              (self.name[e.source()], self.name[e.target()]))
```

With the above class defined, we can perform the Bellman-Ford search as follows.

```
>>> nweight = g.copy_property(weight)
>>> nweight.a[6] *= -1   # include negative weight in edge (Carlos, Alice)
>>> minimized, dist, pred = gt.bellman_ford_search(g, g.vertex(0), nweight, VisitorExample(na
edge (Bob, Eve) has been minimized...
edge (Bob, Chuck) has been minimized...
edge (Bob, Carlos) has been minimized...
edge (Bob, Isaac) has been minimized...
edge (Alice, Oscar) has been minimized...
edge (Alice, Dave) has been minimized...
edge (Alice, Carlos) has been minimized...
edge (Carol, Eve) has been minimized...
edge (Carol, Imothep) has been minimized...
edge (Carlos, Eve) has been minimized...
edge (Carlos, Imothep) has been minimized...
edge (Chuck, Eve) has been minimized...
edge (Chuck, Isaac) has been minimized...
edge (Chuck, Imothep) has been minimized...
edge (Dave, Oscar) has been minimized...
edge (Eve, Isaac) has been minimized...
edge (Eve, Imothep) has been minimized...
>>> print(minimized)
True
>>> print(pred.a)
[3 3 9 1 6 1 3 6 1 3]
>>> print(dist.a)
[-28.42555934 -37.34471821 -25.20438243 -41.97110592 -35.20316571
 -34.02873843 -36.58860946 -33.55645565 -35.2199616  -36.0029274 ]
```

class graph_tool.search.**AStarVisitor**

A visitor object that is invoked at the event-points inside the astar_search() (page 181) algorithm. By default, it performs no action, and should be used as a base class in order to be useful.

initialize_vertex(*self*, *u*)

This is invoked on every vertex of the graph before the start of the graph search.

examine_vertex(*self*, *u*)

This is invoked on a vertex as it is popped from the queue (i.e. it has the lowest cost on the OPEN list). This happens immediately before examine_edge() is invoked on each of the out-edges of vertex u.

examine_edge(*self*, *e*)

This is invoked on every out-edge of each vertex after it is examined.

discover_vertex(*self*, *u*)
This is invoked when a vertex is first discovered and is added to the OPEN list.

edge_relaxed(*self*, *e*)
Upon examination, if the following condition holds then the edge is relaxed (its distance is reduced), and this method is invoked.

```
(u, v) = tuple(e)
assert(compare(combine(d[u], weight[e]), d[v]))
```

edge_not_relaxed(*self*, *e*)
Upon examination, if the edge is not relaxed (see edge_relaxed() (page 181)) then this method is invoked.

black_target(*self*, *e*)
This is invoked when a vertex that is on the CLOSED list is "rediscovered" via a more efficient path, and is re-added to the OPEN list.

finish_vertex(*self*, *u*)
This is invoked on a vertex when it is added to the CLOSED list, which happens after all of its out edges have been examined.

graph_tool.search.**astar_search**(*g*, *source*, *weight*, *visitor=<graph_tool.search.AStarVisitor object at 0x7fc145948690>*, *heuristic=<function <lambda> at 0x7fc14594c440>*, *dist_map=None*, *pred_map=None*, *cost_map=None*, *combine=<function <lambda> at 0x7fc14594c950>*, *compare=<function <lambda> at 0x7fc14594c9e0>*, *zero=0*, *infinity=inf*, *implicit=False*)

Heuristic A^* search on a weighted, directed or undirected graph for the case where all edge weights are non-negative.

Parameters **g** : Graph (page 28)

Graph to be used.

source : Vertex (page 33)

Source vertex.

weight : PropertyMap (page 35)

Edge property map with weight values.

visitor : AStarVisitor (page 180) (optional, default: AStarVisitor())

A visitor object that is invoked at the event points inside the algorithm. This should be a subclass of AStarVisitor (page 180).

heuristic : unary function (optional, default: lambda v: 1)

The heuristic function that guides the search. It should take a single argument which is a Vertex (page 33), and output an estimated distance from the source vertex.

dist_map : PropertyMap (page 35) (optional, default: None)

A vertex property map where the distances from the source will be stored.

pred_map : PropertyMap (page 35) (optional, default: None)

A vertex property map where the predecessor map will be stored (must have value type "int").

cost_map : PropertyMap (page 35) (optional, default: None)

A vertex property map where the vertex costs will be stored. It must have the same value type as `dist_map`. This parameter is only used if `implicit` is True.

combine : binary function (optional, default: `lambda a, b:  a + b`)

This function is used to combine distances to compute the distance of a path.

compare : binary function (optional, default: `lambda a, b:  a < b`)

This function is use to compare distances to determine which vertex is closer to the source vertex.

implicit : bool (optional, default: `False`)

If true, the underlying graph will be assumed to be implicit (i.e. constructed during the search).

zero : int or float (optional, default: `0`)

Value assumed to correspond to a distance of zero by the combine and compare functions.

infinity : int or float (optional, default: `float('inf')`)

Value assumed to correspond to a distance of infinity by the combine and compare functions.

Returns **dist_map** : `PropertyMap` (page 35)

A vertex property map with the computed distances from the source.

See Also:

`bfs_search` (page 169) Breadth-first search

`dfs_search` (page 171) Depth-first search

`dijkstra_search` (page 174) Dijkstra's search algorithm

Notes

The A^* algorithm is a heuristic graph search algorithm: an A^* search is "guided" by a heuristic function. A heuristic function $h(v)$ is one which estimates the cost from a non-goal state (v) in the graph to some goal state, t. Intuitively, A^* follows paths (through the graph) to the goal that are estimated by the heuristic function to be the best paths. Unlike best-first search, A^* takes into account the known cost from the start of the search to v; the paths A^* takes are guided by a function $f(v) = g(v) + h(v)$, where $h(v)$ is the heuristic function, and $g(v)$ (sometimes denoted $c(s, v)$) is the known cost from the start to v. Clearly, the efficiency of A^* is highly dependent on the heuristic function with which it is used.

The time complexity is $O((E + V) \log V)$.

The pseudo-code for the A^* algorithm is listed below, with the annotated event points, for which the given visitor object will be called with the appropriate method.

```
A*(G, source, h)
  for each vertex u in V                  initialize vertex u
    d[u] := f[u] := infinity
    color[u] := WHITE
  end for
  color[s] := GRAY
  d[s] := 0
  f[s] := h(source)
  INSERT(Q, source)                       discover vertex source
```

```
while (Q != ∅)
  u := EXTRACT-MIN(Q)                   examine vertex u
  for each vertex v in Adj[u]           examine edge (u,v)
    if (w(u,v) + d[u] < d[v])
      d[v] := w(u,v) + d[u]             edge (u,v) relaxed
      f[v] := d[v] + h(v)
      if (color[v] = WHITE)
        color[v] := GRAY
        INSERT(Q, v)                    discover vertex v
      else if (color[v] = BLACK)
        color[v] := GRAY
        INSERT(Q, v)                    reopen vertex v
      end if
    else
      ...                               edge (u,v) not relaxed
  end for
  color[u] := BLACK                     finish vertex u
end while
```

References

[astar] (page 231), [astar-bgl] (page 231), [astar-wikipedia] (page 231)

Examples

We will use an irregular two-dimensional lattice as an example, where the heuristic function will be based on the euclidean distance to the target.

The heuristic function will be defined as:

```
def h(v, target, pos):
    return sqrt(sum((pos[v].a - pos[target].a) ** 2))
```

where `pos` is the vertex position in the plane.

We must define what should be done during the search by subclassing `AStarVisitor` (page 180), and specializing the appropriate methods. In the following we will keep track of the discovered vertices, and which edges were examined, as well as the predecessor tree. We will also abort the search when a given target vertex is found, by raising the `StopSearch` (page 186) exception.

```
class VisitorExample(gt.AStarVisitor):

    def __init__(self, touched_v, touched_e, target):
        self.touched_v = touched_v
        self.touched_e = touched_e
        self.target = target

    def discover_vertex(self, u):
        self.touched_v[u] = True

    def examine_edge(self, e):
        self.touched_e[e] = True

    def edge_relaxed(self, e):
        if e.target() == self.target:
            raise gt.StopSearch()
```

With the above class defined, we can perform the A^* search as follows.

```
>>> points = random((500, 2)) * 4
>>> points[0] = [-0.01, 0.01]
>>> points[1] = [4.01, 4.01]
>>> g, pos = gt.triangulation(points, type="delaunay")
>>> weight = g.new_edge_property("double") # Edge weights corresponding to
...                                        # Euclidean distances
>>> for e in g.edges():
...     weight[e] = sqrt(sum((pos[e.source()].a -
...                           pos[e.target()].a) ** 2))
>>> touch_v = g.new_vertex_property("bool")
>>> touch_e = g.new_edge_property("bool")
>>> target = g.vertex(1)
>>> dist, pred = gt.astar_search(g, g.vertex(0), weight,
...                              VisitorExample(touch_v, touch_e, target),
...                              heuristic=lambda v: h(v, target, pos))
```

We can now observe the best path found, and how many vertices and edges were visited in the process.

```
>>> ecolor = g.new_edge_property("string")
>>> ewidth = g.new_edge_property("double")
>>> ewidth.a = 1
>>> for e in g.edges():
...     ecolor[e] = "blue" if touch_e[e] else "black"
>>> v = target
>>> while v != g.vertex(0):
...     p = g.vertex(pred[v])
...     for e in v.out_edges():
...         if e.target() == p:
...             ecolor[e] = "#a40000"
...             ewidth[e] = 3
...     v = p
>>> gt.graph_draw(g, pos=pos, output_size=(300, 300), vertex_fill_color=touch_v,
...               vcmap=matplotlib.cm.binary, edge_color=ecolor,
...               edge_pen_width=ewidth, output="astar-delaunay.pdf")
<...>
```

The A^* algorithm is very useful for searching *implicit* graphs, i.e. graphs which are not entirely stored in memory and are generated "on-the-fly" during the search. In the following example we will carry out a search in a hamming hypercube of 10 bits witch has random weights on its edges in the range $[0, 1]$. The vertices of the hypercube will be created during the search.

The heuristic function will use the Hamming distance between vertices:

```
def h(v, target, state):
    return sum(abs(state[v].a - target)) / 2
```

In the following visitor we will keep growing the graph on-the-fly, and abort the search when a given target vertex is found, by raising the `StopSearch` (page 186) exception.

```
from numpy.random import random

class HammingVisitor(gt.AStarVisitor):

    def __init__(self, g, target, state, weight, dist, cost):
        self.g = g
        self.state = state
        self.target = target
        self.weight = weight
        self.dist = dist
        self.cost = cost
        self.visited = {}
```

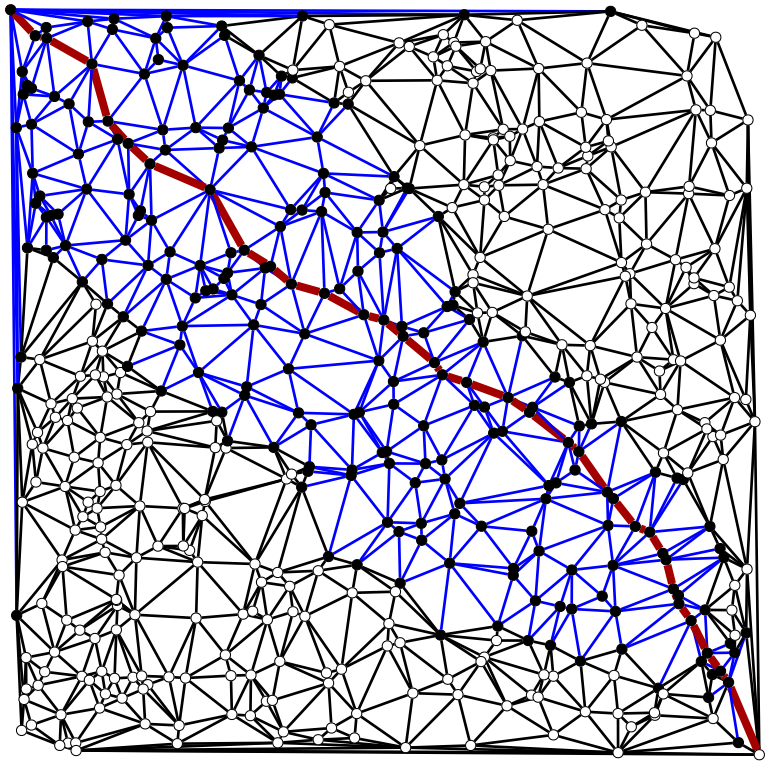


Figure 3.44: The shortest path is shown in red. The visited edges are shown in blue, and the visited vertices in black.

```
    def examine_vertex(self, u):
        for i in range(len(self.state[u])):
            nstate = list(self.state[u])
            nstate[i] ^= 1
            if tuple(nstate) in self.visited:
                v = self.visited[tuple(nstate)]
            else:
                v = self.g.add_vertex()
                self.visited[tuple(nstate)] = v
                self.state[v] = nstate
                self.dist[v] = self.cost[v] = float('inf')
            for e in u.out_edges():
                if e.target() == v:
                    break
            else:
                e = self.g.add_edge(u, v)
                self.weight[e] = random()
        self.visited[tuple(self.state[u])] = u

    def edge_relaxed(self, e):
        if self.state[e.target()] == self.target:
            self.visited[tuple(self.target)] = e.target()
            raise gt.StopSearch()
```

With the above class defined, we can perform the A^* search as follows.

```
>>> g = gt.Graph(directed=False)
>>> state = g.new_vertex_property("vector<bool>")
>>> v = g.add_vertex()
>>> state[v] = [0] * 10
>>> target = [1] * 10
```

```
>>> weight = g.new_edge_property("double")
>>> dist = g.new_vertex_property("double")
>>> cost = g.new_vertex_property("double")
>>> visitor = HammingVisitor(g, target, state, weight, dist, cost)
>>> dist, pred = gt.astar_search(g, g.vertex(0), weight, visitor, dist_map=dist,
...                              cost_map=cost, heuristic=lambda v: h(v, array(target), state
...                              implicit=True)
```

We can now observe the best path found, and how many vertices and edges were visited in the process.

```
>>> ecolor = g.new_edge_property("string")
>>> vcolor = g.new_vertex_property("string")
>>> ewidth = g.new_edge_property("double")
>>> ewidth.a = 1
>>> for e in g.edges():
...     ecolor[e] = "black"
>>> for v in g.vertices():
...     vcolor[v] = "white"
>>> v = visitor.visited[tuple(target)]
>>> while v != g.vertex(0):
...     vcolor[v] = "black"
...     p = g.vertex(pred[v])
...     for e in v.out_edges():
...         if e.target() == p:
...             ecolor[e] = "#a40000"
...             ewidth[e] = 3
...     v = p
>>> vcolor[v] = "black"
>>> pos = gt.graph_draw(g, output_size=(300, 300), vertex_fill_color=vcolor, edge_color=ecolo
...                     edge_pen_width=ewidth, output="astar-implicit.pdf")
```

exception graph_tool.search.**StopSearch**
If this exception is raised from inside any search visitor object, the search is aborted.

3.2.11 graph_tool.spectral - Spectral properties

Summary

adjacency (page 186)	Return the adjacency matrix of the graph.
laplacian (page 188)	Return the Laplacian matrix of the graph.
incidence (page 189)	Return the incidence matrix of the graph.

Contents

graph_tool.spectral.**adjacency**(*g*, *weight=None*, *index=None*)
Return the adjacency matrix of the graph.

Parameters **g** : Graph (page 28)

Graph to be used.

weight : PropertyMap (page 35) (optional, default: True)

Edge property map with the edge weights.

index : PropertyMap (page 35) (optional, default: None)

Vertex property map specifying the row/column indexes. If not provided, the internal vertex index is used.

Returns **a** : csr_matrix

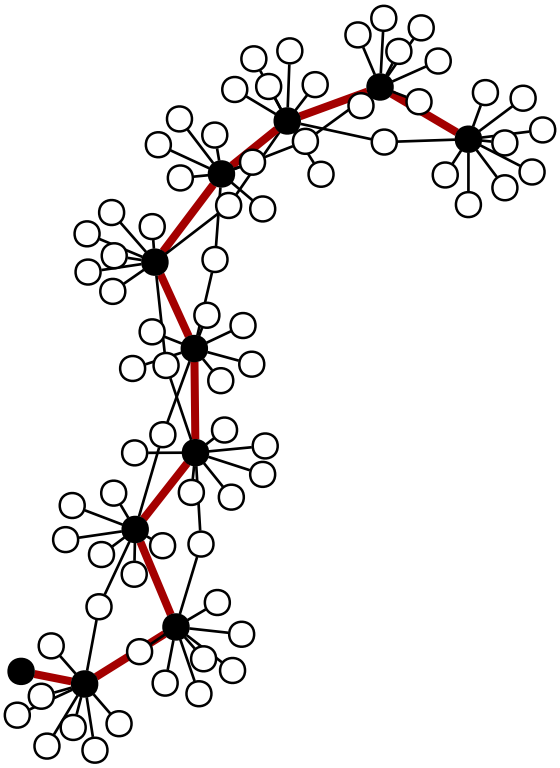


Figure 3.45: The shortest path is shown in red, and the vertices which belong to it are in black. Note that the number of vertices visited is much smaller than the total number $2^{10} = 1024$.

The (sparse) adjacency matrix.

Notes

The adjacency matrix is defined as

$$a_{i,j} = \begin{cases} 1 & \text{if } v_i \text{ is adjacent to } v_j, \\ 2 & \text{if } i = j, \text{ the graph is undirected and there is a self-loop incident in } v_i, \\ 0 & \text{otherwise} \end{cases}$$

In the case of weighted edges, the entry values are multiplied by the weight of the respective edge.

In the case of networks with parallel edges, the entries in the matrix become simply the edge multiplicities.

References

[wikipedia-adjacency] (page 231)

Examples

```
>>> g = gt.collection.data["polblogs"]
>>> A = gt.adjacency(g)
>>> ew, ev = scipy.linalg.eig(A.todense())
```

```
>>> figure(figsize=(8, 2))
<...>
>>> scatter(real(ew), imag(ew), c=abs(ew))
<...>
>>> xlabel(r"$\operatorname{Re}(\lambda)$")
<...>
>>> ylabel(r"$\operatorname{Im}(\lambda)$")
<...>
>>> tight_layout()
>>> savefig("adjacency-spectrum.pdf")
```

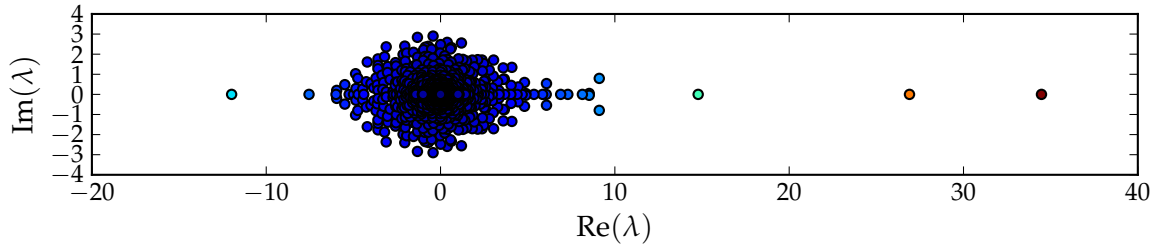


Figure 3.46: Adjacency matrix spectrum for the political blog network.

graph_tool.spectral.**laplacian**(*g*, *deg='total'*, *normalized=False*, *weight=None*, *index=None*)

Return the Laplacian matrix of the graph.

Parameters **g** : Graph (page 28)

Graph to be used.

deg : str (optional, default: "total")

Degree to be used, in case of a directed graph.

normalized : bool (optional, default: False)

Whether to compute the normalized Laplacian.

weight : PropertyMap (page 35) (optional, default: True)

Edge property map with the edge weights.

index : PropertyMap (page 35) (optional, default: None)

Vertex property map specifying the row/column indexes. If not provided, the internal vertex index is used.

Returns **l** : csr_matrix

The (sparse) Laplacian matrix.

Notes

The weighted Laplacian matrix is defined as

$$\ell_{ij} = \begin{cases} \Gamma(v_i) & \text{if } i = j \\ -w_{ij} & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise.} \end{cases}$$

Where $\Gamma(v_i) = \sum_j A_{ij}w_{ij}$ is sum of the weights of the edges incident on vertex v_i. The normalized version is

$$\ell_{ij} = \begin{cases} 1 & \text{if } i = j \text{ and } \Gamma(v_i) \neq 0 \\ -\frac{w_{ij}}{\sqrt{\Gamma(v_i)\Gamma(v_j)}} & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise.} \end{cases}$$

In the case of unweighted edges, it is assumed $w_{ij} = 1$.

For directed graphs, it is assumed $\Gamma(v_i) = \sum_j A_{ij}w_{ij} + \sum_j A_{ji}w_{ji}$ if `deg=="total"`, $\Gamma(v_i) = \sum_j A_{ij}w_{ij}$ if `deg=="out"` or $\Gamma(v_i) = \sum_j A_{ji}w_{ji}$ `deg=="in"`.

References

[wikipedia-laplacian] (page 231)

Examples

```
>>> g = gt.collection.data["polblogs"]
>>> L = gt.laplacian(g)
>>> ew, ev = scipy.linalg.eig(L.todense())

>>> figure(figsize=(8, 2))
<...>
>>> scatter(real(ew), imag(ew), c=abs(ew))
<...>
>>> xlabel(r"$\operatorname{Re}(\lambda)$")
<...>
>>> ylabel(r"$\operatorname{Im}(\lambda)$")
<...>
>>> tight_layout()
>>> savefig("laplacian-spectrum.pdf")
```

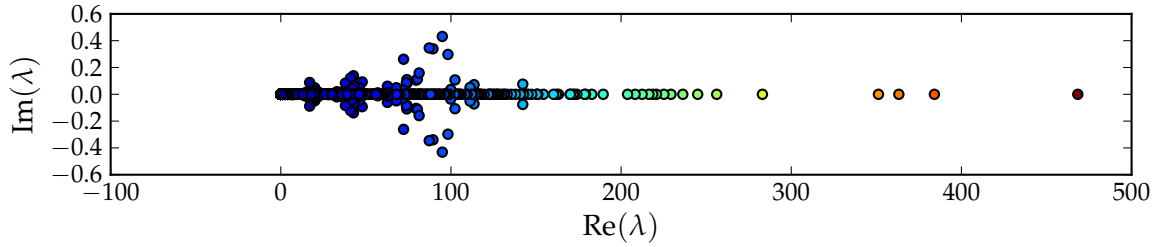


Figure 3.47: Laplacian matrix spectrum for the political blog network.

```
>>> L = gt.laplacian(g, normalized=True)
>>> ew, ev = scipy.linalg.eig(L.todense())

>>> figure(figsize=(8, 2))
<...>
>>> scatter(real(ew), imag(ew), c=abs(ew))
<...>
>>> xlabel(r"$\operatorname{Re}(\lambda)$")
<...>
>>> ylabel(r"$\operatorname{Im}(\lambda)$")
<...>
>>> tight_layout()
>>> savefig("norm-laplacian-spectrum.pdf")
```

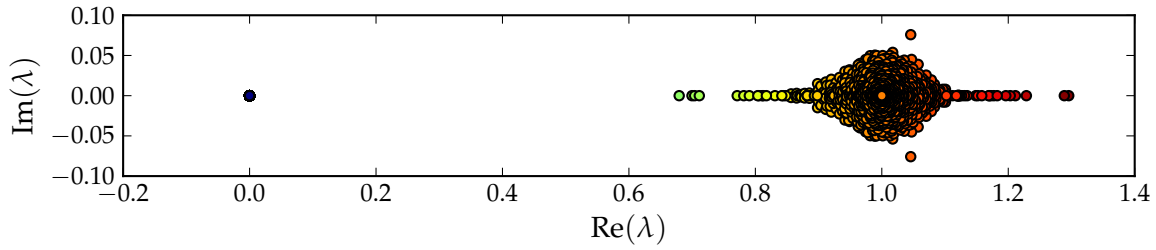


Figure 3.48: Normalized Laplacian matrix spectrum for the political blog network.

graph_tool.spectral.**incidence**(*g*, *vindex=None*, *eindex=None*)

Return the incidence matrix of the graph.

Parameters **g** : Graph (page 28)

Graph to be used.

vindex : PropertyMap (page 35) (optional, default: None)

Vertex property map specifying the row indexes. If not provided, the internal vertex index is used.

eindex : PropertyMap (page 35) (optional, default: None)

Edge property map specifying the column indexes. If not provided, the internal edge index is used.

Returns **a** : csr_matrix

The (sparse) incidence matrix.

Notes

For undirected graphs, the incidence matrix is defined as

$$b_{i,j} = \begin{cases} 1 & \text{if vertex } v_i \text{and edge } e_j \text{ are incident,} \\ 0 & \text{otherwise} \end{cases}$$

For directed graphs, the definition is

$$b_{i,j} = \begin{cases} 1 & \text{if edge } e_j \text{ enters vertex } v_i, \\ -1 & \text{if edge } e_j \text{ leaves vertex } v_i, \\ 0 & \text{otherwise} \end{cases}$$

References

[wikipedia-incidence] (page 231)

Examples

```
>>> g = gt.random_graph(100, lambda: (2,2))
>>> m = gt.incidence(g)
>>> print(m.todense())
[[-1. -1.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]
```

```
...,
[ 0.  0. -1. ...,  0.  0.  0.]
[ 0.  0.  0. ...,  0.  0.  0.]
[ 0.  0.  0. ...,  1.  0.  0.]]
```

3.2.12 graph_tool.stats - Miscellaneous statistics

Summary

vertex_hist (page 190)	Return the vertex histogram of the given degree type or property.
edge_hist (page 191)	Return the edge histogram of the given property.
vertex_average (page 192)	Return the average of the given degree or vertex property.
edge_average (page 192)	Return the average of the given degree or vertex property.
label_parallel_edges (page 193)	Label edges which are parallel, i.e, have the same source and targe
remove_parallel_edges (page 193)	Remove all parallel edges from the graph.
label_self_loops (page 193)	Label edges which are self-loops, i.e, the source and target vertices
remove_self_loops (page 193)	Remove all self-loops edges from the graph.
remove_labeled_edges (page 193)	Remove every edge *e* such that *label[e] != 0*.
distance_histogram (page 193)	Return the shortest-distance histogram for each vertex pair in the

Contents

graph_tool.stats.**vertex_hist**(*g, deg, bins=[0, 1], float_count=True*)

Return the vertex histogram of the given degree type or property.

Parameters **g** : Graph (page 28)

Graph to be used.

deg : string or PropertyMap (page 35)

Degree or property to be used for the histogram. It can be either "in", "out" or "total", for in-, out-, or total degree of the vertices. It can also be a vertex property map.

bins : list of bins (optional, default: [0, 1])

List of bins to be used for the histogram. The values given represent the edges of the bins (i.e. lower and upper bounds). If the list contains two values, this will be used to automatically create an appropriate bin range, with a constant width given by the second value, and starting from the first value.

float_count : bool (optional, default: True)

If True, the counts in each histogram bin will be returned as floats. If False, they will be returned as integers.

Returns **counts** : ndarray

The bin counts.

bins : ndarray

The bin edges.

See Also:

edge_hist (page 191) Edge histograms.

vertex_average (page 192) Average of vertex properties, degrees.

edge_average (page 192) Average of edge properties.

distance_histogram (page 193) Shortest-distance histogram.

Notes

The algorithm runs in $O(|V|)$ time.

If enabled during compilation, this algorithm runs in parallel.

Examples

```
>>> from numpy.random import poisson
>>> g = gt.random_graph(1000, lambda: (poisson(5), poisson(5)))
>>> print(gt.vertex_hist(g, "out"))
[array([  10.,   36.,   90.,  147.,  164.,  165.,  142.,  109.,   70.,
         31.,   28.,    7.,    1.]), array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 1
```

graph_tool.stats.**edge_hist**(*g, eprop, bins=[0, 1], float_count=True*)

Return the edge histogram of the given property.

Parameters **g** : Graph (page 28)

Graph to be used.

eprop : PropertyMap (page 35)

Edge property to be used for the histogram.

bins : list of bins (optional, default: [0, 1])

List of bins to be used for the histogram. The values given represent the edges of the bins (i.e. lower and upper bounds). If the list contains two values, this will be used to automatically create an appropriate bin range, with a constant width given by the second value, and starting from the first value.

float_count : bool (optional, default: True)

If True, the counts in each histogram bin will be returned as floats. If False, they will be returned as integers.

Returns **counts** : ndarray

The bin counts.

bins : ndarray

The bin edges.

See Also:

vertex_hist (page 190) Vertex histograms.

vertex_average (page 192) Average of vertex properties, degrees.

edge_average (page 192) Average of edge properties.

distance_histogram (page 193) Shortest-distance histogram.

Notes

The algorithm runs in $O(|E|)$ time.

If enabled during compilation, this algorithm runs in parallel.

Examples

```
>>> from numpy import arange
>>> from numpy.random import random
>>> g = gt.random_graph(1000, lambda: (5, 5))
>>> eprop = g.new_edge_property("double")
>>> eprop.get_array()[:] = random(g.num_edges())
>>> print(gt.edge_hist(g, eprop, linspace(0, 1, 11)))
[array([ 483.,  462.,  467.,  493.,  498.,  486.,  515.,  552.,  496.,  548.]), array([ 0. ,
```

graph_tool.stats.**vertex_average**(*g*, *deg*)

Return the average of the given degree or vertex property.

Parameters **g** : Graph (page 28)

Graph to be used.

deg : string or PropertyMap (page 35)

Degree or property to be used for the histogram. It can be either "in", "out" or "total", for in-, out-, or total degree of the vertices. It can also be a vertex property map.

Returns **average** : float

The average of the given degree or property.

std : float

The standard deviation of the average.

See Also:

vertex_hist (page 190) Vertex histograms.

edge_hist (page 191) Edge histograms.

edge_average (page 192) Average of edge properties.

distance_histogram (page 193) Shortest-distance histogram.

Notes

The algorithm runs in $O(|V|)$ time.

If enabled during compilation, this algorithm runs in parallel.

Examples

```
>>> from numpy.random import poisson
>>> g = gt.random_graph(1000, lambda: (poisson(5), poisson(5)))
>>> print(gt.vertex_average(g, "in"))
(4.982, 0.06855418295042251)
```

graph_tool.stats.**edge_average**(*g*, *eprop*)

Return the average of the given degree or vertex property.

Parameters **g** : Graph (page 28)

Graph to be used.

eprop : PropertyMap (page 35)

Edge property to be used for the histogram.

Returns **average** : float

The average of the given property.

std : float

The standard deviation of the average.

See Also:

`vertex_hist` **(page 190)** Vertex histograms.

`edge_hist` **(page 191)** Edge histograms.

`vertex_average` **(page 192)** Average of vertex degree, properties.

`distance_histogram` **(page 193)** Shortest-distance histogram.

Notes

The algorithm runs in $O(|E|)$ time.

If enabled during compilation, this algorithm runs in parallel.

Examples

```
>>> from numpy import arange
>>> from numpy.random import random
>>> g = gt.random_graph(1000, lambda: (5, 5))
>>> eprop = g.new_edge_property("double")
>>> eprop.get_array()[:] = random(g.num_edges())
>>> print(gt.edge_average(g, eprop))
(0.49849732125677476, 0.004086182531863621)
```

graph_tool.stats.**remove_labeled_edges**(*g*, *label*)
Remove every edge *e* such that *label[e] != 0*.

graph_tool.stats.**label_parallel_edges**(*g*, *mark_only=False*, *count_all=False*, *eprop=None*)
Label edges which are parallel, i.e, have the same source and target vertices. For each parallel edge set PE, the labelling starts from 0 to $|PE|-1$. (If *count_all==True*, the range is 0 to $|PE|$ instead). If *mark_only==True*, all parallel edges are simply marked with the value 1. If the *eprop* parameter is given (a `PropertyMap` (page 35)), the labelling is stored there.

graph_tool.stats.**remove_parallel_edges**(*g*)
Remove all parallel edges from the graph. Only one edge from each parallel edge set is left.

graph_tool.stats.**label_self_loops**(*g*, *mark_only=False*, *eprop=None*)
Label edges which are self-loops, i.e, the source and target vertices are the same. For each self-loop edge set SL, the labelling starts from 0 to $|SL|-1$. If *mark_only == True*, self-loops are labeled with 1 and others with 0. If the *eprop* parameter is given (a `PropertyMap` (page 35)), the labelling is stored there.

graph_tool.stats.**remove_self_loops**(*g*)
Remove all self-loops edges from the graph.

graph_tool.stats.**distance_histogram**(*g*, *weight=None*, *bins=[0, 1]*, *samples=None*, *float_count=True*)
Return the shortest-distance histogram for each vertex pair in the graph.

Parameters **g** : `Graph`

Graph to be used.

weight : `PropertyMap` (page 35) (optional, default: None)

Edge weights.

bins : list of bins (optional, default: [0, 1])

List of bins to be used for the histogram. The values given represent the edges of the bins (i.e. lower and upper bounds). If the list contains two values, this will be used to automatically create an appropriate bin range, with a constant width given by the second value, and starting from the first value.

samples : int (optional, default: None)

If supplied, the distances will be randomly sampled from a number of source vertices given by this parameter. It *samples == None* (default), all pairs are used.

float_count : bool (optional, default: True)

If True, the counts in each histogram bin will be returned as floats. If False, they will be returned as integers.

Returns **counts** : ndarray

The bin counts.

bins : ndarray

The bin edges.

See Also:

vertex_hist (page 190) Vertex histograms.

edge_hist (page 191) Edge histograms.

vertex_average (page 192) Average of vertex degree, properties.

distance_histogram (page 193) Shortest-distance histogram.

Notes

The algorithm runs in $O(V^2)$ time, or $O(V^2 \log V)$ if *weight != None*. If *samples* is supplied, the complexities are $O(\text{samples} \times V)$ and $O(\text{samples} \times V \log V)$, respectively.

If enabled during compilation, this algorithm runs in parallel.

Examples

```
>>> g = gt.random_graph(100, lambda: (3, 3))
>>> hist = gt.distance_histogram(g)
>>> print(hist)
[array([   0.,  300.,  866., 2206., 3893., 2476.,  159.]), array([0, 1, 2, 3, 4, 5, 6,
>>> hist = gt.distance_histogram(g, samples=10)
>>> print(hist)
[array([  0.,  30.,  84., 217., 385., 249.,  25.]), array([0, 1, 2, 3, 4, 5, 6, 7], dt
```

3.2.13 graph_tool.topology - Assessing graph topology

Summary

shortest_distance (page 209)	Calculate the distance from a source to a target vertex, or

Table 3.18 – continued fro

shortest_path (page 211)	Return the shortest path from *source* to *target*.
pseudo_diameter (page 212)	Compute the pseudo-diameter of the graph.
similarity (page 195)	Return the adjacency similarity between the two graphs.
isomorphism (page 196)	Check whether two graphs are isomorphic.
subgraph_isomorphism (page 196)	Obtain all subgraph isomorphisms of *sub* in *g* (or at most
mark_subgraph (page 198)	Mark a given subgraph *sub* on the graph *g*.
max_cardinality_matching (page 217)	Find a maximum cardinality matching in the graph.
max_independent_vertex_set (page 218)	Find a maximal independent vertex set in the graph.
min_spanning_tree (page 198)	Return the minimum spanning tree of a given graph.
random_spanning_tree (page 200)	Return a random spanning tree of a given graph, which ca
dominator_tree (page 203)	Return a vertex property map the dominator vertices for ea
topological_sort (page 204)	Return the topological sort of the given graph.
transitive_closure (page 204)	Return the transitive closure graph of g.
tsp_tour (page 221)	Return a traveling salesman tour of the graph, which is gu
sequential_vertex_coloring (page 222)	Returns a vertex coloring of the graph.
label_components (page 205)	Label the components to which each vertex in the graph be
label_biconnected_components (page 207)	Label the edges of biconnected components, and the vertic
label_largest_component (page 206)	Label the largest component in the graph.
label_out_component (page 206)	Label the out-component (or simply the component for und
kcore_decomposition (page 208)	Perform a k-core decomposition of the given graph.
is_bipartite (page 213)	Test if the graph is bipartite.
is_DAG (page 216)	Return *True* if the graph is a directed acyclic graph (DAG).
is_planar (page 214)	Test if the graph is planar.
make_maximal_planar (page 215)	Add edges to the graph to make it maximally planar.
edge_reciprocity (page 220)	Calculate the edge reciprocity of the graph.

Contents

graph_tool.topology.**similarity**(*g1*, *g2*, *label1=None*, *label2=None*, *norm=True*)

Return the adjacency similarity between the two graphs.

Parameters **g1** : Graph (page 28)

First graph to be compared.

g2 : Graph (page 28)

Second graph to be compared.

label1 : PropertyMap (page 35) (optional, default: None)

Vertex labels for the first graph to be used in comparison. If not supplied, the vertex indexes are used.

label2 : PropertyMap (page 35) (optional, default: None)

Vertex labels for the second graph to be used in comparison. If not supplied, the vertex indexes are used.

norm : bool (optional, default: True)

If True, the returned value is normalized by the total number of edges.

Returns **similarity** : float

Adjacency similarity value.

Notes

The adjacency similarity is the sum of equal entries in the adjacency matrix, given a vertex ordering determined by the vertex labels. In other words it counts the number of edges which have the same source and target labels in both graphs.

The algorithm runs with complexity $O(E_1 + V_1 + E_2 + V_2)$.

Examples

```
>>> g = gt.random_graph(100, lambda: (3,3))
>>> u = g.copy()
>>> gt.similarity(u, g)
1.0
>>> gt.random_rewire(u)
21
>>> gt.similarity(u, g)
0.03
```

graph_tool.topology.**isomorphism**(*g1*, *g2*, *isomap=False*)

Check whether two graphs are isomorphic.

If *isomap* is True, a vertex PropertyMap (page 35) with the isomorphism mapping is returned as well.

Examples

```
>>> g = gt.random_graph(100, lambda: (3,3))
>>> g2 = gt.Graph(g)
>>> gt.isomorphism(g, g2)
True
>>> g.add_edge(g.vertex(0), g.vertex(1))
<...>
>>> gt.isomorphism(g, g2)
False
```

graph_tool.topology.**subgraph_isomorphism**(*sub*, *g*, *max_n=0*, *random=False*)

Obtain all subgraph isomorphisms of *sub* in *g* (or at most *max_n* subgraphs, if *max_n* > *0*).

Parameters **sub** : Graph (page 28)

Subgraph for which to be searched.

g : Graph (page 28)

Graph in which the search is performed.

max_n : int (optional, default: 0)

Maximum number of matches to find. If *max_n* == *0*, all matches are found.

random : bool (optional, default: False)

If *True*, the vertices of *g* are indexed in random order before the search.

Returns **vertex_maps** : list of PropertyMap (page 35) objects

List containing vertex property map objects which indicate different isomorphism mappings. The property maps vertices in *sub* to the corresponding vertex index in *g*.

edge_maps : list of `PropertyMap` (page 35) objects

List containing edge property map objects which indicate different isomorphism mappings. The property maps edges in *sub* to the corresponding edge index in *g*.

Notes

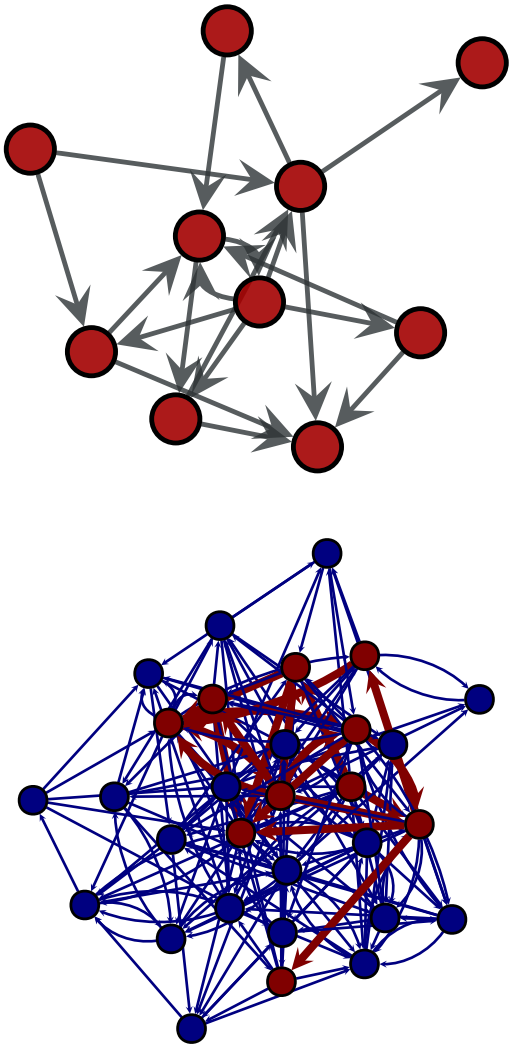
The algorithm used is described in [ullmann-algorithm-1976] (page 231). It has a worse-case complexity of $O(N_g^{N_{sub}})$, but for random graphs it typically has a complexity of $O(N_g^{\gamma})$ with γ depending sub-linearly on the size of *sub*.

References

[ullmann-algorithm-1976] (page 231), [subgraph-isormophism-wikipedia] (page 231)

Examples

```
>>> from numpy.random import poisson
>>> g = gt.random_graph(30, lambda: (poisson(6.1), poisson(6.1)))
>>> sub = gt.random_graph(10, lambda: (poisson(1.9), poisson(1.9)))
>>> vm, em = gt.subgraph_isomorphism(sub, g)
>>> print(len(vm))
35
>>> for i in range(len(vm)):
...   g.set_vertex_filter(None)
...   g.set_edge_filter(None)
...   vmask, emask = gt.mark_subgraph(g, sub, vm[i], em[i])
...   g.set_vertex_filter(vmask)
...   g.set_edge_filter(emask)
...   assert(gt.isomorphism(g, sub))
>>> g.set_vertex_filter(None)
>>> g.set_edge_filter(None)
>>> ewidth = g.copy_property(emask, value_type="double")
>>> ewidth.a += 0.5
>>> ewidth.a *= 2
>>> gt.graph_draw(g, vertex_fill_color=vmask, edge_color=emask,
...               edge_pen_width=ewidth, output_size=(200, 200),
...               output="subgraph-iso-embed.pdf")
<...>
>>> gt.graph_draw(sub, output_size=(200, 200), output="subgraph-iso.pdf")
<...>
```



Left: Subgraph searched, **Right:** One isomorphic subgraph found in main graph.

graph_tool.topology.**mark_subgraph**(*g*, *sub*, *vmap*, *emap*, *vmask=None*, *emask=None*)

Mark a given subgraph *sub* on the graph *g*.

The mapping must be provided by the *vmap* and *emap* parameters, which map vertices/edges of *sub* to indexes of the corresponding vertices/edges in *g*.

This returns a vertex and an edge property map, with value type 'bool', indicating whether or not a vertex/edge in *g* corresponds to the subgraph *sub*.

graph_tool.topology.**min_spanning_tree**(*g*, *weights=None*, *root=None*, *tree_map=None*)

Return the minimum spanning tree of a given graph.

Parameters **g** : Graph (page 28)

Graph to be used.

weights : PropertyMap (page 35) (optional, default: *None*)

The edge weights. If provided, the minimum spanning tree will minimize the edge weights.

root : Vertex (page 33) (optional, default: *None*)

Root of the minimum spanning tree. If this is provided, Prim's algorithm is used. Otherwise, Kruskal's algorithm is used.

tree_map : PropertyMap (page 35) (optional, default: *None*)

If provided, the edge tree map will be written in this property map.

Returns **tree_map** : PropertyMap (page 35)

Edge property map with mark the tree edges: 1 for tree edge, 0 otherwise.

Notes

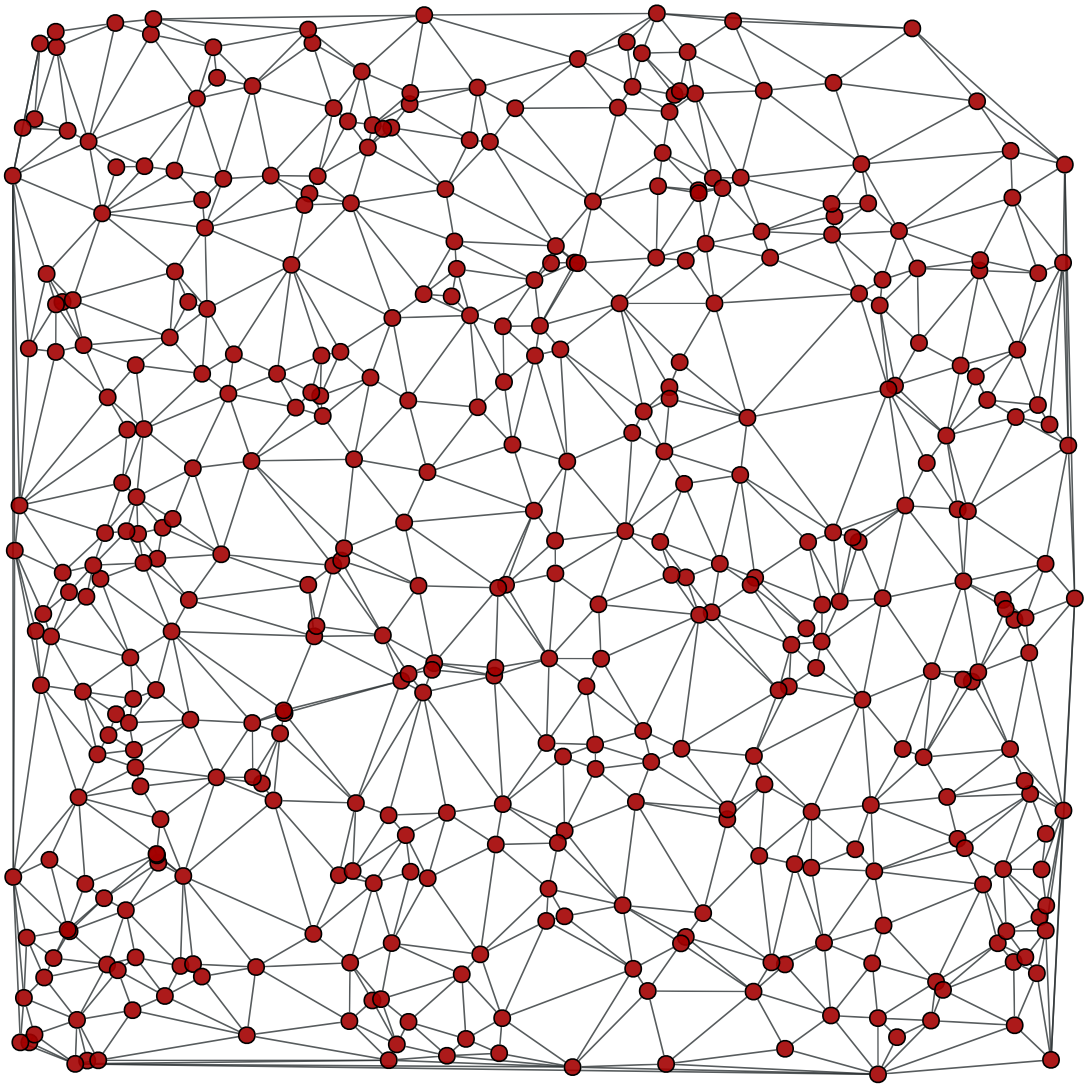
The algorithm runs with $O(E \log E)$ complexity, or $O(E \log V)$ if *root* is specified.

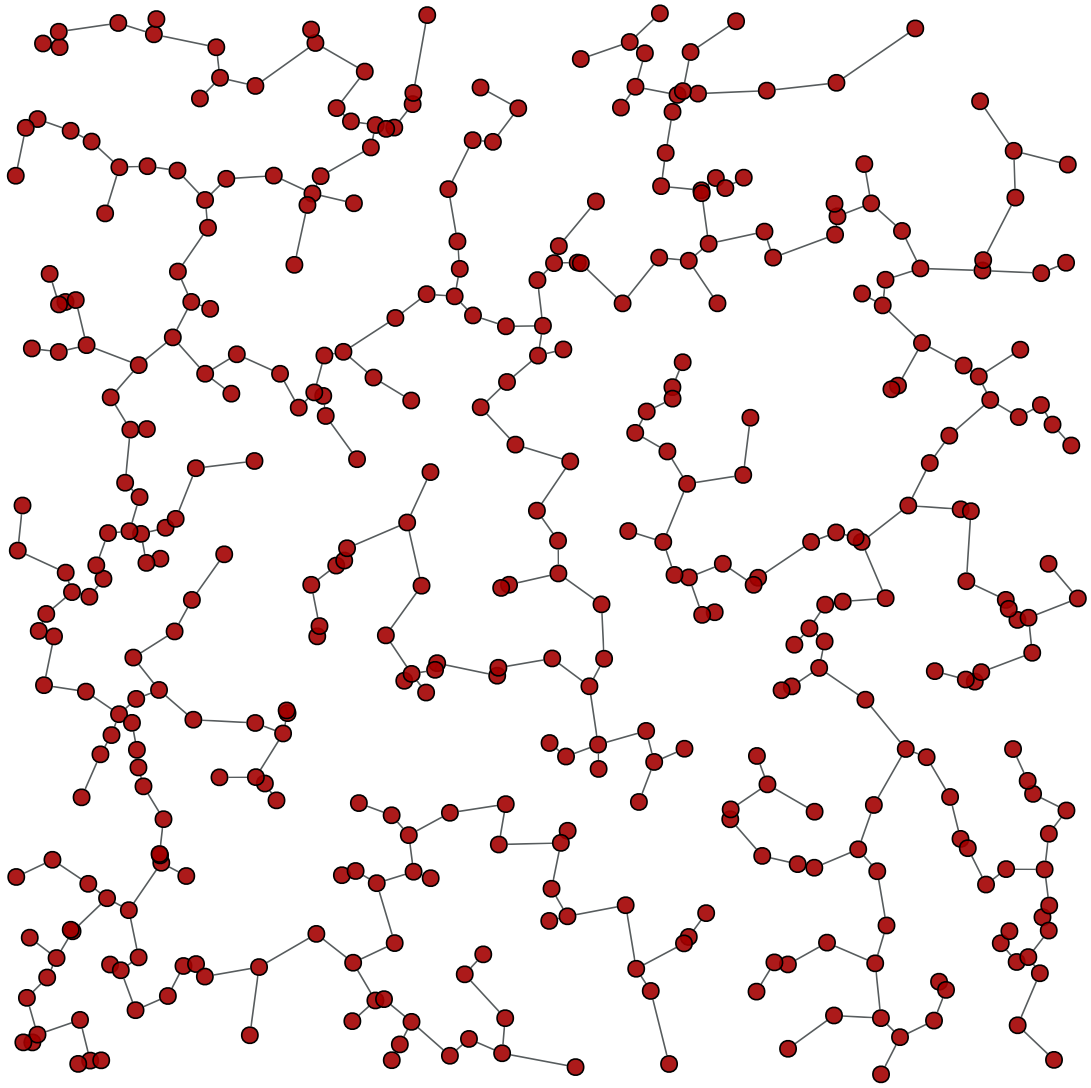
References

[kruskal-shortest-1956] (page 231), [prim-shortest-1957] (page 231), [boost-mst] (page 231), [mst-wiki] (page 231)

Examples

```
>>> from numpy.random import random
>>> g, pos = gt.triangulation(random((400, 2)) * 10, type="delaunay")
>>> weight = g.new_edge_property("double")
>>> for e in g.edges():
...     weight[e] = linalg.norm(pos[e.target()].a - pos[e.source()].a)
>>> tree = gt.min_spanning_tree(g, weights=weight)
>>> gt.graph_draw(g, pos=pos, output="triang_orig.pdf")
<...>
>>> g.set_edge_filter(tree)
>>> gt.graph_draw(g, pos=pos, output="triang_min_span_tree.pdf")
<...>
```





Left: Original graph, *Right:* The minimum spanning tree.

graph_tool.topology.**random_spanning_tree**(*g*, *weights=None*, *root=None*, *tree_map=None*)

Return a random spanning tree of a given graph, which can be directed or undirected.

Parameters **g** : Graph (page 28)

Graph to be used.

weights : PropertyMap (page 35) (optional, default: *None*)

The edge weights. If provided, the probability of a particular spanning tree being selected is the product of its edge weights.

root : Vertex (page 33) (optional, default: *None*)

Root of the spanning tree. If not provided, it will be selected randomly.

tree_map : PropertyMap (page 35) (optional, default: *None*)

If provided, the edge tree map will be written in this property map.

Returns **tree_map** : PropertyMap (page 35)

Edge property map with mark the tree edges: 1 for tree edge, 0 otherwise.

Notes

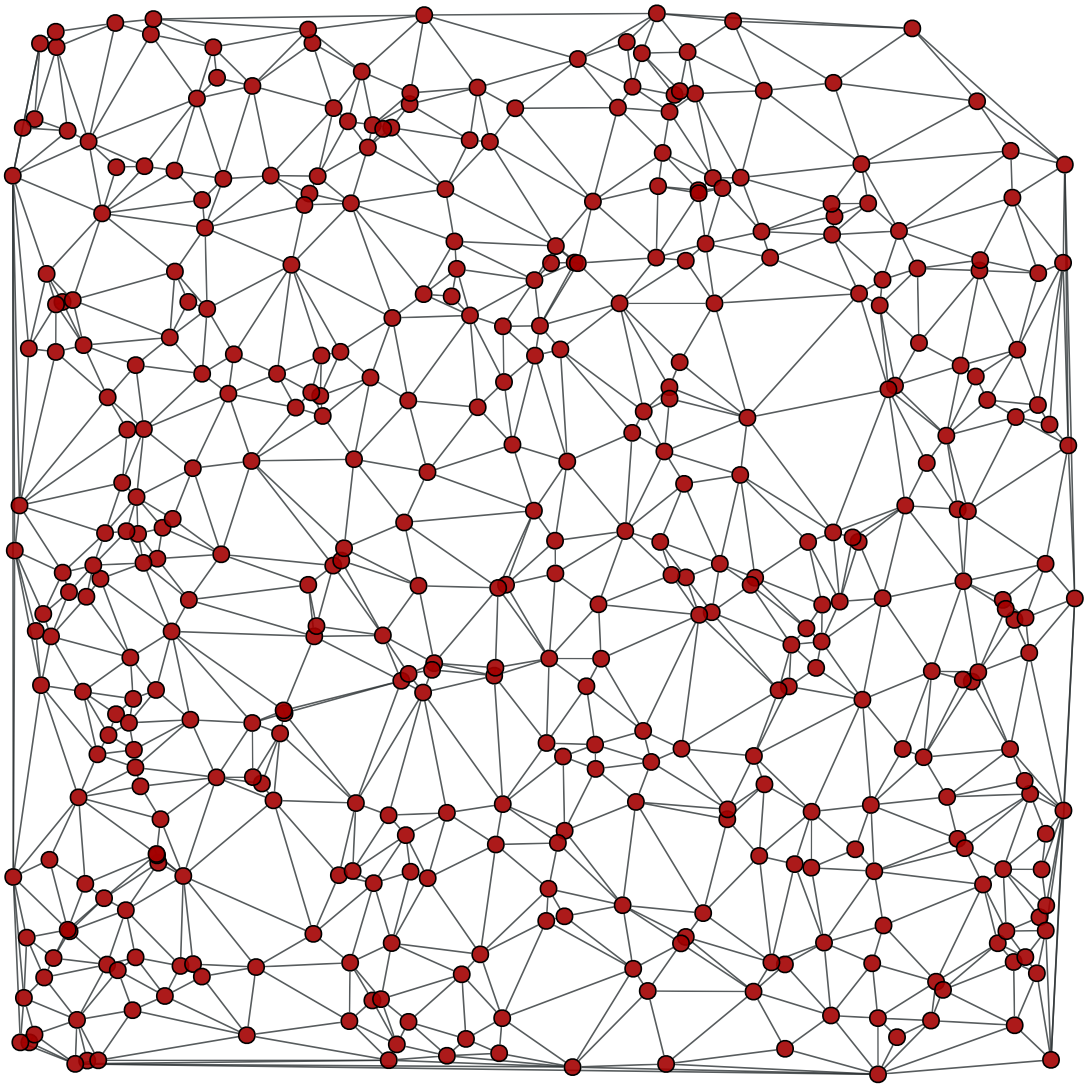
The typical running time for random graphs is $O(N \log N)$.

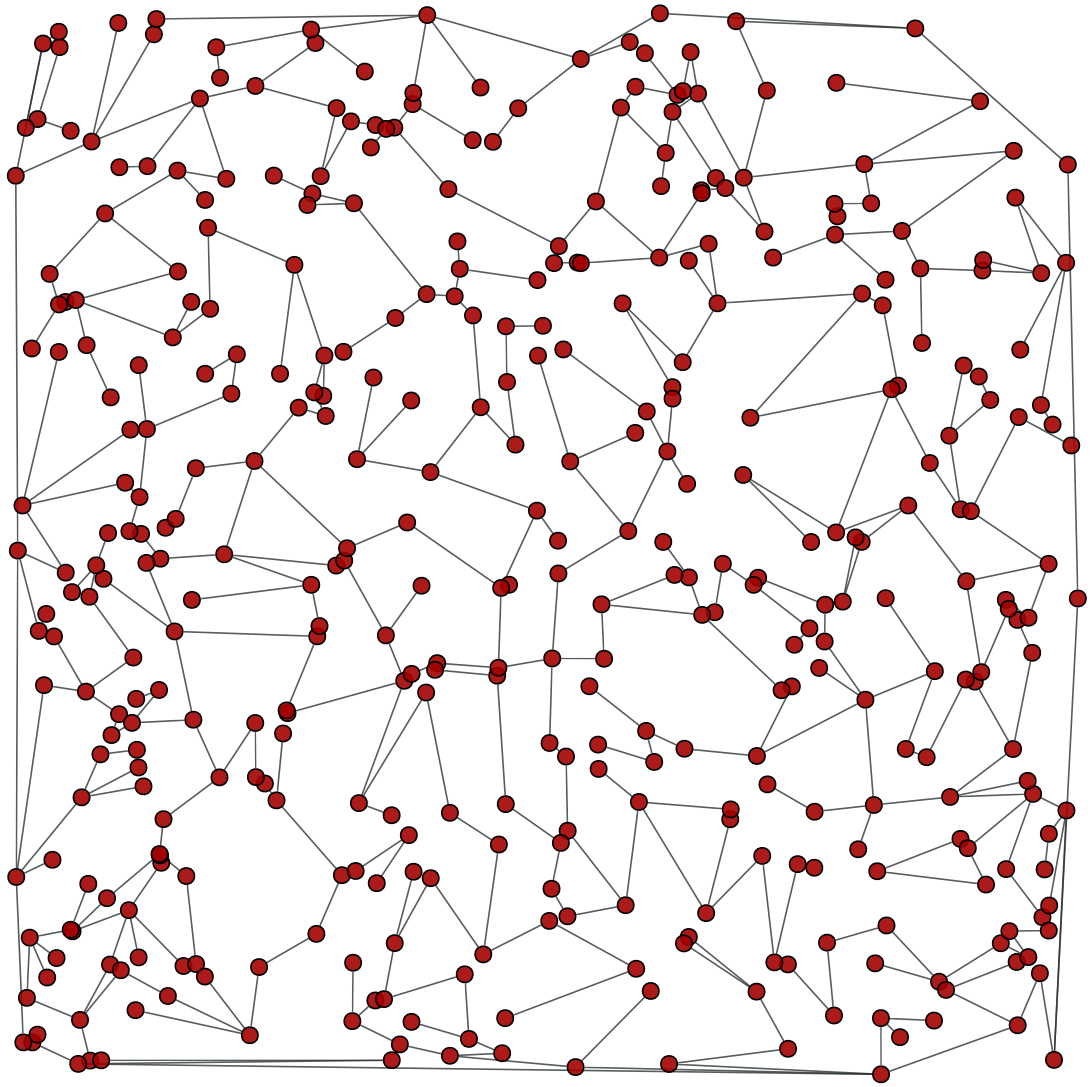
References

[wilson-generating-1996] (page 232), [boost-rst] (page 232)

Examples

```
>>> from numpy.random import random
>>> g, pos = gt.triangulation(random((400, 2)) * 10, type="delaunay")
>>> weight = g.new_edge_property("double")
>>> for e in g.edges():
...     weight[e] = linalg.norm(pos[e.target()].a - pos[e.source()].a)
>>> tree = gt.random_spanning_tree(g, weights=weight)
>>> gt.graph_draw(g, pos=pos, output="rtriang_orig.pdf")
<...>
>>> g.set_edge_filter(tree)
>>> gt.graph_draw(g, pos=pos, output="triang_random_span_tree.pdf")
<...>
```





Left: Original graph, *Right:* A random spanning tree.

graph_tool.topology.**dominator_tree**(*g*, *root*, *dom_map=None*)

Return a vertex property map the dominator vertices for each vertex.

Parameters **g** : Graph (page 28)

Graph to be used.

root : Vertex (page 33)

The root vertex.

dom_map : PropertyMap (page 35) (optional, default: None)

If provided, the dominator map will be written in this property map.

Returns **dom_map** : PropertyMap (page 35)

The dominator map. It contains for each vertex, the index of its dominator vertex.

Notes

A vertex u dominates a vertex v, if every path of directed graph from the entry to v must go through u.

The algorithm runs with $O((V + E)\log(V + E))$ complexity.

References

[dominator-bgl] (page 232)

Examples

```
>>> g = gt.random_graph(100, lambda: (2, 2))
>>> tree = gt.min_spanning_tree(g)
>>> g.set_edge_filter(tree)
>>> root = [v for v in g.vertices() if v.in_degree() == 0]
>>> dom = gt.dominator_tree(g, root[0])
>>> print(dom.a)
[ 0  0  0  0  0  0 62  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
```

graph_tool.topology.**topological_sort**(*g*)

Return the topological sort of the given graph. It is returned as an array of vertex indexes, in the sort order.

Notes

The topological sort algorithm creates a linear ordering of the vertices such that if edge (u,v) appears in the graph, then v comes before u in the ordering. The graph must be a directed acyclic graph (DAG).

The time complexity is $O(V + E)$.

References

[topological-boost] (page 232), [topological-wiki] (page 232)

Examples

```
>>> g = gt.random_graph(30, lambda: (3, 3))
>>> tree = gt.min_spanning_tree(g)
>>> g.set_edge_filter(tree)
>>> sort = gt.topological_sort(g)
>>> print(sort)
[ 1 14  2  7 17  0  3  4  5  6  8  9 22 10 11 12 13 16 23 27 15 18 19 20 21
 24 25 26 28 29]
```

graph_tool.topology.**transitive_closure**(*g*)

Return the transitive closure graph of g.

Notes

The transitive closure of a graph G = (V,E) is a graph G* = (V,E*) such that E* contains an edge (u,v) if and only if G contains a path (of at least one edge) from u to v. The transitive_closure() function transforms the input graph g into the transitive closure graph tc.

The time complexity (worst-case) is $O(VE)$.

References

[transitive-boost] (page 232), [transitive-wiki] (page 232)

Examples

```
>>> g = gt.random_graph(30, lambda: (3, 3))
>>> tc = gt.transitive_closure(g)
```

graph_tool.topology.**label_components**(*g*, *vprop=None*, *directed=None*, *attractors=False*)

Label the components to which each vertex in the graph belongs. If the graph is directed, it finds the strongly connected components.

A property map with the component labels is returned, together with an histogram of component labels.

Parameters **g** : Graph (page 28)

Graph to be used.

vprop : PropertyMap (page 35) (optional, default: None)

Vertex property to store the component labels. If none is supplied, one is created.

directed : bool (optional, default: None)

Treat graph as directed or not, independently of its actual directionality.

attractors : bool (optional, default: False)

If True, and the graph is directed, an additional array with Boolean values is returned, specifying if the strongly connected components are attractors or not.

Returns **comp** : PropertyMap (page 35)

Vertex property map with component labels.

hist : ndarray

Histogram of component labels.

is_attractor : ndarray

A Boolean array specifying if the strongly connected components are attractors or not. This returned only if attractors == True, and the graph is directed.

Notes

The components are arbitrarily labeled from 0 to N-1, where N is the total number of components.

The algorithm runs in $O(V + E)$ time.

Examples

```
>>> g = gt.random_graph(100, lambda: (poisson(2), poisson(2)))
>>> comp, hist, is_attractor = gt.label_components(g, attractors=True)
>>> print(comp.a)
[14 15 14 14 14  5 14 14 18 14 14  8 14 14 13 14 14 21 14 14  7 23 10 14 14
 14 24  4 14 14  0 14 14 14 25 14 14  1 14 26 14 19  9 14 14  3 14 14 27 28
 29 14 14  6 14 14 14 30 14 14 20 14  2 14 22 33 34 14 14 14 35 14 14 16 14
 11 36 37 14 14 31 14 14 17 14 14 14 14 14  0 14 38 39 32 14 12 14 40 14 14]
>>> print(hist)
[ 2  1  1  1  1  1  1  1  1  1  1  1  1  1 59  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1]
>>> print(is_attractor)
[ True  True  True False False False  True  True False False False False
  True  True False False False False False False False False  True False
 False False False False False False False False False False False False
 False False False False False]
```

graph_tool.topology.**label_largest_component**(*g*, *directed=None*)

Label the largest component in the graph. If the graph is directed, then the largest strongly connected component is labelled.

A property map with a boolean label is returned.

Parameters **g** : Graph (page 28)

Graph to be used.

directed : bool (optional, default:None)

Treat graph as directed or not, independently of its actual directionality.

Returns **comp** : PropertyMap (page 35)

Boolean vertex property map which labels the largest component.

Notes

The algorithm runs in $O(V + E)$ time.

Examples

```
>>> g = gt.random_graph(100, lambda: poisson(1), directed=False)
>>> l = gt.label_largest_component(g)
>>> print(l.a)
[1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0
 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0
 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
>>> u = gt.GraphView(g, vfilt=l)   # extract the largest component as a graph
>>> print(u.num_vertices())
16
```

graph_tool.topology.**label_out_component**(*g*, *root*)

Label the out-component (or simply the component for undirected graphs) of a root vertex.

Parameters **g** : Graph (page 28)

Graph to be used.

root : Vertex (page 33)

The root vertex.

Returns **comp** : PropertyMap (page 35)

Boolean vertex property map which labels the out-component.

Notes

The algorithm runs in $O(V + E)$ time.

Examples

```
>>> g = gt.random_graph(100, lambda: poisson(2.2), directed=False)
>>> l = gt.label_out_component(g, g.vertex(2))
>>> print(l.a)
[1 1 1 1 0 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 0 0 1 1 1 0
 1 1 0 0 1 1 0 1 1 0 0 1 1 1 1 0 1 0 0 1 1 1 1 1 1 1 1 1 0 0 1 0 1 1 1 1 1
 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 0 0]
```

The in-component can be obtained by reversing the graph.

```
>>> l = gt.label_out_component(gt.GraphView(g, reversed=True, directed=True),
...                            g.vertex(1))
>>> print(l.a)
[1 1 1 0 0 1 1 0 1 0 0 0 0 1 0 0 0 1 0 0 1 1 0 0 1 0 1 0 0 0 0 0 0 1 1 0 0
 0 1 0 0 0 0 1 0 1 1 0 0 0 0 1 1 0 0 0 0 1 0 0 0 1 1 0 1 0 0 0 1 0 0 1 1 0 1
 1 0 0 0 0 0 1 1 0 1 1 0 1 1 1 0 0 1 0 0 0 0 0 1 0 0 0]
```

graph_tool.topology.**label_biconnected_components**(*g*, *eprop=None*, *vprop=None*)

Label the edges of biconnected components, and the vertices which are articulation points.

An edge property map with the component labels is returned, together a boolean vertex map marking the articulation points, and an histogram of component labels.

Parameters **g** : Graph (page 28)

Graph to be used.

eprop : PropertyMap (page 35) (optional, default: None)

Edge property to label the biconnected components.

vprop : PropertyMap (page 35) (optional, default: None)

Vertex property to mark the articulation points. If none is supplied, one is created.

Returns **bicomp** : PropertyMap (page 35)

Edge property map with the biconnected component labels.

articulation : PropertyMap (page 35)

Boolean vertex property map which has value 1 for each vertex which is an articulation point, and zero otherwise.

nc : int

Number of biconnected components.

Notes

A connected graph is biconnected if the removal of any single vertex (and all edges incident on that vertex) can not disconnect the graph. More generally, the biconnected components of a graph are the maximal subsets of vertices such that the removal of a vertex from a particular component will not disconnect the component. Unlike connected components, vertices may belong to multiple biconnected components: those vertices that belong to more than one biconnected component are called "articulation points" or, equivalently, "cut vertices". Articulation points are vertices whose removal would increase the number of connected components in the graph. Thus, a graph without articulation points is biconnected. Vertices can be present in multiple biconnected components, but each edge can only be contained in a single biconnected component.

The algorithm runs in $O(V + E)$ time.

Examples

```
>>> g = gt.random_graph(100, lambda: poisson(2), directed=False)
>>> comp, art, hist = gt.label_biconnected_components(g)
>>> print(comp.a)
[51 51 51 51 51 51 11 52 51 51 44 42 41 45 49 23 19 51 51 32 38 51 24 37 51
 51 51 10  8 51 20 43 51 51 51 51 51 47 46 51 51 13 14 51 51 51 51 33 30 51
  1 21 51 51 51 35 36  6 51 26 27  7 12  4  3 29 28 51 51 51 31 51 51  0 39
 51 51 51 34 40 51 51  9 17 51 51 18 15 22  2 16 50  5 48 51 51 53 51 51 25]
>>> print(art.a)
[1 0 1 0 0 1 0 0 1 1 0 0 1 1 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 1 1 0
 0 1 0 0 1 1 0 1 1 0 0 0 1 1 0 1 1 0 0 0 1 0 0 1 0 1 0 0 1 1 0 0 0 0 0 1 1
 1 0 1 1 0 1 0 1 1 0 0 0 0 1 0 1 0 0 0 1 0 0 0 1 0 0]
>>> print(hist)
[ 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1 47  1  1]
```

graph_tool.topology.**kcore_decomposition**(*g*, *deg='out'*, *vprop=None*)

Perform a k-core decomposition of the given graph.

Parameters **g** : Graph (page 28)

Graph to be used.

deg : string

Degree to be used for the decomposition. It can be either "in", "out" or "total", for in-, out-, or total degree of the vertices.

vprop : PropertyMap (page 35) (optional, default: None)

Vertex property to store the decomposition. If None is supplied, one is created.

Returns **kval** : PropertyMap (page 35)

Vertex property map with the k-core decomposition, i.e. a given vertex v belongs to the kval[v]-core.

Notes

The k-core is a maximal set of vertices such that its induced subgraph only contains vertices with degree larger than or equal to k.

This algorithm is described in [batagelk-algorithm] (page 232) and runs in $O(V+E)$ time.

References

[k-core] (page 232), [batagelk-algorithm] (page 232)

Examples

```
>>> g = gt.collection.data["netscience"]
>>> g = gt.GraphView(g, vfilt=gt.label_largest_component(g))
>>> kcore = gt.kcore_decomposition(g)
>>> gt.graph_draw(g, pos=g.vp["pos"], vertex_fill_color=kcore, vertex_text=kcore, output="net
<...>
```

graph_tool.topology.**shortest_distance**(*g*, *source=None*, *target=None*, *weights=None*, *max_dist=None*, *directed=None*, *dense=False*, *dist_map=None*, *pred_map=False*)

Calculate the distance from a source to a target vertex, or to of all vertices from a given source, or the all pairs shortest paths, if the source is not specified.

Parameters **g** : Graph (page 28)

Graph to be used.

source : Vertex (page 33) (optional, default: None)

Source vertex of the search. If unspecified, the all pairs shortest distances are computed.

target : Vertex (page 33) (optional, default: None)

Target vertex of the search. If unspecified, the distance to all vertices from the source will be computed.

weights : PropertyMap (page 35) (optional, default: None)

The edge weights. If provided, the minimum spanning tree will minimize the edge weights.

max_dist : scalar value (optional, default: None)

If specified, this limits the maximum distance of the vertices searched. This parameter has no effect if source is None.

directed : bool (optional, default:None)

Treat graph as directed or not, independently of its actual directionality.

dense : bool (optional, default: False)

If true, and source is None, the Floyd-Warshall algorithm is used, otherwise the Johnson algorithm is used. If source is not None, this option has no effect.

dist_map : PropertyMap (page 35) (optional, default: None)

Vertex property to store the distances. If none is supplied, one is created.

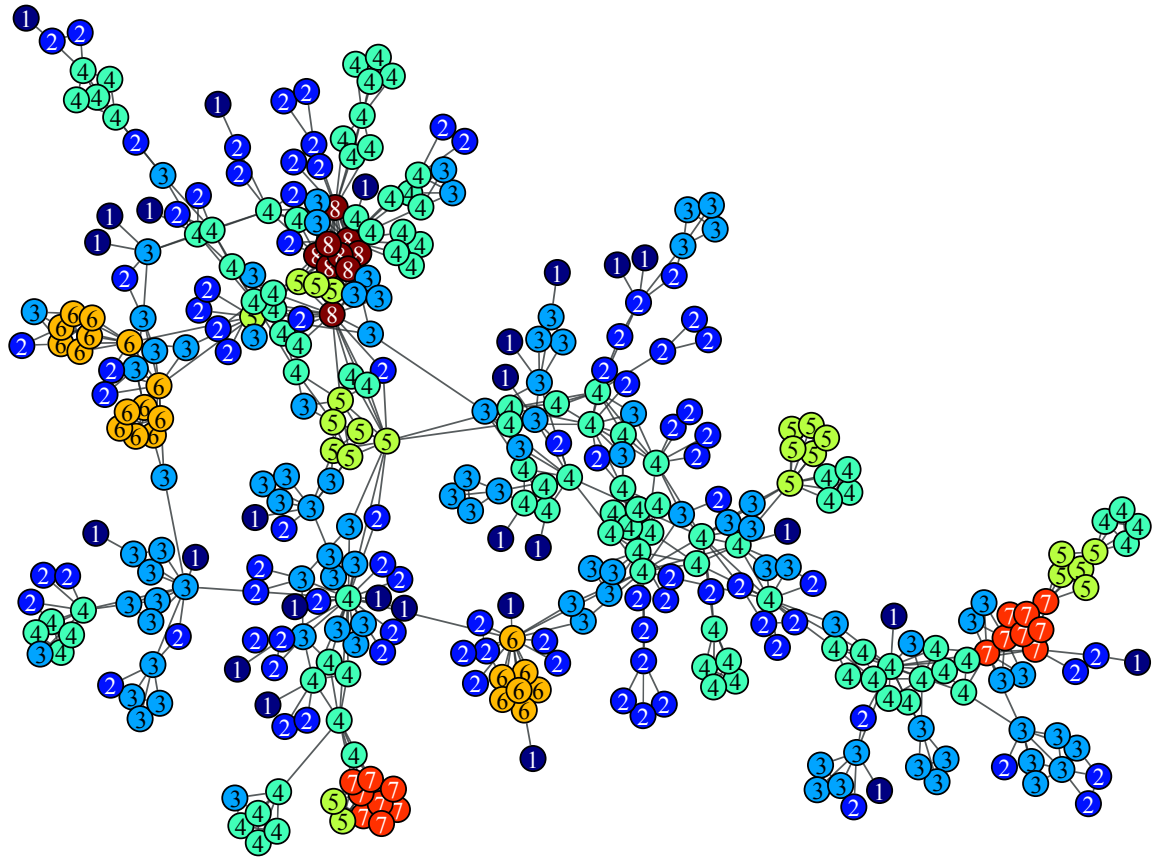


Figure 3.49: K-core decomposition of a network of network scientists.

pred_map : bool (optional, default: False)

If true, a vertex property map with the predecessors is returned. Ignored if source=None.

Returns **dist_map** : `PropertyMap` (page 35)

Vertex property map with the distances from source. If source is 'None', it will have a vector value type, with the distances to every vertex.

Notes

If a source is given, the distances are calculated with a breadth-first search (BFS) or Dijkstra's algorithm [dijkstra] (page **??**), if weights are given. If source is not given, the distances are calculated with Johnson's algorithm [johnson-apsp] (page 232). If dense=True, the Floyd-Warshall algorithm [floyd-warshall-apsp] (page 232) is used instead.

If source is specified, the algorithm runs in $O(V + E)$ time, or $O(V \log V)$ if weights are given. If source is not specified, it runs in $O(VE \log V)$ time, or $O(V^3)$ if dense == True.

References

[bfs] (page **??**), [bfs-boost] (page **??**), [dijkstra] (page **??**), [dijkstra-boost] (page **??**), [johnson-apsp] (page 232), [floyd-warshall-apsp] (page 232)

Examples

```
>>> from numpy.random import poisson
>>> g = gt.random_graph(100, lambda: (poisson(3), poisson(3)))
>>> dist = gt.shortest_distance(g, source=g.vertex(0))
>>> print(dist.a)
[          0           6           3           6 2147483647 2147483647
           6           5           2           4           5           6
           6           3           7           5           4           4
           3           4           2           4           3           3
           4           4           6           6           4           1
           5           2           4           5           3           5
           6           5           4           5 2147483647           9
           4           4           4           6           3           4
           6           6           3           2           4           4
           5           4           5           8           6           6
           5           5           4           5           6           3
           4           3           5           5 2147483647 2147483647
           5           5           8           3           7           4
           5           2           7           5           2           5
           5           5           7           7           4           3
           6           5           5           4           5           5
           4           4           6           5]

>>> dist = gt.shortest_distance(g)
>>> print(dist[g.vertex(0)].a)
[          0           6           3           6 2147483647 2147483647
           6           5           2           4           5           6
           6           3           7           5           4           4
           3           4           2           4           3           3
           4           4           6           6           4           1
           5           2           4           5           3           5
           6           5           4           5 2147483647           9
```

```
4          4          4          6          3          4
6          6          3          2          4          4
5          4          5          8          6          6
5          5          4          5          6          3
4          3          5          5 2147483647 2147483647
5          5          8          3          7          4
5          2          7          5          2          5
5          5          7          7          4          3
6          5          5          4          5          5
4          4          6          5]
```

graph_tool.topology.**shortest_path**(*g*, *source*, *target*, *weights=None*, *pred_map=None*)

Return the shortest path from *source* to *target*.

Parameters **g** : Graph (page 28)

Graph to be used.

source : Vertex (page 33)

Source vertex of the search.

target : Vertex (page 33)

Target vertex of the search.

weights : PropertyMap (page 35) (optional, default: None)

The edge weights.

pred_map : PropertyMap (page 35) (optional, default: None)

Vertex property map with the predecessors in the search tree. If this is provided, the shortest paths are not computed, and are obtained directly from this map.

Returns **vertex_list** : list of Vertex (page 33)

List of vertices from *source* to *target* in the shortest path.

edge_list : list of Edge (page 34)

List of edges from *source* to *target* in the shortest path.

Notes

The paths are computed with a breadth-first search (BFS) or Dijkstra's algorithm [dijkstra] (page **??**), if weights are given.

The algorithm runs in $O(V + E)$ time, or $O(V \log V)$ if weights are given.

References

[bfs] (page **??**), [bfs-boost] (page **??**), [dijkstra] (page **??**), [dijkstra-boost] (page **??**)

Examples

```
>>> from numpy.random import poisson
>>> g = gt.random_graph(300, lambda: (poisson(4), poisson(4)))
>>> vlist, elist = gt.shortest_path(g, g.vertex(10), g.vertex(11))
>>> print([str(v) for v in vlist])
['10', '131', '184', '265', '223', '11']
```

```
>>> print([str(e) for e in elist])
['(10, 131)', '(131, 184)', '(184, 265)', '(265, 223)', '(223, 11)']
```

graph_tool.topology.**pseudo_diameter**(*g*, *source=None*, *weights=None*)

Compute the pseudo-diameter of the graph.

Parameters **g** : Graph (page 28)

Graph to be used.

source : Vertex (page 33) (optional, default: *None*)

Source vertex of the search. If not supplied, the first vertex in the graph will be chosen.

weights : PropertyMap (page 35) (optional, default: *None*)

The edge weights.

Returns **pseudo_diameter** : int

The pseudo-diameter of the graph.

end_points : pair of Vertex (page 33)

The two vertices which correspond to the pseudo-diameter found.

Notes

The pseudo-diameter is an approximate graph diameter. It is obtained by starting from a vertex *source*, and finds a vertex *target* that is farthest away from *source*. This process is repeated by treating *target* as the new starting vertex, and ends when the graph distance no longer increases. A vertex from the last level set that has the smallest degree is chosen as the final starting vertex u, and a traversal is done to see if the graph distance can be increased. This graph distance is taken to be the pseudo-diameter.

The paths are computed with a breadth-first search (BFS) or Dijkstra's algorithm [dijkstra] (page **??**), if weights are given.

The algorithm runs in $O(V + E)$ time, or $O(V \log V)$ if weights are given.

References

[pseudo-diameter] (page 232)

Examples

```
>>> from numpy.random import poisson
>>> g = gt.random_graph(300, lambda: (poisson(3), poisson(3)))
>>> dist, ends = gt.pseudo_diameter(g)
>>> print(dist)
9.0
>>> print(int(ends[0]), int(ends[1]))
0 140
```

graph_tool.topology.**is_bipartite**(*g*, *partition=False*)

Test if the graph is bipartite.

Parameters **g** : Graph (page 28)

Graph to be used.

partition : bool (optional, default: False)

If `True`, return the two partitions in case the graph is bipartite.

Returns **is_bipartite** : bool

Whether or not the graph is bipartite.

partition : `PropertyMap` (page 35) (only if *partition=True*)

A vertex property map with the graph partitioning (or *None*) if the graph is not bipartite.

Notes

An undirected graph is bipartite if one can partition its set of vertices into two sets, such that all edges go from one set to the other.

This algorithm runs in $O(V + E)$ time.

References

[boost-bipartite] (page 232)

Examples

```
>>> g = gt.lattice([10, 10])
>>> is_bi, part = gt.is_bipartite(g, partition=True)
>>> print(is_bi)
True
>>> gt.graph_draw(g, vertex_fill_color=part, output_size=(300, 300), output="bipartite.pdf")
<...>
```

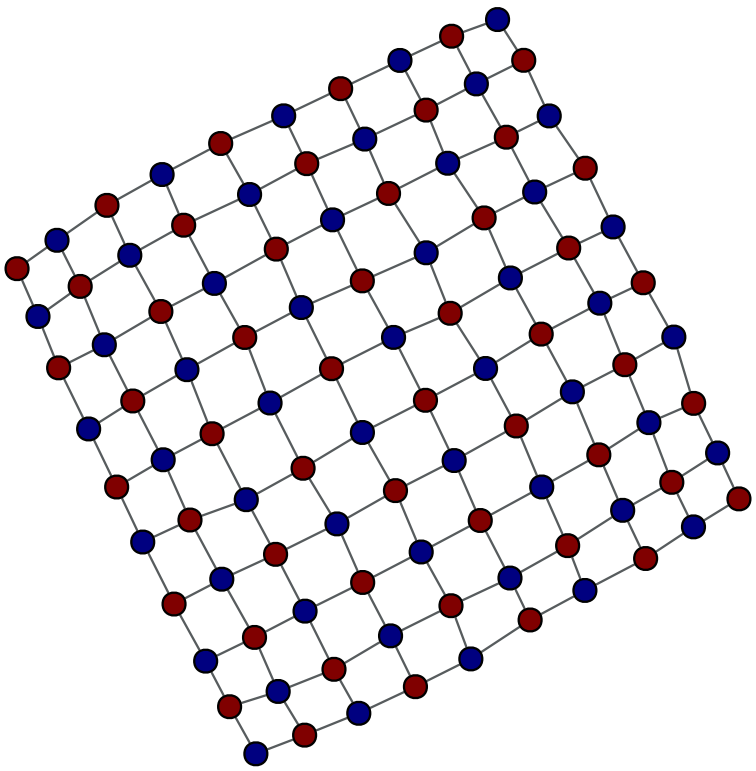


Figure 3.50: Bipartition of a 2D lattice.

graph_tool.topology.**is_planar**(*g*, *embedding=False*, *kuratowski=False*)

Test if the graph is planar.

Parameters **g** : Graph (page 28)

Graph to be used.

embedding : bool (optional, default: False)

If true, return a mapping from vertices to the clockwise order of out-edges in the planar embedding.

kuratowski : bool (optional, default: False)

If true, the minimal set of edges that form the obstructing Kuratowski subgraph will be returned as a property map, if the graph is not planar.

Returns **is_planar** : bool

Whether or not the graph is planar.

embedding : PropertyMap (page 35) (only if *embedding=True*)

A vertex property map with the out-edges indexes in clockwise order in the planar embedding,

kuratowski : PropertyMap (page 35) (only if *kuratowski=True*)

An edge property map with the minimal set of edges that form the obstructing Kuratowski subgraph (if the value of kuratowski[e] is 1, the edge belongs to the set)

Notes

A graph is planar if it can be drawn in two-dimensional space without any of its edges crossing. This algorithm performs the Boyer-Myrvold planarity testing [boyer-myrvold] (page 232). See [boost-planarity] (page **??**) for more details.

This algorithm runs in $O(V)$ time.

References

[boyer-myrvold] (page 232), [boost-planarity] (page **??**)

Examples

```
>>> from numpy.random import random
>>> g = gt.triangulation(random((100,2)))[0]
>>> p, embed_order = gt.is_planar(g, embedding=True)
>>> print(p)
True
>>> print(list(embed_order[g.vertex(0)]))
[0, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> g = gt.random_graph(100, lambda: 4, directed=False)
>>> p, kur = gt.is_planar(g, kuratowski=True)
>>> print(p)
False
>>> g.set_edge_filter(kur, True)
>>> gt.graph_draw(g, output_size=(300, 300), output="kuratowski.pdf")
<...>
```

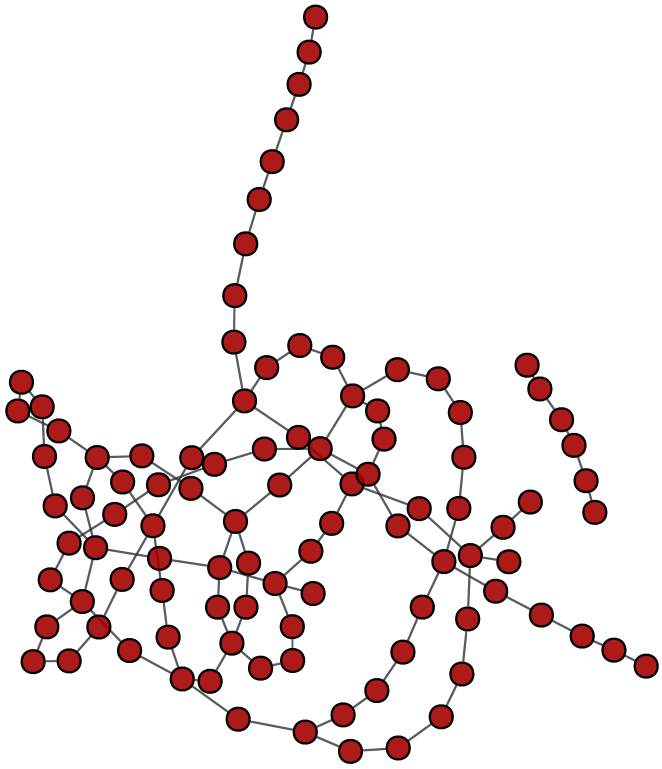


Figure 3.51: Obstructing Kuratowski subgraph of a random graph.

graph_tool.topology.**make_maximal_planar**(*g*, *unfilter=False*)

Add edges to the graph to make it maximally planar.

Parameters **g** : Graph (page 28)

Graph to be used. It must be a biconnected planar graph with at least 3 vertices.

Notes

A graph is maximal planar if no additional edges can be added to it without creating a non-planar graph. By Euler's formula, a maximal planar graph with V > 2 vertices always has 3V - 6 edges and 2V - 4 faces.

The input graph to make_maximal_planar() must be a biconnected planar graph with at least 3 vertices.

This algorithm runs in $O(V + E)$ time.

References

[boost-planarity] (page **??**)

Examples

```
>>> g = gt.lattice([42, 42])
>>> gt.make_maximal_planar(g)
>>> gt.is_planar(g)
True
```

```
>>> print(g.num_vertices(), g.num_edges())
1764 5286
>>> gt.graph_draw(g, output_size=(300, 300), output="maximal_planar.pdf")
<...>
```

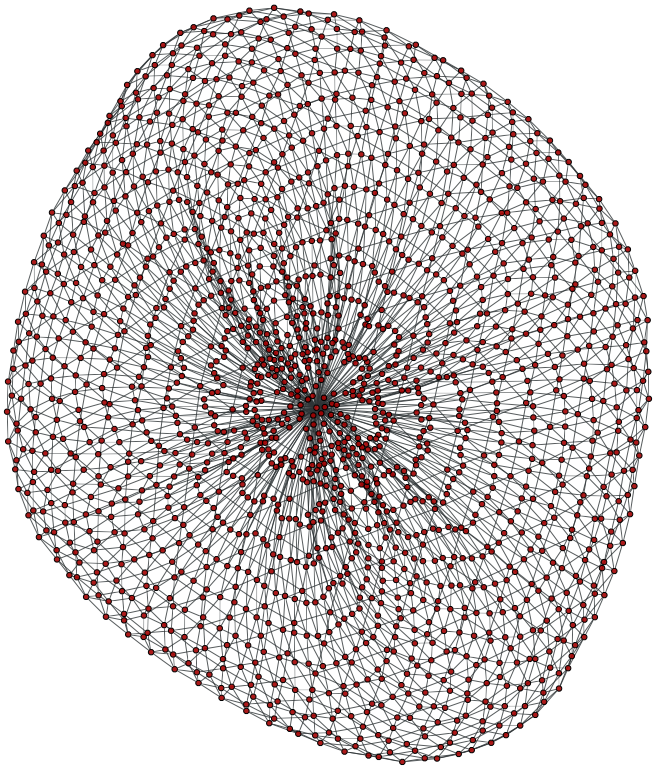


Figure 3.52: A maximally planar graph.

graph_tool.topology.**is_DAG**(*g*)

Return *True* if the graph is a directed acyclic graph (DAG).

Notes

The time complexity is $O(V + E)$.

References

[DAG-wiki] (page 232)

Examples

```
>>> g = gt.random_graph(30, lambda: (3, 3))
>>> print(gt.is_DAG(g))
False
>>> tree = gt.min_spanning_tree(g)
>>> g.set_edge_filter(tree)
>>> print(gt.is_DAG(g))
True
```

graph_tool.topology.**max_cardinality_matching**(*g*, *heuristic=False*, *weight=None*, *minimize=True*, *match=None*)

Find a maximum cardinality matching in the graph.

Parameters **g** : Graph (page 28)

Graph to be used.

heuristic : bool (optional, default: *False*)

If true, a random heuristic will be used, which runs in linear time.

weight : PropertyMap (page 35) (optional, default: *None*)

If provided, the matching will minimize the edge weights (or maximize if `minimize == False`). This option has no effect if `heuristic == False`.

minimize : bool (optional, default: *True*)

If *True*, the matching will minimize the weights, otherwise they will be maximized. This option has no effect if `heuristic == False`.

match : PropertyMap (page 35) (optional, default: *None*)

Edge property map where the matching will be specified.

Returns **match** : PropertyMap (page 35)

Boolean edge property map where the matching is specified.

Notes

A *matching* is a subset of the edges of a graph such that no two edges share a common vertex. A *maximum cardinality matching* has maximum size over all matchings in the graph.

If the parameter `weight` is provided, as well as `heuristic == True` a matching with maximum cardinality *and* maximum (or minimum) weight is returned.

If `heuristic == True` the algorithm does not necessarily return the maximum matching, instead the focus is to run on linear time.

This algorithm runs in time $O(EV \times \alpha(E,V))$, where $\alpha(m,n)$ is a slow growing function that is at most 4 for any feasible input. If *heuristic* == *True*, the algorithm runs in time $O(V+E)$.

For a more detailed description, see [boost-max-matching] (page 232).

References

[boost-max-matching] (page 232), [matching-heuristic] (page 232)

Examples

```
>>> g = gt.GraphView(gt.price_network(300), directed=False)
>>> res = gt.max_cardinality_matching(g)
>>> print(res[1])
True
>>> w = res[0].copy("double")
>>> w.a = 2 * w.a + 2
>>> gt.graph_draw(g, edge_color=res[0], edge_pen_width=w, vertex_fill_color="grey",
...               output="max_card_match.pdf")
<...>
```

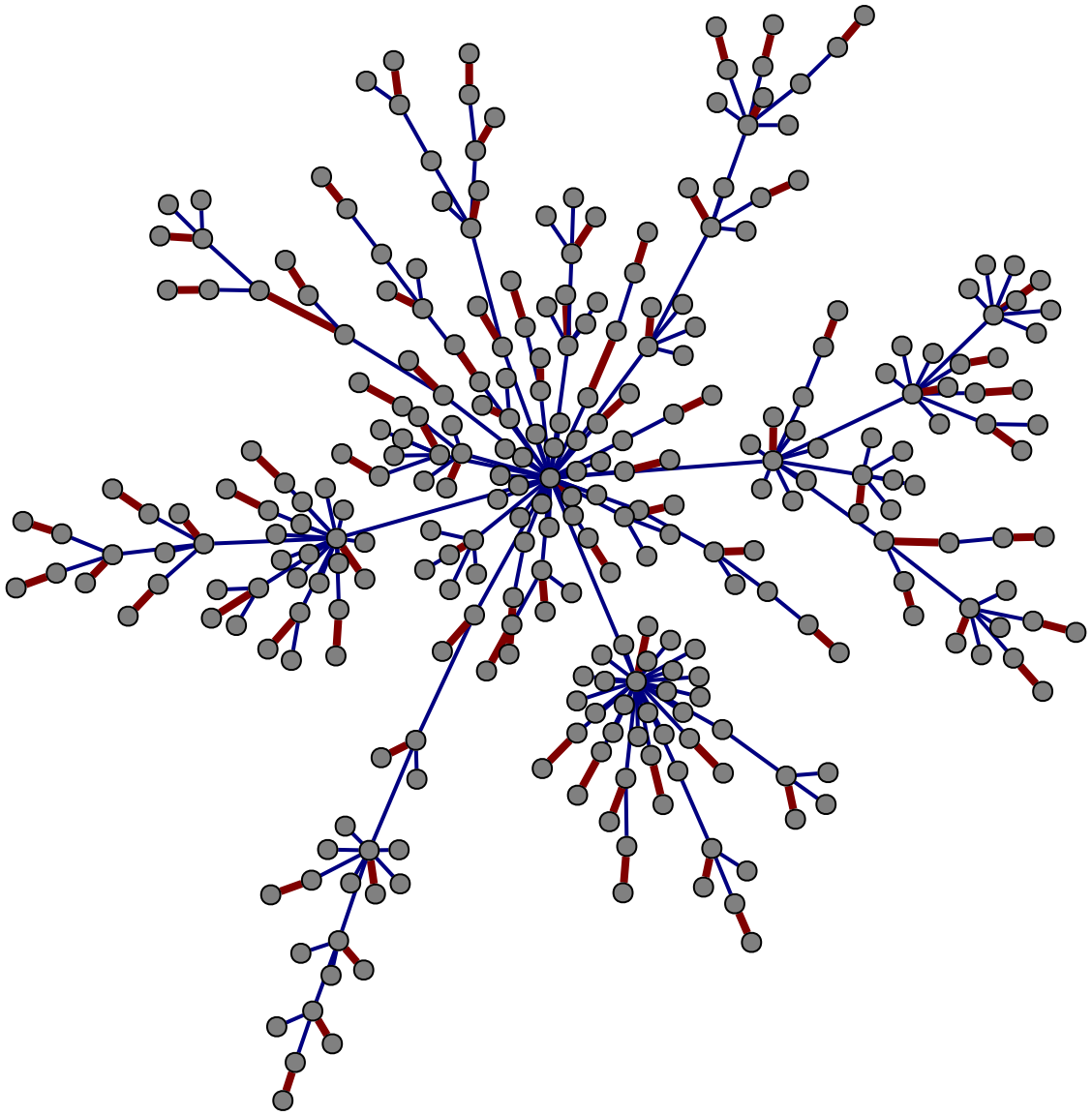


Figure 3.53: Edges belonging to the matching are in red.

graph_tool.topology.**max_independent_vertex_set**(*g*, *high_deg=False*, *mivs=None*)

Find a maximal independent vertex set in the graph.

Parameters **g** : Graph (page 28)

Graph to be used.

high_deg : bool (optional, default: *False*)

If *True*, vertices with high degree will be included first in the set, otherwise they will be included last.

mivs : PropertyMap (page 35) (optional, default: *None*)

Vertex property map where the vertex set will be specified.

Returns **mivs** : PropertyMap (page 35)

Boolean vertex property map where the set is specified.

Notes

A maximal independent vertex set is an independent set such that adding any other vertex to the set forces the set to contain an edge between two vertices of the set.

This implements the algorithm described in [mivs-luby] (page 232), which runs in time $O(V + E)$.

References

[mivs-wikipedia] (page 232), [mivs-luby] (page 232)

Examples

```
>>> g = gt.GraphView(gt.price_network(300), directed=False)
>>> res = gt.max_independent_vertex_set(g)
>>> gt.graph_draw(g, vertex_fill_color=res, output="mivs.pdf")
<...>
```

graph_tool.topology.**edge_reciprocity**(*g*)

Calculate the edge reciprocity of the graph.

Parameters **g** : Graph (page 28)

Graph to be used edges.

Returns **reciprocity** : float

The reciprocity value.

Notes

The edge [reciprocity] (page 233) is defined as $E^{\leftrightarrow}/E$, where $E^{\leftrightarrow}$ and E are the number of bidirectional and all edges in the graph, respectively.

The algorithm runs with complexity $O(E + V)$.

References

[reciprocity] (page 233), [lopez-reciprocity-2007] (page 233)

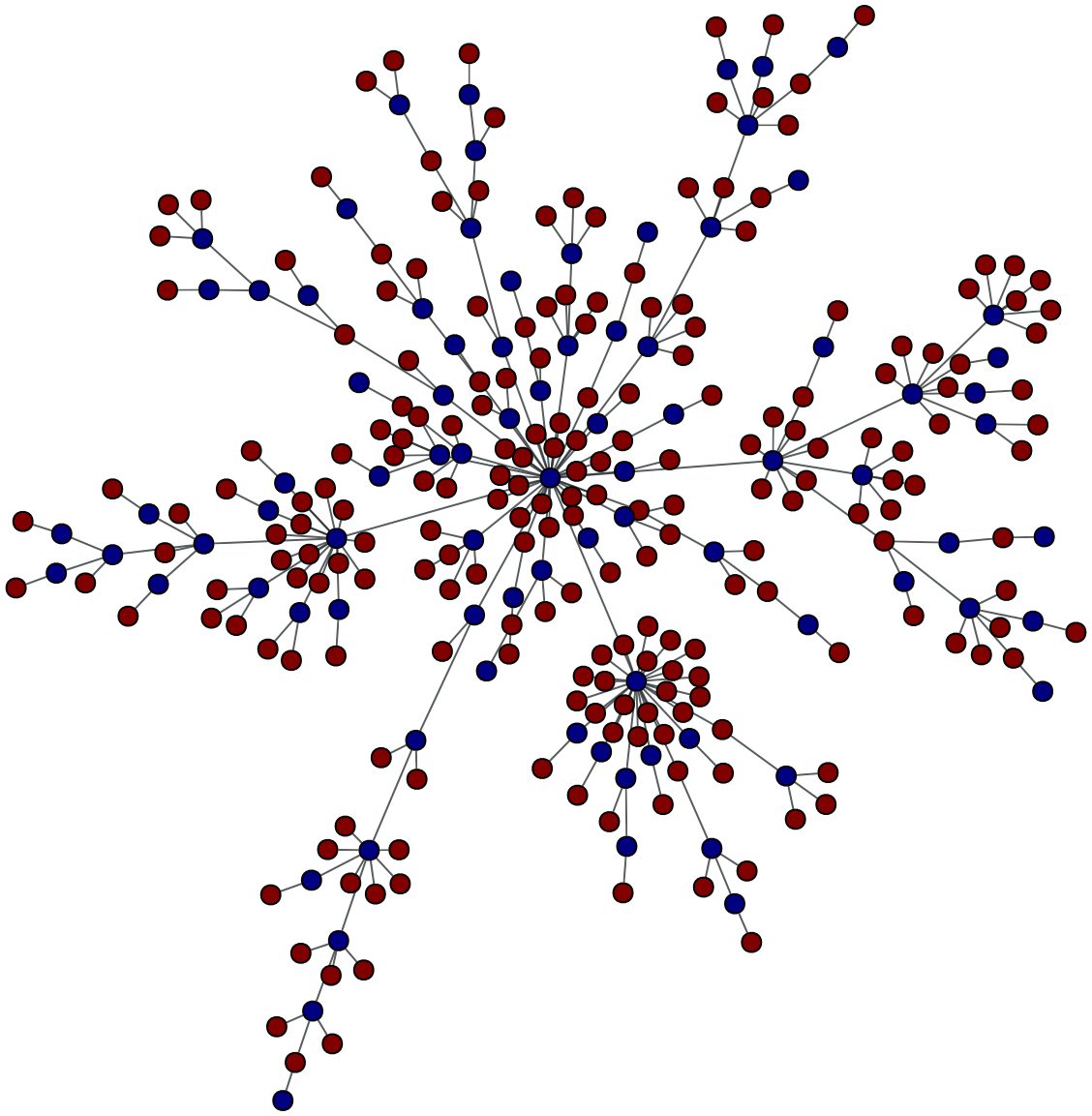


Figure 3.54: Vertices belonging to the set are in red.

Examples

```
>>> g = gt.Graph()
>>> g.add_vertex(2)
<...>
>>> g.add_edge(g.vertex(0), g.vertex(1))
<Edge object with source '0' and target '1' at 0x33bc710>
>>> gt.edge_reciprocity(g)
0.0
>>> g.add_edge(g.vertex(1), g.vertex(0))
<Edge object with source '1' and target '0' at 0x33bc7a0>
>>> gt.edge_reciprocity(g)
1.0
```

graph_tool.topology.**tsp_tour**(*g*, *src*, *weight=None*)

Return a traveling salesman tour of the graph, which is guaranteed to be twice as long as the optimal tour in the worst case.

Parameters **g** : Graph (page 28)

Graph to be used.

src : Vertex (page 33)

The source (and target) of the tour.

weight : PropertyMap (page 35) (optional, default: None)

Edge weights.

Returns **tour** : numpy.ndarray

List of vertex indexes corresponding to the tour.

Notes

The algorithm runs with $O(E \log V)$ complexity.

References

[tsp-bgl] (page 233), [tsp] (page 233)

Examples

```
>>> g = gt.lattice([10, 10])
>>> tour = gt.tsp_tour(g, g.vertex(0))
>>> print(tour)
[ 0  1  2 11 12 21 22 31 32 41 42 51 52 61 62 71 72 81 82 83 73 63 53 43 33
 23 13  3  4  5  6  7  8  9 19 29 39 49 59 69 79 89 14 24 34 44 54 64 74 84
 91 92 93 94 95 85 75 65 55 45 35 25 15 16 17 18 27 28 37 38 47 48 57 58 67
 68 77 78 87 88 97 98 99 26 36 46 56 66 76 86 96 10 20 30 40 50 60 70 80 90
  0]
```

graph_tool.topology.**sequential_vertex_coloring**(*g*, *order=None*, *color=None*)

Returns a vertex coloring of the graph.

Parameters **g** : Graph (page 28)

Graph to be used.

order : PropertyMap (page 35) (optional, default: None)

Order with which the vertices will be colored.

color : PropertyMap (page 35) (optional, default: None)

Integer-valued vertex property map to store the colors.

Returns **color** : PropertyMap (page 35)

Integer-valued vertex property map with the vertex colors.

Notes

The time complexity is $O(V(d+k))$, where V is the number of vertices, d is the maximum degree of the vertices in the graph, and k is the number of colors used.

References

[sgc-bgl] (page 233), [graph-coloring] (page 233)

Examples

```
>>> g = gt.lattice([10, 10])
>>> colors = gt.sequential_vertex_coloring(g)
>>> print(colors.a)
[0 1 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1
 0 1 0 0 1 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 1 0 1 0
 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1 0 1 0]
```

3.2.14 graph_tool.util - Graph utilities

Summary

find_vertex (page 223)	Find all vertices *v* for which *prop[v]* = *match*.
find_vertex_range (page 223)	Find all vertices *v* for which *range[0]* <= *prop[v]* <= *range[1]*.
find_edge (page 223)	Find all vertices *e* for which *prop[e]* = *match*.
find_edge_range (page 223)	Find all vertices *e* for which *range[0]* <= *prop[e]* <= *range[1]*.

Contents

graph_tool.util.**find_vertex**(*g*, *prop*, *match*)

Find all vertices *v* for which *prop[v]* = *match*. The parameter prop can be either a PropertyMap (page 35) or string with value "in", "out" or "total", representing a degree type.

graph_tool.util.**find_vertex_range**(*g*, *prop*, *range*)

Find all vertices *v* for which *range[0]* <= *prop[v]* <= *range[1]*. The parameter prop can be either a PropertyMap (page 35) or string with value"in", "out" or "total", representing a degree type.

graph_tool.util.**find_edge**(*g*, *prop*, *match*)

Find all vertices *e* for which *prop[e]* = *match*. The parameter prop must be a PropertyMap (page 35).

graph_tool.util.**find_edge_range**(*g*, *prop*, *range*)

Find all vertices *e* for which *range[0]* <= *prop[e]* <= *range[1]*. The parameter prop can be either a PropertyMap (page 35).

CHAPTER

FOUR

INDEXES AND TABLES

- *genindex*
- *modindex*
- *search*

BIBLIOGRAPHY

[pagerank-wikipedia] http://en.wikipedia.org/wiki/Pagerank

[lawrence-pagerank-1998] P. Lawrence, B. Sergey, M. Rajeev, W. Terry, "The pagerank citation ranking: Bringing order to the web", Technical report, Stanford University, 1998

[Langville-survey-2005] A. N. Langville, C. D. Meyer, "A Survey of Eigenvector Methods for Web Information Retrieval", SIAM Review, vol. 47, no. 1, pp. 135-161, 2005, DOI: 10.1137/S0036144503424786

[adamic-polblogs] L. A. Adamic and N. Glance, "The political blogosphere and the 2004 US Election", in Proceedings of the WWW-2005 Workshop on the Weblogging Ecosystem (2005). DOI: 10.1145/1134271.1134277

[betweenness-wikipedia] http://en.wikipedia.org/wiki/Centrality#Betweenness_centrality

[brandes-faster-2001] U. Brandes, "A faster algorithm for betweenness centrality", Journal of Mathematical Sociology, 2001, DOI: 10.1080/0022250X.2001.9990249

[adamic-polblogs] L. A. Adamic and N. Glance, "The political blogosphere and the 2004 US Election", in Proceedings of the WWW-2005 Workshop on the Weblogging Ecosystem (2005). DOI: 10.1145/1134271.1134277

[closeness-wikipedia] https://en.wikipedia.org/wiki/Closeness_centrality

[opsahl-node-2010] Opsahl, T., Agneessens, F., Skvoretz, J., "Node centrality in weighted networks: Generalizing degree and shortest paths". Social Networks 32, 245-251, 2010 DOI: 10.1016/j.socnet.2010.03.006

[adamic-polblogs] L. A. Adamic and N. Glance, "The political blogosphere and the 2004 US Election", in Proceedings of the WWW-2005 Workshop on the Weblogging Ecosystem (2005). DOI: 10.1145/1134271.1134277

[freeman-set-1977] Linton C. Freeman, "A Set of Measures of Centrality Based on Betweenness", Sociometry, Vol. 40, No. 1, pp. 35-41, 1977, http://www.jstor.org/stable/3033543

[eigenvector-centrality] http://en.wikipedia.org/wiki/Centrality#Eigenvector_centrality

[power-method] http://en.wikipedia.org/wiki/Power_iteration

[langville-survey-2005] A. N. Langville, C. D. Meyer, "A Survey of Eigenvector Methods for Web Information Retrieval", SIAM Review, vol. 47, no. 1, pp. 135-161, 2005, DOI: 10.1137/S0036144503424786

[adamic-polblogs] L. A. Adamic and N. Glance, "The political blogosphere and the 2004 US Election", in Proceedings of the WWW-2005 Workshop on the Weblogging Ecosystem (2005). DOI: 10.1145/1134271.1134277

[katz-centrality] http://en.wikipedia.org/wiki/Katz_centrality

[katz-new] L. Katz, "A new status index derived from sociometric analysis", Psychometrika 18, Number 1, 39-43, 1953, DOI: 10.1007/BF02289026

[adamic-polblogs] L. A. Adamic and N. Glance, "The political blogosphere and the 2004 US Election", in Proceedings of the WWW-2005 Workshop on the Weblogging Ecosystem (2005). DOI: 10.1145/1134271.1134277

[hits-algorithm] http://en.wikipedia.org/wiki/HITS_algorithm

[kleinberg-authoritative] J. Kleinberg, "Authoritative sources in a hyperlinked environment", Journal of the ACM 46 (5): 604-632, 1999, DOI: 10.1145/324133.324140.

[power-method] http://en.wikipedia.org/wiki/Power_iteration

[adamic-polblogs] L. A. Adamic and N. Glance, "The political blogosphere and the 2004 US Election", in Proceedings of the WWW-2005 Workshop on the Weblogging Ecosystem (2005). DOI: 10.1145/1134271.1134277

[kamvar-eigentrust-2003] S. D. Kamvar, M. T. Schlosser, H. Garcia-Molina "The eigentrust algorithm for reputation management in p2p networks", Proceedings of the 12th international conference on World Wide Web, Pages: 640 - 651, 2003, DOI: 10.1145/775152.775242

[adamic-polblogs] L. A. Adamic and N. Glance, "The political blogosphere and the 2004 US Election", in Proceedings of the WWW-2005 Workshop on the Weblogging Ecosystem (2005). DOI: 10.1145/1134271.1134277

[richters-trust-2010] Oliver Richters and Tiago P. Peixoto, "Trust Transitivity in Social Networks," PLoS ONE 6, no. 4: e1838 (2011), DOI: 10.1371/journal.pone.0018384

[adamic-polblogs] L. A. Adamic and N. Glance, "The political blogosphere and the 2004 US Election", in Proceedings of the WWW-2005 Workshop on the Weblogging Ecosystem (2005). DOI: 10.1145/1134271.1134277

[watts-collective-1998] D. J. Watts and Steven Strogatz, "Collective dynamics of 'small-world' networks", Nature, vol. 393, pp 440-442, 1998. DOI: 10.1038/30918

[newman-structure-2003] M. E. J. Newman, "The structure and function of complex networks", SIAM Review, vol. 45, pp. 167-256, 2003, DOI: 10.1137/S003614450342480

[abdo-clustering] A. H. Abdo, A. P. S. de Moura, "Clustering as a measure of the local topology of networks", arXiv: physics/0605235

[wernicke-efficient-2006] S. Wernicke, "Efficient detection of network motifs", IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB), Volume 3, Issue 4, Pages 347-359, 2006. DOI: 10.1109/TCBB.2006.51

[holland-stochastic-1983] Paul W. Holland, Kathryn Blackmond Laskey, Samuel Leinhardt, "Stochastic blockmodels: First steps", Carnegie-Mellon University, Pittsburgh, PA 15213, U.S.A., DOI: 10.1016/0378-8733(83)90021-7

[faust-blockmodels-1992] Katherine Faust, and Stanley Wasserman. "Blockmodels: Interpretation and Evaluation." Social Networks 14, no. 1-2 (1992): 5-61. DOI: 10.1016/0378-8733(92)90013-W

[karrer-stochastic-2011] Brian Karrer, and M. E. J. Newman. "Stochastic Blockmodels and Community Structure in Networks." Physical Review E 83, no. 1 (2011): 016107. DOI: 10.1103/PhysRevE.83.016107.

[peixoto-entropy-2012] Tiago P. Peixoto "Entropy of Stochastic Blockmodel Ensembles." Physical Review E 85, no. 5 (2012): 056122. DOI: 10.1103/PhysRevE.85.056122, arXiv: 1112.6028.

[peixoto-parsimonious-2013] Tiago P. Peixoto, "Parsimonious module inference in large networks", Phys. Rev. Lett. 110, 148701 (2013), DOI: 10.1103/PhysRevLett.110.148701, arXiv: 1212.4794.

[holland-stochastic-1983] Paul W. Holland, Kathryn Blackmond Laskey, Samuel Leinhardt, "Stochastic blockmodels: First steps", Carnegie-Mellon University, Pittsburgh, PA 15213, U.S.A., DOI: 10.1016/0378-8733(83)90021-7

[faust-blockmodels-1992] Katherine Faust, and Stanley Wasserman. "Blockmodels: Interpretation and Evaluation." Social Networks 14, no. 1-2 (1992): 5-61. DOI: 10.1016/0378-8733(92)90013-W

[karrer-stochastic-2011] Brian Karrer, and M. E. J. Newman. "Stochastic Blockmodels and Community Structure in Networks." Physical Review E 83, no. 1 (2011): 016107. DOI: 10.1103/PhysRevE.83.016107.

[peixoto-entropy-2012] Tiago P. Peixoto "Entropy of Stochastic Blockmodel Ensembles." Physical Review E 85, no. 5 (2012): 056122. DOI: 10.1103/PhysRevE.85.056122, arXiv: 1112.6028.

[peixoto-parsimonious-2013] Tiago P. Peixoto, "Parsimonious module inference in large networks", Phys. Rev. Lett. 110, 148701 (2013), DOI: 10.1103/PhysRevLett.110.148701, arXiv: 1212.4794.

[mezard-information-2009] Marc Mézard, Andrea Montanari, "Information, Physics, and Computation", Oxford Univ Press, 2009.

[mezard-information-2009] Marc Mézard, Andrea Montanari, "Information, Physics, and Computation", Oxford Univ Press, 2009.

[peixoto-parsimonious-2013] Tiago P. Peixoto, "Parsimonious module inference in large networks", Phys. Rev. Lett. 110, 148701 (2013), DOI: 10.1103/PhysRevLett.110.148701, arXiv: 1212.4794.

[peixoto-parsimonious-2013] Tiago P. Peixoto, "Parsimonious module inference in large networks", Phys. Rev. Lett. 110, 148701 (2013), DOI: 10.1103/PhysRevLett.110.148701, arXiv: 1212.4794.

[peixoto-parsimonious-2013] Tiago P. Peixoto, "Parsimonious module inference in large networks", Phys. Rev. Lett. 110, 148701 (2013), DOI: 10.1103/PhysRevLett.110.148701, arXiv: 1212.4794.

[reichard-statistical-2006] Joerg Reichardt and Stefan Bornholdt, "Statistical Mechanics of Community Detection", Phys. Rev. E 74 016110 (2006), DOI: 10.1103/PhysRevE.74.016110, arXiv: cond-mat/0603718

[newman-modularity-2006] M. E. J. Newman, "Modularity and community structure in networks", Proc. Natl. Acad. Sci. USA 103, 8577-8582 (2006), DOI: 10.1073/pnas.0601602103, arXiv: physics/0602124

[newman-modularity-2006] M. E. J. Newman, "Modularity and community structure in networks", Proc. Natl. Acad. Sci. USA 103, 8577-8582 (2006), DOI: 10.1073/pnas.0601602103, arXiv: physics/0602124

[newman-mixing-2003] M. E. J. Newman, "Mixing patterns in networks", Phys. Rev. E 67, 026126 (2003), DOI: 10.1103/PhysRevE.67.026126

[newman-mixing-2003] M. E. J. Newman, "Mixing patterns in networks", Phys. Rev. E 67, 026126 (2003), DOI: 10.1103/PhysRevE.67.026126

[hu-multilevel-2005] Yifan Hu, "Efficient and High Quality Force-Directed Graph", Mathematica Journal, vol. 10, Issue 1, pp. 37-71, (2005) http://www.mathematica-journal.com/issue/v10i1/graph_draw.html

[fruchterman-reingold] Fruchterman, Thomas M. J.; Reingold, Edward M. "Graph Drawing by Force-Directed Placement". Software - Practice & Experience (Wiley) 21 (11): 1129-1164. (1991) DOI: 10.1002/spe.4380211102

[geipel-self-organization-2007] Markus M. Geipel, "Self-Organization applied to Dynamic Network Layout", International Journal of Modern Physics C vol. 18, no. 10 (2007), pp. 1537-1549, DOI: 10.1142/S0129183107011558, arXiv: 0704.1748v5

[graphviz] http://www.graphviz.org

[boost-edmonds-karp] http://www.boost.org/libs/graph/doc/edmonds_karp_max_flow.html

[edmonds-theoretical-1972] Jack Edmonds and Richard M. Karp, "Theoretical improvements in the algorithmic efficiency for network flow problems. Journal of the ACM", 19:248-264, 1972 DOI: 10.1145/321694.321699

[ravindra-network-1993] Ravindra K. Ahuja and Thomas L. Magnanti and James B. Orlin,"Network Flows: Theory, Algorithms, and Applications". Prentice Hall, 1993.

[boost-push-relabel] http://www.boost.org/libs/graph/doc/push_relabel_max_flow.html

[goldberg-new-1985] A. V. Goldberg, "A New Max-Flow Algorithm", MIT Tehnical report MIT/LCS/TM-291, 1985.

[boost-kolmogorov] http://www.boost.org/libs/graph/doc/boykov_kolmogorov_max_flow.html

[kolmogorov-graph-2003] Vladimir Kolmogorov, "Graph Based Algorithms for Scene Reconstruction from Two or More Views", PhD thesis, Cornell University, September 2003.

[boykov-experimental-2004] Yuri Boykov and Vladimir Kolmogorov, "An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 26, no. 9, pp. 1124-1137, Sept. 2004. DOI: 10.1109/TPAMI.2004.60

[max-flow-min-cut] http://en.wikipedia.org/wiki/Max-flow_min-cut_theorem

[stoer$_simple_1$997] Stoer, Mechthild and Frank Wagner, "A simple min-cut algorithm". Journal of the ACM 44 (4), 585-591, 1997. DOI: 10.1145/263867.263872

[metropolis-equations-1953] Metropolis, N.; Rosenbluth, A.W.; Rosenbluth, M.N.; Teller, A.H.; Teller, E. "Equations of State Calculations by Fast Computing Machines". Journal of Chemical Physics 21 (6): 1087-1092 (1953). DOI: 10.1063/1.1699114

[hastings-monte-carlo-1970] Hastings, W.K. "Monte Carlo Sampling Methods Using Markov Chains and Their Applications". Biometrika 57 (1): 97-109 (1970). DOI: 10.1093/biomet/57.1.97

[holland-stochastic-1983] Paul W. Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt, "Stochastic blockmodels: First steps," Social Networks 5, no. 2: 109-13 (1983) DOI: 10.1016/0378-8733(83)90021-7

[karrer-stochastic-2011] Brian Karrer and M. E. J. Newman, "Stochastic blockmodels and community structure in networks," Physical Review E 83, no. 1: 016107 (2011) DOI: 10.1103/PhysRevE.83.016107 arXiv: 1008.3926

[metropolis-equations-1953] Metropolis, N.; Rosenbluth, A.W.; Rosenbluth, M.N.; Teller, A.H.; Teller, E. "Equations of State Calculations by Fast Computing Machines". Journal of Chemical Physics 21 (6): 1087-1092 (1953). DOI: 10.1063/1.1699114

[hastings-monte-carlo-1970] Hastings, W.K. "Monte Carlo Sampling Methods Using Markov Chains and Their Applications". Biometrika 57 (1): 97-109 (1970). DOI: 10.1093/biomet/57.1.97

[holland-stochastic-1983] Paul W. Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt, "Stochastic blockmodels: First steps," Social Networks 5, no. 2: 109-13 (1983) DOI: 10.1016/0378-8733(83)90021-7

[karrer-stochastic-2011] Brian Karrer and M. E. J. Newman, "Stochastic blockmodels and community structure in networks," Physical Review E 83, no. 1: 016107 (2011) DOI: 10.1103/PhysRevE.83.016107 arXiv: 1008.3926

[line-wiki] http://en.wikipedia.org/wiki/Line_graph

[cgal-triang] http://www.cgal.org/Manual/last/doc_html/cgal_manual/Triangulation_3/Chapter_main.htm

[lattice] http://en.wikipedia.org/wiki/Square_lattice

[complete] http://en.wikipedia.org/wiki/Complete_graph

[geometric-graph] Jesper Dall and Michael Christensen, "Random geometric graphs", Phys. Rev. E 66, 016121 (2002), DOI: 10.1103/PhysRevE.66.016121

[yule] Yule, G. U. "A Mathematical Theory of Evolution, based on the Conclusions of Dr. J. C. Willis, F.R.S.". Philosophical Transactions of the Royal Society of London, Ser. B 213: 21-87, 1925, DOI: 10.1098/rstb.1925.0002

[price] Derek De Solla Price, "A general theory of bibliometric and other cumulative advantage processes", Journal of the American Society for Information Science, Volume 27, Issue 5, pages 292-306, September 1976, DOI: 10.1002/asi.4630270505

[barabasi-albert] Barabási, A.-L., and Albert, R., "Emergence of scaling in random networks", Science, 286, 509, 1999, DOI: 10.1126/science.286.5439.509

[dorogovtsev-evolution] S. N. Dorogovtsev and J. F. F. Mendes, "Evolution of networks", Advances in Physics, 2002, Vol. 51, No. 4, 1079-1187, DOI: 10.1080/00018730110112519

[Boost] http://www.boost.org/libs/graph/doc/table_of_contents.html

[Weave] http://www.scipy.org/Weave

[bfs] Edward Moore, "The shortest path through a maze", International Symposium on the Theory of Switching, 1959

[bfs-bgl] http://www.boost.org/doc/libs/release/libs/graph/doc/breadth_first_search.html

[bfs-wikipedia] http://en.wikipedia.org/wiki/Breadth-first_search

[dfs-bgl] http://www.boost.org/doc/libs/release/libs/graph/doc/depth_first_search.html

[dfs-wikipedia] http://en.wikipedia.org/wiki/Depth-first_search

[dijkstra] E. Dijkstra, "A note on two problems in connexion with graphs", Numerische Mathematik, 1:269-271, 1959.

[dijkstra-bgl] http://www.boost.org/doc/libs/release/libs/graph/doc/dijkstra_shortest_paths_no_color_ma

[dijkstra-wikipedia] http://en.wikipedia.org/wiki/Dijkstra's_algorithm

[bellman-ford] R. Bellman, "On a routing problem", Quarterly of Applied Mathematics, 16(1):87-90, 1958.

[bellman-ford-bgl] http://www.boost.org/doc/libs/release/libs/graph/doc/bellman_ford_shortest.html

[bellman-ford-wikipedia] http://en.wikipedia.org/wiki/Bellman-Ford_algorithm

[astar] Hart, P. E.; Nilsson, N. J.; Raphael, B. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". IEEE Transactions on Systems Science and Cybernetics SSC4 4 (2): 100-107, 1968. DOI: 10.1109/TSSC.1968.300136

[astar-bgl] http://www.boost.org/doc/libs/release/libs/graph/doc/astar_search.html

[astar-wikipedia] http://en.wikipedia.org/wiki/A*_search_algorithm

[wikipedia-adjacency] http://en.wikipedia.org/wiki/Adjacency_matrix

[wikipedia-laplacian] http://en.wikipedia.org/wiki/Laplacian_matrix

[wikipedia-incidence] http://en.wikipedia.org/wiki/Incidence_matrix

[ullmann-algorithm-1976] Ullmann, J. R., "An algorithm for subgraph isomorphism", Journal of the ACM 23 (1): 31-42, 1976, DOI: 10.1145/321921.321925

[subgraph-isormophism-wikipedia] http://en.wikipedia.org/wiki/Subgraph_isomorphism_problem

[kruskal-shortest-1956] J. B. Kruskal. "On the shortest spanning subtree of a graph and the traveling salesman problem", In Proceedings of the American Mathematical Society, volume 7, pages 48-50, 1956. DOI: 10.1090/S0002-9939-1956-0078686-7

[prim-shortest-1957] R. Prim. "Shortest connection networks and some generalizations", Bell System Technical Journal, 36:1389-1401, 1957.

[boost-mst] http://www.boost.org/libs/graph/doc/graph_theory_review.html#sec:minimum-spanning-tree

[mst-wiki] http://en.wikipedia.org/wiki/Minimum_spanning_tree

[wilson-generating-1996] David Bruce Wilson, "Generating random spanning trees more quickly than the cover time", Proceedings of the twenty-eighth annual ACM symposium on Theory of computing, Pages 296-303, ACM New York, 1996, DOI: 10.1145/237814.237880

[boost-rst] http://www.boost.org/libs/graph/doc/random_spanning_tree.html

[dominator-bgl] http://www.boost.org/libs/graph/doc/lengauer_tarjan_dominator.htm

[topological-boost] http://www.boost.org/libs/graph/doc/topological_sort.html

[topological-wiki] http://en.wikipedia.org/wiki/Topological_sorting

[transitive-boost] http://www.boost.org/libs/graph/doc/transitive_closure.html

[transitive-wiki] http://en.wikipedia.org/wiki/Transitive_closure

[k-core] http://en.wikipedia.org/wiki/Degeneracy_%28graph_theory%29

[batagelk-algorithm] V. Batagelj, M. Zaversnik, "An O(m) Algorithm for Cores Decomposition of Networks", 2003, arXiv: cs/0310049

[bfs] Edward Moore, "The shortest path through a maze", International Symposium on the Theory of Switching (1959), Harvard University Press;

[bfs-boost] http://www.boost.org/libs/graph/doc/breadth_first_search.html

[dijkstra] E. Dijkstra, "A note on two problems in connexion with graphs." Numerische Mathematik, 1:269-271, 1959.

[dijkstra-boost] http://www.boost.org/libs/graph/doc/dijkstra_shortest_paths.html

[johnson-apsp] http://www.boost.org/libs/graph/doc/johnson_all_pairs_shortest.html

[floyd-warshall-apsp] http://www.boost.org/libs/graph/doc/floyd_warshall_shortest.html

[bfs] Edward Moore, "The shortest path through a maze", International Symposium on the Theory of Switching (1959), Harvard University Press

[bfs-boost] http://www.boost.org/libs/graph/doc/breadth_first_search.html

[dijkstra] E. Dijkstra, "A note on two problems in connexion with graphs." Numerische Mathematik, 1:269-271, 1959.

[dijkstra-boost] http://www.boost.org/libs/graph/doc/dijkstra_shortest_paths.html

[pseudo-diameter] http://en.wikipedia.org/wiki/Distance_%28graph_theory%29

[boost-bipartite] http://www.boost.org/libs/graph/doc/is_bipartite.html

[boyer-myrvold] John M. Boyer and Wendy J. Myrvold, "On the Cutting Edge: Simplified O(n) Planarity by Edge Addition" Journal of Graph Algorithms and Applications, 8(2): 241-273, 2004. http://www.emis.ams.org/journals/JGAA/accepted/2004/BoyerMyrvold2004.8.3.pdf

[boost-planarity] http://www.boost.org/libs/graph/doc/boyer_myrvold.html

[boost-planarity] http://www.boost.org/libs/graph/doc/make_maximal_planar.html

[DAG-wiki] http://en.wikipedia.org/wiki/Directed_acyclic_graph

[boost-max-matching] http://www.boost.org/libs/graph/doc/maximum_matching.html

[matching-heuristic] B. Hendrickson and R. Leland. "A Multilevel Algorithm for Partitioning Graphs." In S. Karin, editor, Proc. Supercomputing '95, San Diego. ACM Press, New York, 1995, DOI: 10.1145/224170.224228

[mivs-wikipedia] http://en.wikipedia.org/wiki/Independent_set_%28graph_theory%29

[mivs-luby] Luby, M., "A simple parallel algorithm for the maximal independent set problem", Proc. 17th Symposium on Theory of Computing, Association for Computing Machinery, pp. 1-10, (1985) DOI: 10.1145/22145.22146.

[reciprocity] S. Wasserman and K. Faust, "Social Network Analysis". (Cambridge University Press, Cambridge, 1994)

[lopez-reciprocity-2007] Gorka Zamora-López, Vinko Zlatić, Changsong Zhou, Hrvoje Štefančić, and Jürgen Kurths "Reciprocity of networks with degree correlations and arbitrary degree sequences", Phys. Rev. E 77, 016106 (2008) DOI: 10.1103/PhysRevE.77.016106, arXiv: 0706.3372

[tsp-bgl] http://www.boost.org/libs/graph/doc/metric_tsp_approx.html

[tsp] http://en.wikipedia.org/wiki/Travelling_salesman_problem

[sgc-bgl] http://www.boost.org/libs/graph/doc/sequential_vertex_coloring.html

[graph-coloring] http://en.wikipedia.org/wiki/Graph_coloring

PYTHON MODULE INDEX

g

graph_tool, 27
graph_tool.centrality, 39
graph_tool.clustering, 59
graph_tool.collection, 65
graph_tool.community, 68
graph_tool.correlations, 93
graph_tool.draw, 102
graph_tool.flow, 126
graph_tool.generation, 136
graph_tool.run_action, 163
graph_tool.search, 166
graph_tool.spectral, 186
graph_tool.stats, 189
graph_tool.topology, 194
graph_tool.util, 222

PYTHON MODULE INDEX

g

graph_tool, 27
graph_tool.centrality, 39
graph_tool.clustering, 59
graph_tool.collection, 65
graph_tool.community, 68
graph_tool.correlations, 93
graph_tool.draw, 102
graph_tool.flow, 126
graph_tool.generation, 136
graph_tool.run_action, 163
graph_tool.search, 166
graph_tool.spectral, 186
graph_tool.stats, 189
graph_tool.topology, 194
graph_tool.util, 222