igraph Reference Manual

Gábor Csárdi
Department of Statistics, Harvard University

1 Oxford street, Cambridge, MA, 02138 USA

Tamás Nepusz
Department of Biological Physics, Eötvös University

1/a Pázmány Péter sétány, 1117 Budapest, Hungary

igraph Reference Manual

by Gábor Csárdi and Tamás Nepusz

Table of Contents

List of Examples

Chapter 1. Introduction

This is another library for creating and manipulating graphs. You can look at it two ways: first, igraph contains the implementation of quite a lot graph algorithms. These include classic graph algorithms like graph isomorphism, graph girth and connectivity and also the new wave graph algorithms like transitivity, graph motifs and community structure detection. Skim through the table of contents or the index of this book to get an impression.

Second, igraph provides a platform for the developing and/or implementing graph algorithms. It has a quite efficient data structure for representing graphs and a number of other data structures like flexible vectors, stacks, heaps, queues, adjacency lists to accomplish this. In fact these data structures evolved along the implementation of the classic and non-classic graph algorithms which make up the major part of the igraph library. This way they were fine tuned and checked for correctness several times.

Our main goal with developing igraph was to create a graph library which is efficient on large but not extremely large graphs. More precisely, it is assumed that the graph(s) fit into the physical memory of the computer. Nowadays this means graphs with several million vertices and/or edges. Our definition of efficient is that it runs fast, both in theory and (more importantly) in practice.

We believe that one of the big strengths of igraph is that it can be embedded into a higher level language or environment. Two such embeddings (or interfaces if you look at them the other way) are currently being developed by us: igraph as a GNU R package and igraph as a Python extension module. A third embedding, being developed by another developer is a Ruby extension. Other are likely to come. The high level languages as R or Python make it possible to do use graph routines with mush greater comfort, without actually writing a single line of C code. They have some, usually very small, speed penalty compared to the C version, but add ease and much flexibility. This manual however covers only the C library. If you want to use Python or GNU R, please see the documentation written specifically for these interfaces and come back here only if you're interested in some detail which is not covered in those documents.

We still consider igraph as a child project. It has much room for development and we are sure that it will improve a lot in the near future. Any feedback we can get from the users is very important for us, as most of the time these questions and comments guide us in what to add and what to improve.

igraph is open source and distributed under the terms of the GNU GPL. We strongly believe that all the algorithms used in science, let that be graph theory or not, should have an efficient open source implementation allowing use and modification for anyone.

1.1. igraph is free software

igraph library

1.2. Citing igraph

To cite igraph in publications, please use the following reference:

Gábor Csárdi, Tamás Nepusz: The igraph software package for complex network research. InterJournal Complex Systems, 1695, 2006.

Chapter 2. Installation

First download the latest version of the **igraph** C library from SourceForge (https://sourceforge.net/projects/igraph/files/), and uncompress it to a temporary directory:

```
$ tar xzf igraph-0.2.tar.gz
$ cd igraph-0.2
```

To install the complete C library typing

```
$ ./configure
$ make
$ make install
```

(the latter as root) should work on most systems. You can try

```
$ ./configure --help
```

to see installations options, and reading the `INSTALL` file.

Installing the **igraph** R package is very simple, you don't need to download anything by hand, just give the command

```
> install.packages("igraph", lib="~/.R/library")
```

in R and select a mirror site close to you. The *`lib`* argument specifies the directory to which the package will be installed. If not specified, this will be default system wide R package directory. You must have write permissions for this directory.

Also, consult your R documentation.

Installing the **igraph** Python package is a little bit more difficult, since chances are that you have to compile it for yourself (as long as there is no compile farm at the Python Package Index and we can't compile it ourselves to all platforms). First, check if there is a compiled version available for your system at igraph's Python Package Index page (http://www.python.org/pypi/igraph). If there is, just use that. (Python eggs should be put anywhere in your Python library path, executable installers should be executed of course). If there isn't, you'll have to compile it by hand. So, first install a recent C compiler.

We usually compile **igraph** with the GNU C compiler (http://gcc.gnu.org). If you are a Windows user, you can find it as part of the Cygwin (http://www.cygwin.com) environment or in the MinGW+MSYS (http://www.mingw.org) project. You can also try Microsoft's free C compiler suite (or even worse, Visual Studio), but there are known issues with the compilation of **igraph** in MSVC, and you'll have to resolve them yourself by tweaking the source code. (Patches are welcome! :)). If you use Linux, **gcc** is usually included in your default system, but even if it isn't, there will be a package from which you can install it. (In Debian and Ubuntu Linux, you'll have to install the package called **build-essential**). If you have a Mac, **gcc** is part of the Xcode developer suite, which is usually included in your OS X install DVD, or can be obtained freely from the Apple Developer Connection (http://developer.apple.com/tools/downloads/) website.

After having obtained a C compiler, you'll have to install an XML processing library called **libxml2**. Windows users should get it from this website (http://xmlsoft.org) (there are binary versions, no need to compile anything), Linux users should be able to find a package again in their respective distribution (Debian and Ubuntu users: install the **libxml2** and **libxml2-dev** libraries). Mac users should not do anything, since **libxml2** is part of the default system installation. However, you should check it anyway, launch a terminal and type the following command:

```
$ xml2-config
```

If you don't receive any error message, you can go on to the next step.

Now, get the **igraph** source from the Python Package Index (http://www.python.org/pypi/igraph), extract it to a directory and start the compilation. On Windows, launch the Cygwin or MinGW environment and type:

```
$ python setup.py build --compiler=cygwin
```

(Instead of **--compiler=cygwin**, **--compiler=mingw32** should also work, then you have no POSIX emulation available (you don't really need it for **igraph** yet), but you also won't need `cygwin1.dll`). If the shell keeps on complaining that it does not find the Python interpreter, use its full path. For instance, if you have Python installed in `C:\Devel\Python24`, use the following command:

```
$ /cygdrive/c/devel/python24/python setup.py build --compiler=cygwin
```

If the compilation finished without errors, you can install the library:

```
$ python setup.py install
```

(Use the full path again if necessary).

Linux and Mac users should succeed with the following commands issued from the **igraph** root directory:

```
$ python setup.py build
$ python setup.py install
```

Note that you'll need write permissions to the Python library path, so usually you must have root permissions to issue the second command. If you want to install it to a different directory, just copy everything from the `build/lib.*` subdirectory to wherever you want.

Chapter 3. Tutorial

3.1. Lesson 1. Compiling programs using igraph.

The following short example program demonstrates the basic usage of the **igraph** library.

```
#include <igraph.h>

int main(void)
{
     igraph_integer_t diameter;
     igraph_t graph;
     igraph_erdos_renyi_game(&graph, IGRAPH_ERDOS_RENYI_GNP, 1000, 5.0/1000,
                             IGRAPH_UNDIRECTED, IGRAPH_NO_LOOPS);
     igraph_diameter(&graph, &diameter, 0, 0, 0, IGRAPH_UNDIRECTED, 1);
     printf("Diameter of a random graph with average degree 5: %d\n",
             (int) diameter);
     igraph_destroy(&graph);
     return 0;
}
```

This example illustrates a couple of points. First, programs using the **igraph** library should include the `igraph.h` header file. Second, **igraph** uses the igraph_real_t type for real numbers instead of double. Third, **igraph** graph objects are represented by the igraph_t data type. Fourth, the `igraph_erdos_renyi_game()` creates a graph and `igraph_destroy()` destroys it, ie. deallocates the memory associated to it.

For compiling this program you need a C compiler, if this is called **gcc** and the previous code is saved in file `igraph_test.c`, you will need a command like this:

```
gcc igraph_test.c -I/usr/local/igraph -L/usr/local/lib -ligraph -o igraph_test
```

The exact form depends on where **igraph** was installed on your system. The directory after the `-I` switch is the one containing the `igraph.h` file, while the one following `-L` should contain the library file itself, usually a file called `libigraph.so`, `libigraph.a` or `igraph.dll`. It your system has the **pkg-config** utility you are likely to get the neccessary compile options by issuing the command

```
pkg-config --libs --cflags igraph
```

The executable can be run by simply typing its name like this:

```
./igraph_test
```

on most systems. If you use dynamic linking and the **igraph** libraries are not at a standard place, you may need to set the LD_LIBRARY_PATH variable, the syntax depends on the shell use are using. In **bash** it goes like this:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/user/libs/igraph
./igraph_test
```

Here we assumed that the **igraph** library is installed in `/home/user/libs/igraph`. Alternatively, you can use the LD_PRELOAD variable to preload the **igraph** library before invoking your program:

```
LD_PRELOAD=/home/user/libs/igraph/libigraph.so ./igraph_test
```

Please note that LD_PRELOAD and LD_LIBRARY_PATH are usually available only on Un*x-like systems. On Windows using Cygwin it is usually enough to set the PATH enviroment variable to include the folder in which the **igraph** library is installed, look for the `cygigraph-0.dll` or similar file.

3.2. Lesson 2. Creating your first graphs.

The functions generating graph objects are called graph generators. Stochastic (=randomized) graph generators are called "games".

igraph can handle directed and undirected graphs. Most graph generators are able to create both types of graphs and most other functions are usually also capable of handling both. Eg. `igraph_shortest_paths()` which (surprisingly) calculates shortest paths from a vertex to another vertices can calculate directed or undirected paths.

igraph has sophisticated ways for creating graphs. The simplest graphs are deterministic regular structures like star graphs (`igraph_star()`), ring graphs (`igraph_ring()`), lattices (`igraph_lattice()`) or trees (`igraph_tree()`).

The following example creates an undirected regular circular lattice, adds some random edges to it and calculates the average length of shortest paths between all pairs of vertices in the graph before and after adding the random edges. (The message is that some random edges can reduce path lengths a lot.)

```
#include <igraph.h>

int main(void) {
  igraph_real_t avg_path;
  igraph_t graph;
  igraph_vector_t dimvector;
  igraph_vector_t edges;
  int i;

  igraph_vector_init(&dimvector, 2);
  VECTOR(dimvector)[0]=30;
```

```
  VECTOR(dimvector)[1]=30;
  igraph_lattice(&graph, &dimvector, 0, IGRAPH_UNDIRECTED, 0, 1);

  srand(100);
  igraph_vector_init(&edges, 20);
  for (i=0; i<igraph_vector_size(&edges); i++) {
    VECTOR(edges)[i] = rand() % (int)igraph_vcount(&graph);
  }

  igraph_average_path_length(&graph, &avg_path, IGRAPH_UNDIRECTED, 1);
  printf("Average path length (lattice):            %f\n", (double) avg_path);

  igraph_add_edges(&graph, &edges, 0);
  igraph_average_path_length(&graph, &avg_path, IGRAPH_UNDIRECTED, 1);
  printf("Average path length (randomized lattice): %f\n", (double) avg_path);

  igraph_vector_destroy(&dimvector);
  igraph_vector_destroy(&edges);
  igraph_destroy(&graph);

  return 0;
}
```

This example illustrates some new points. **igraph** uses igraph_vector_t instead of plain C arrays. igraph_vector_t is superior to regular arrays in almost every sense. Vectors are created by the `igraph_vector_init()` function and like graphs they should be destroyed if not needed any more by calling `igraph_vector_destroy()` on them. A vector can be indexed by the `VECTOR()` function (right now it is a macro). Vectors can be resized, eg. most **igraph** functions returning the result in a vector resize it to the size of the result.

`igraph_lattice()` takes a vector argument specifying the dimensions of the lattice, in this example we generate a 30x30 two dimensional lattice. See the documentation of `igraph_lattice()` in the reference manual for the other arguments.

The vertices in a graph are identified by an integer number between 0 and N-1, N is the number of vertices in the graph (this can be obtained by `igraph_vcount()`, as in the example).

The `igraph_add_edges()` function simply takes a graph and a vector of vertex ids defining the new edges. The first edge is between the first two vertex ids in the vector, the second edge is between the second two, etc. This way we add ten random edges to the lattice.

Note that in the example it is possible to add loop edges, edges pointing to the same vertex and multiple edges, more than one edge between the same pair of vertices. igraph_t can of course represent loops and multiple edges, although some routines expect simple graphs, ie. graphs without loop and multiple edges, because for example some structural properties are ill-defined for non-simple graphs. Loop edges can be removed by calling `igraph_simplify()`.

3.3. Lesson 3. Calculating various properties of graphs.

In our next example we will calculate various centrality measures in a friendship graph. The friendship graph is from the famous Zachary karate club study. (Web search on 'Zachary karate' if you want to know more about this.) Centrality measures quantify how central is the position of individual vertices in the graph.

```
#include <igraph.h>

int main(void) {
     igraph_t graph;
     igraph_vector_t v;
     igraph_vector_t result;
     igraph_real_t edges[] = { 0, 1, 0, 2, 0, 3, 0, 4, 0, 5, 0, 6, 0, 7, 0, 8,
                               0,10, 0,11, 0,12, 0,13, 0,17, 0,19, 0,21, 0,31,
                               1, 2, 1, 3, 1, 7, 1,13, 1,17, 1,19, 1,21, 1,30,
                               2, 3, 2, 7, 2,27, 2,28, 2,32, 2, 9, 2, 8, 2,13,
                               3, 7, 3,12, 3,13, 4, 6, 4,10, 5, 6, 5,10, 5,16,
                               6,16, 8,30, 8,32, 8,33, 9,33,13,33,14,32,14,33,
                              15,32,15,33,18,32,18,33,19,33,20,32,20,33,
                              22,32,22,33,23,25,23,27,23,32,23,33,23,29,
                              24,25,24,27,24,31,25,31,26,29,26,33,27,33,
                              28,31,28,33,29,32,29,33,30,32,30,33,31,32,31,33,
                              32,33
     };

     igraph_vector_view(&v, edges, sizeof(edges)/sizeof(double));
     igraph_create(&graph, &v, 0, IGRAPH_UNDIRECTED);

     igraph_vector_init(&result, 0);

     igraph_degree(&graph, &result, igraph_vss_all(), IGRAPH_ALL,
                   IGRAPH_LOOPS);
     printf("Maximum degree is      %10i, vertex %2i.\n",
            (int)igraph_vector_max(&result), (int)igraph_vector_which_max(&result));

     igraph_closeness(&graph, &result, igraph_vss_all(), IGRAPH_ALL);
     printf("Maximum closeness is   %10f, vertex %2i.\n",
             (double)igraph_vector_max(&result), (int)igraph_vector_which_max(&result));

     igraph_betweenness(&graph, &result, igraph_vss_all(),
                        IGRAPH_UNDIRECTED);
     printf("Maximum betweenness is %10f, vertex %2i.\n",
             (double)igraph_vector_max(&result), (int)igraph_vector_which_max(&result));

     igraph_vector_destroy(&result);
     igraph_destroy(&graph);
```

```
    return 0;
}
```

This example reflects some new features. First of all, it shows a way to define a graph simply as defining a C array with its edges. Function `igraph_vector_view()` creates a *view* of a C array. It does not copy any data, this also means that you should not call `igraph_vector_destroy()` on a vector created this way. This vector is then used to create the undirected graph.

Then the degree, closeness and betweenness centrality of the vertices is calculated and the highest values are printed. Note that the vector (`result`) which returns the result from these functions has to be initialized first, and also that the functions resize it to be able to hold the result.

The `igraph_vss_all()` argument tells the functions to calculate the property for every vertex in the graph, it is shorthand for a *vertex selector* (igraph_vs_t). Vertex selectors help performing operations on a subset of vertices, you can read more about them in one of the following chapters.

Chapter 4. About igraph graphs, the basic interface

4.1. The igraph data model

The igraph library can handle directed and undirected graphs. The igraph graphs are multisets of ordered (if directed) or unordered (if undirected) labeled pairs. The labels of the pairs plus the number of vertices always starts with zero and ends with the number of edges minus one. In addition to that a table of metadata is also attached to every graph, its most important entries are the number of vertices in the graph and whether the graph is directed or undirected.

Like the edges, the igraph vertices are also labeled by number between zero and the number of vertices minus one. So, to summarize, a directed graph can be imagined like this:

```
( vertices: 6,
  directed: yes,
  {
   (0,2),
   (2,2),
   (2,3),
   (3,3),
   (3,4),
   (3,4),
   (4,1)
  }
)
```

Here the edges are ordered pairs or vertex ids, and the graph is a multiset of edges plus some meta-data.

An undirected graph is like this:

```
( vertices: 6,
  directed: no,
  {
   {0,2},
   {2},
   {2,3},
   {3},
   {3,4},
   {3,4},
   {4,1}
  }
)
```

Here an edge is a set of one or two vertex ids, two for most of the time, except for loop edges. A graph is a multiset of edges plus meta data, just like in the directed case.

It is possible to convert a directed graph to an undirected one, see the `igraph_to_directed()` and `igraph_to_undirected()` functions.

Note that igraph has some limited support for graphs with multiple edges. The support means that multiple edges can be stored in igraph graphs, but for most functions (like `igraph_betweenness()`) it is not checked that they work well on graphs with multiple edges. To eliminate multiple edges from a graph, you can use `igraph_simplify()`.

4.2. The basic interface

This is the very minimal API in **igraph**. All the other functions use this minimal set for creating and manipulating graphs.

This is a very important principle since it makes possible to implement other data representations by implementing only this minimal set.

4.2.1. Graph Constructors and Destructors

4.2.1.1. `igraph_empty` — Creates an empty graph with some vertices and no edges.

```
int igraph_empty(igraph_t *graph, igraph_integer_t n, igraph_bool_t directed);
```

The most basic constructor, all the other constructors should call this to create a minimal graph object. Our use of the term "empty graph" in the above description should be distinguished from the mathematical definition of the empty or null graph. Strictly speaking, the empty or null graph in graph theory is the graph with no vertices and no edges. However by "empty graph" as used in `igraph` we mean a graph having zero or more vertices, but no edges.

Arguments:

graph:

Pointer to a not-yet initialized graph object.

n:

The number of vertices in the graph, a non-negative integer number is expected.

directed:

Boolean; whether the graph is directed or not. Supported values are:

`IGRAPH_DIRECTED`

The graph will be *directed.*

`IGRAPH_UNDIRECTED`

The graph will be *undirected.*

Returns:

Error code: `IGRAPH_EINVAL`: invalid number of vertices.

Time complexity: O(|V|) for a graph with |V| vertices (and no edges).

Example 4-1. File examples/simple/igraph_empty.c

4.2.1.2. `igraph_empty_attrs` — Creates an empty graph with some vertices, no edges and some graph attributes.

```
int igraph_empty_attrs(igraph_t *graph, igraph_integer_t n, igraph_bool_t directed, void* a
```

Use this instead of `igraph_empty()` if you wish to add some graph attributes right after initialization. This function is currently not very interesting for the ordinary user. Just supply 0 here or use `igraph_empty()`.

Arguments:

graph:

Pointer to a not-yet initialized graph object.

n:

The number of vertices in the graph; a non-negative integer number is expected.

directed:

Boolean; whether the graph is directed or not. Supported values are:

`IGRAPH_DIRECTED`

Create a *directed* graph.

`IGRAPH_UNDIRECTED`

Create an *undirected* graph.

attr:

The attributes.

Returns:

Error code: `IGRAPH_EINVAL`: invalid number of vertices.

Time complexity: O(|V|) for a graph with |V| vertices (and no edges).

4.2.1.3. `igraph_copy` — Creates an exact (deep) copy of a graph.

```
int igraph_copy(igraph_t *to, const igraph_t *from);
```

This function deeply copies a graph object to create an exact replica of it. The new replica should be destroyed by calling `igraph_destroy()` on it when not needed any more.

You can also create a shallow copy of a graph by simply using the standard assignment operator, but be careful and do *not* destroy a shallow replica. To avoid this mistake, creating shallow copies is not recommended.

Arguments:

to:

Pointer to an uninitialized graph object.

from:

Pointer to the graph object to copy.

Returns:

Error code.

Time complexity: O(|V|+|E|) for a graph with |V| vertices and |E| edges.

Example 4-2. File examples/simple/igraph_copy.c

4.2.1.4. `igraph_destroy` — Frees the memory allocated for a graph object.

```
int igraph_destroy(igraph_t *graph);
```

This function should be called for every graph object exactly once.

This function invalidates all iterators (of course), but the iterators of a graph should be destroyed before the graph itself anyway.

Arguments:

graph:

Pointer to the graph to free.

Returns:

Error code.

Time complexity: operating system specific.

4.2.2. Basic Query Operations

4.2.2.1. `igraph_vcount` — The number of vertices in a graph.

```
igraph_integer_t igraph_vcount(const igraph_t *graph);
```

Arguments:

graph:

The graph.

Returns:

Number of vertices.

Time complexity: O(1)

4.2.2.2. `igraph_ecount` — The number of edges in a graph.

```
igraph_integer_t igraph_ecount(const igraph_t *graph);
```

Arguments:

graph:

The graph.

Returns:

Number of edges.

Time complexity: O(1)

4.2.2.3. `igraph_edge` — Gives the head and tail vertices of an edge.

```
int igraph_edge(const igraph_t *graph, igraph_integer_t eid,
  igraph_integer_t *from, igraph_integer_t *to);
```

Arguments:

graph:

The graph object.

eid:

The edge id.

from:

Pointer to an igraph_integer_t. The tail of the edge will be placed here.

to:

Pointer to an igraph_integer_t. The head of the edge will be placed here.

Returns:

Error code. The current implementation always returns with success.

See also:

`igraph_get_eid()` for the opposite operation.

Added in version 0.2.

Time complexity: O(1).

4.2.2.4. `igraph_get_eid` — Get the edge id from the end points of an edge.

```
int igraph_get_eid(const igraph_t *graph, igraph_integer_t *eid,
     igraph_integer_t pfrom, igraph_integer_t pto,
     igraph_bool_t directed, igraph_bool_t error);
```

For undirected graphs `pfrom` and `pto` are exchangeable.

Arguments:

`graph`:

The graph object.

`eid`:

Pointer to an integer, the edge id will be stored here.

`pfrom`:

The starting point of the edge.

`pto`:

The end point of the edge.

`directed`:

Logical constant, whether to search for directed edges in a directed graph. Ignored for undirected graphs.

`error`:

Logical scalar, whether to report an error if the edge was not found. If it is false, then -1 will be assigned to *`eid`*.

Returns:

Error code.

See also:

`igraph_edge()` for the opposite operation.

Time complexity: O(log (d)), where d is smaller of the out-degree of `pfrom` and in-degree of `pto` if *`directed`* is true. If *`directed`* is false, then it is O(log(d)+log(d2)), where d is the same as before and d2 is the minimum of the out-degree of `pto` and the in-degree of `pfrom`.

Example 4-3. File examples/simple/igraph_get_eid.c

Added in version 0.2.

4.2.2.5. `igraph_get_eids` — Return edge ids based on the adjacent vertices.

```
int igraph_get_eids(const igraph_t *graph, igraph_vector_t *eids,
      const igraph_vector_t *pairs,
      const igraph_vector_t *path,
      igraph_bool_t directed, igraph_bool_t error);
```

This function operates in two modes. If the `pairs` argument is not a null pointer, but the `path` argument is, then it searches for the edge ids of all pairs of vertices given in `pairs`. The pairs of vertex ids are taken consecutively from the vector, i.e. `VECTOR(pairs)[0]` and `VECTOR(pairs)[1]` give the first pair, `VECTOR(pairs)[2]` and `VECTOR(pairs)[3]` the second pair, etc.

If the `pairs` argument is a null pointer, and `path` is not a null pointer, then the `path` is interpreted as a path given by vertex ids and the edges along the path are returned.

If neither `pairs` nor `path` are null pointers, then both are considered (first `pairs` and then `path`), and the results are concatenated.

If the `error` argument is true, then it is an error to give pairs of vertices that are not connected. Otherwise -1 is reported for not connected vertices.

If there are multiple edges in the graph, then these are ignored; i.e. for a given pair of vertex ids, always the same edge id is returned, even if the pair is given multiple time in `pairs` or in `path`. See `igraph_get_eids_multi()` for a similar function that works differently in case of multiple edges.

Arguments:

`graph`:

The input graph.

`eids`:

Pointer to an initialized vector, the result is stored here. It will be resized as needed.

`pairs`:

Vector giving pairs of vertices, or a null pointer.

path:

Vector giving vertex ids along a path, or a null pointer.

directed:

Logical scalar, whether to consider edge directions in directed graphs. This is ignored for undirected graphs.

error:

Logical scalar, whether it is an error to supply non-connected vertices. If false, then -1 is returned for non-connected pairs.

Returns:

Error code.

Time complexity: O(n log(d)), where n is the number of queried edges and d is the average degree of the vertices.

See also:

`igraph_get_eid()` for a single edge, `igraph_get_eids_multi()` for a version that handles multiple edges better (at a cost).

Example 4-4. File examples/simple/igraph_get_eids.c

4.2.2.6. `igraph_get_eids_multi` — Query edge ids based on their adjacent vertices, handle multiple edges.

```
int igraph_get_eids_multi(const igraph_t *graph, igraph_vector_t *eids,
     const igraph_vector_t *pairs,
     const igraph_vector_t *path,
     igraph_bool_t directed, igraph_bool_t error);
```

This function operates in two modes. If the `pairs` argument is not a null pointer, but the `path` argument is, then it searches for the edge ids of all pairs of vertices given in `pairs`. The pairs of vertex ids are taken consecutively from the vector, i.e. `VECTOR(pairs)[0]` and `VECTOR(pairs)[1]` give the first pair, `VECTOR(pairs)[2]` and `VECTOR(pairs)[3]` the second pair, etc.

If the `pairs` argument is a null pointer, and `path` is not a null pointer, then the `path` is interpreted as a path given by vertex ids and the edges along the path are returned.

If the `error` argument is true, then it is an error to give pairs of vertices that are not connected. Otherwise -1 is returned for not connected vertex pairs.

An error is triggered if both `pairs` and `path` are non-null pointers.

This function handles multiple edges properly, i.e. if the same pair is given multiple times and they are indeed connected by multiple edges, then each time a different edge id is reported.

Arguments:

`graph`:

The input graph.

`eids`:

Pointer to an initialized vector, the result is stored here. It will be resized as needed.

`pairs`:

Vector giving pairs of vertices, or a null pointer.

`path`:

Vector giving vertex ids along a path, or a null pointer.

`directed`:

Logical scalar, whether to consider edge directions in directed graphs. This is ignored for undirected graphs.

`error`:

Logical scalar, whether to report an error if non-connected vertices are specified. If false, then -1 is returned for non-connected vertex pairs.

Returns:

Error code.

Time complexity: O(|E|+n log(d)), where |E| is the number of edges in the graph, n is the number of queried edges and d is the average degree of the vertices.

See also:

`igraph_get_eid()` for a single edge, `igraph_get_eids()` for a faster version that does not handle multiple edges.

4.2.2.7. `igraph_neighbors` — Adjacent vertices to a vertex.

```
int igraph_neighbors(const igraph_t *graph, igraph_vector_t *neis, igraph_integer_t pnode,
        igraph_neimode_t mode);
```

Arguments:

`graph`:

The graph to work on.

`neis`:

This vector will contain the result. The vector should be initialized beforehand and will be resized. Starting from igraph version 0.4 this vector is always sorted, the vertex ids are in increasing order.

`pnode`:

The id of the node for which the adjacent vertices are to be searched.

`mode`:

Defines the way adjacent vertices are searched in directed graphs. It can have the following values: `IGRAPH_OUT`, vertices reachable by an edge from the specified vertex are searched; `IGRAPH_IN`, vertices from which the specified vertex is reachable are searched; `IGRAPH_ALL`, both kinds of vertices are searched. This parameter is ignored for undirected graphs.

Returns:

Error code: `IGRAPH_EINVVID`: invalid vertex id. `IGRAPH_EINVMODE`: invalid mode argument. `IGRAPH_ENOMEM`: not enough memory.

Time complexity: O(d), d is the number of adjacent vertices to the queried vertex.

Example 4-5. File examples/simple/igraph_neighbors.c

4.2.2.8. `igraph_incident` — Gives the incident edges of a vertex.

```
int igraph_incident(const igraph_t *graph, igraph_vector_t *eids,
      igraph_integer_t pnode, igraph_neimode_t mode);
```

Arguments:

`graph`:

The graph object.

`eids`:

An initialized vector_t object. It will be resized to hold the result.

`pnode`:

A vertex id.

`mode`:

Specifies what kind of edges to include for directed graphs. `IGRAPH_OUT` means only outgoing edges, `IGRAPH_IN` only incoming edges, `IGRAPH_ALL` both. This parameter is ignored for undirected graphs.

Returns:

Error code. `IGRAPH_EINVVID`: invalid *`pnode`* argument, `IGRAPH_EINVMODE`: invalid *`mode`* argument.

Added in version 0.2.

Time complexity: O(d), the number of incident edges to *`pnode`*.

4.2.2.9. `igraph_is_directed` — Is this a directed graph?

```
igraph_bool_t igraph_is_directed(const igraph_t *graph);
```

Arguments:

graph:

The graph.

Returns:

Logical value, `TRUE` if the graph is directed, `FALSE` otherwise.

Time complexity: O(1)

Example 4-6. File examples/simple/igraph_is_directed.c

4.2.2.10. `igraph_degree` — The degree of some vertices in a graph.

```
int igraph_degree(const igraph_t *graph, igraph_vector_t *res,
    const igraph_vs_t vids,
    igraph_neimode_t mode, igraph_bool_t loops);
```

This function calculates the in-, out- or total degree of the specified vertices.

Arguments:

graph:

The graph.

res:

Vector, this will contain the result. It should be initialized and will be resized to be the appropriate size.

vids:

Vector, giving the vertex ids of which the degree will be calculated.

mode:

Defines the type of the degree. Valid modes are: `IGRAPH_OUT`, out-degree; `IGRAPH_IN`, in-degree; `IGRAPH_ALL`, total degree (sum of the in- and out-degree). This parameter is ignored for undirected graphs.

loops:

Boolean, gives whether the self-loops should be counted.

Returns:

Error code: `IGRAPH_EINVVID`: invalid vertex id. `IGRAPH_EINVMODE`: invalid mode argument.

Time complexity: O(v) if loops is TRUE, and O(v*d) otherwise. v is the number of vertices for which the degree will be calculated, and d is their (average) degree.

See also:

`igraph_strength()` for the version that takes into account edge weights.

Example 4-7. File examples/simple/igraph_degree.c

4.2.3. Adding and Deleting Vertices and Edges

4.2.3.1. `igraph_add_edge` — Adds a single edge to a graph.

```
int igraph_add_edge(igraph_t *graph, igraph_integer_t from, igraph_integer_t to);
```

For directed graphs the edge points from *`from`* to *`to`*.

Note that if you want to add many edges to a big graph, then it is inefficient to add them one by one, it is better to collect them into a vector and add all of them via a single `igraph_add_edges()` call.

Arguments:

`igraph`:

The graph.

`from`:

The id of the first vertex of the edge.

`to`:

The id of the second vertex of the edge.

Returns:

Error code.

See also:

`igraph_add_edges()` to add many edges, `igraph_delete_edges()` to remove edges and `igraph_add_vertices()` to add vertices.

Time complexity: O(|V|+|E|), the number of edges plus the number of vertices.

4.2.3.2. `igraph_add_edges` — Adds edges to a graph object.

```
int igraph_add_edges(igraph_t *graph, const igraph_vector_t *edges,
       void *attr);
```

The edges are given in a vector, the first two elements define the first edge (the order is `from`, `to` for directed graphs). The vector should contain even number of integer numbers between zero and the number of vertices in the graph minus one (inclusive). If you also want to add new vertices, call igraph_add_vertices() first.

Arguments:

`graph`:

The graph to which the edges will be added.

`edges`:

The edges themselves.

`attr`:

The attributes of the new edges, only used by high level interfaces currently, you can supply 0 here.

Returns:

Error code: `IGRAPH_EINVEVECTOR`: invalid (odd) edges vector length, `IGRAPH_EINVVID`: invalid vertex id in edges vector.

This function invalidates all iterators.

Time complexity: O(|V|+|E|) where |V| is the number of vertices and |E| is the number of edges in the *new,* extended graph.

Example 4-8. File examples/simple/igraph_add_edges.c

4.2.3.3. `igraph_add_vertices` — Adds vertices to a graph.

```
int igraph_add_vertices(igraph_t *graph, igraph_integer_t nv, void *attr);
```

This function invalidates all iterators.

Arguments:

`graph`:

The graph object to extend.

`nv`:

Non-negative integer giving the number of vertices to add.

`attr`:

The attributes of the new vertices, only used by high level interfaces, you can supply 0 here.

Returns:

Error code: `IGRAPH_EINVAL`: invalid number of new vertices.

Time complexity: O(|V|) where |V| is the number of vertices in the *new,* extended graph.

Example 4-9. File examples/simple/igraph_add_vertices.c

4.2.3.4. `igraph_delete_edges` — Removes edges from a graph.

```
int igraph_delete_edges(igraph_t *graph, igraph_es_t edges);
```

The edges to remove are given as an edge selector.

This function cannot remove vertices, they will be kept, even if they lose all their edges.

This function invalidates all iterators.

Arguments:

graph:

The graph to work on.

edges:

The edges to remove.

Returns:

Error code.

Time complexity: O(|V|+|E|) where |V| and |E| are the number of vertices and edges in the *original* graph, respectively.

Example 4-10. File examples/simple/igraph_delete_edges.c

4.2.3.5. `igraph_delete_vertices` — Removes vertices (with all their edges) from the graph.

```
int igraph_delete_vertices(igraph_t *graph, const igraph_vs_t vertices);
```

This function changes the ids of the vertices (except in some very special cases, but these should not be relied on anyway).

This function invalidates all iterators.

Arguments:

graph:

The graph to work on.

vertices:

The ids of the vertices to remove in a vector. The vector may contain the same id more than once.

Returns:

Error code: `IGRAPH_EINVVID`: invalid vertex id.

Time complexity: O(|V|+|E|), |V| and |E| are the number of vertices and edges in the original graph.

Example 4-11. File examples/simple/igraph_delete_vertices.c

4.2.4. Deprecated functions

4.2.4.1. `igraph_adjacent` — Gives the incident edges of a vertex.

```
int igraph_adjacent(const igraph_t *graph, igraph_vector_t *eids,
      igraph_integer_t pnode, igraph_neimode_t mode);
```

This function was superseded by `igraph_incident()` in igraph 0.6. Please use `igraph_incident()` instead of this function.

Added in version 0.2, deprecated in version 0.6.

Chapter 5. Error Handling

5.1. Error handling basics

igraph functions can run into various problems preventing them from normal operation. The user might have supplied invalid arguments, e.g. a non-square matrix when a square-matrix was expected, or the program has run out of memory while some more memory allocation is required, etc.

By default **igraph** aborts the program when it runs into an error. While this behavior might be good enough for smaller programs, it is without doubt avoidable in larger projects. Please read further if your project requires more sophisticated error handling. You can safely skip the rest of this chapter otherwise.

5.2. Error handlers

If **igraph** runs into an error - an invalid argument was supplied to a function, or we've ran out of memory - the control is transferred to the *error handler* function.

The default error handler is `igraph_error_handler_abort` which prints an error message and aborts the program.

The `igraph_set_error_handler()` function can be used to set a new error handler function of type `igraph_error_handler_t`; see the documentation of this type for details.

There are two other predefined error handler functions, `igraph_error_handler_ignore` and `igraph_error_handler_printignore`. These deallocate the temporarily allocated memory (more about this later) and return with the error code. The latter also prints an error message. If you use these error handlers you need to take care about possible errors yourself by checking the return value of (almost) every non-void **igraph** function.

Independently of the error handler installed, all functions in the library do their best to leave their arguments *semantically* unchanged if an error happens. By semantically we mean that the implementation of an object supplied as an argument might change, but its "meaning" in most cases does not. The rare occasions when this rule is violated are documented in this manual.

5.2.1. `igraph_error_handler_t` — Type of error handler

functions.

```
typedef void igraph_error_handler_t (const char * reason, const char * file,
         int line, int igraph_errno);
```

This is the type of the error handler functions.

Arguments:

reason:

Textual description of the error.

file:

The source file in which the error is noticed.

line:

The number of the line in the source file which triggered the error

igraph_errno:

The **igraph** error code.

5.2.2. `igraph_error_handler_abort` — Abort program in case of error.

```
extern igraph_error_handler_t igraph_error_handler_abort;
```

The default error handler, prints an error message and aborts the program.

5.2.3. `igraph_error_handler_ignore` — Ignore errors.

```
extern igraph_error_handler_t igraph_error_handler_ignore;
```

This error handler frees the temporarily allocated memory and returns with the error code.

5.2.4. `igraph_error_handler_printignore` — Print and ignore errors.

```
extern igraph_error_handler_t igraph_error_handler_printignore;
```

Frees temporarily allocated memory, prints an error message to the standard error and returns with the error code.

5.3. Error codes

Every **igraph** function which can fail return a single integer error code. Some functions are very simple and cannot run into any error, these may return other types, or void as well. The error codes are defined by the `igraph_error_type_t` enumeration.

5.3.1. `igraph_error_type_t` — Error code type.

```
typedef enum {
  IGRAPH_SUCCESS       = 0,
  IGRAPH_FAILURE       = 1,
  IGRAPH_ENOMEM        = 2,
  IGRAPH_PARSEERROR    = 3,
  IGRAPH_EINVAL        = 4,
  IGRAPH_EXISTS        = 5,
  IGRAPH_EINVEVECTOR   = 6,
  IGRAPH_EINVVID       = 7,
  IGRAPH_NONSQUARE     = 8,
  IGRAPH_EINVMODE      = 9,
  IGRAPH_EFILE         = 10,
  IGRAPH_UNIMPLEMENTED = 12,
  IGRAPH_INTERRUPTED   = 13,
  IGRAPH_DIVERGED      = 14,
  IGRAPH_ARPACK_PROD      = 15,
  IGRAPH_ARPACK_NPOS      = 16,
  IGRAPH_ARPACK_NEVNPOS   = 17,
  IGRAPH_ARPACK_NCVSMALL  = 18,
  IGRAPH_ARPACK_NONPOSI   = 19,
  IGRAPH_ARPACK_WHICHINV  = 20,
  IGRAPH_ARPACK_BMATINV   = 21,
```

```
  IGRAPH_ARPACK_WORKLSMALL= 22,
  IGRAPH_ARPACK_TRIDERR   = 23,
  IGRAPH_ARPACK_ZEROSTART = 24,
  IGRAPH_ARPACK_MODEINV   = 25,
  IGRAPH_ARPACK_MODEBMAT  = 26,
  IGRAPH_ARPACK_ISHIFT    = 27,
  IGRAPH_ARPACK_NEVBE     = 28,
  IGRAPH_ARPACK_NOFACT    = 29,
  IGRAPH_ARPACK_FAILED    = 30,
  IGRAPH_ARPACK_HOWMNY    = 31,
  IGRAPH_ARPACK_HOWMNYS   = 32,
  IGRAPH_ARPACK_EVDIFF    = 33,
  IGRAPH_ARPACK_SHUR      = 34,
  IGRAPH_ARPACK_LAPACK    = 35,
  IGRAPH_ARPACK_UNKNOWN   = 36,
  IGRAPH_ENEGLOOP         = 37,
  IGRAPH_EINTERNAL        = 38,
  IGRAPH_ARPACK_MAXIT     = 39,
  IGRAPH_ARPACK_NOSHIFT   = 40,
  IGRAPH_ARPACK_REORDER   = 41,
  IGRAPH_EDIVZERO         = 42,
  IGRAPH_GLP_EBOUND       = 43,
  IGRAPH_GLP_EROOT        = 44,
  IGRAPH_GLP_ENOPFS       = 45,
  IGRAPH_GLP_ENODFS       = 46,
  IGRAPH_GLP_EFAIL        = 47,
  IGRAPH_GLP_EMIPGAP      = 48,
  IGRAPH_GLP_ETMLIM       = 49,
  IGRAPH_GLP_ESTOP        = 50,
  IGRAPH_EATTRIBUTES      = 51,
  IGRAPH_EATTRCOMBINE     = 52,
  IGRAPH_ELAPACK          = 53,
  IGRAPH_EDRL             = 54,
  IGRAPH_EOVERFLOW        = 55,
  IGRAPH_EGLP             = 56,
  IGRAPH_CPUTIME          = 57,
  IGRAPH_EUNDERFLOW       = 58
} igraph_error_type_t;
```

These are the possible values returned by **igraph** functions. Note that these are interesting only if you defined an error handler with `igraph_set_error_handler()`. Otherwise the program is aborted and the function causing the error never returns.

Values:

`IGRAPH_SUCCESS`:

The function successfully completed its task.

`IGRAPH_FAILURE`:

Something went wrong. You'll almost never meet this error as normally more specific error codes are used.

`IGRAPH_ENOMEM`:

There wasn't enough memory to allocate on the heap.

`IGRAPH_PARSEERROR`:

A parse error was found in a file.

`IGRAPH_EINVAL`:

A parameter's value is invalid. Eg. negative number was specified as the number of vertices.

`IGRAPH_EXISTS`:

A graph/vertex/edge attribute is already installed with the given name.

`IGRAPH_EINVEVECTOR`:

Invalid vector of vertex ids. A vertex id is either negative or bigger than the number of vertices minus one.

`IGRAPH_EINVVID`:

Invalid vertex id, negative or too big.

`IGRAPH_NONSQUARE`:

A non-square matrix was received while a square matrix was expected.

`IGRAPH_EINVMODE`:

Invalid mode parameter.

`IGRAPH_EFILE`:

A file operation failed. Eg. a file doesn't exist, or the user has no rights to open it.

`IGRAPH_UNIMPLEMENTED`:

Attempted to call an unimplemented or disabled (at compile-time) function.

`IGRAPH_DIVERGED`:

A numeric algorithm failed to converge.

`IGRAPH_ARPACK_PROD`:

Matrix-vector product failed.

`IGRAPH_ARPACK_NPOS`:

N must be positive.

`IGRAPH_ARPACK_NEVNPOS`:

NEV must be positive.

`IGRAPH_ARPACK_NCVSMALL`:

NCV must be bigger.

`IGRAPH_ARPACK_NONPOSI`:

Maximum number of iterations should be positive.

`IGRAPH_ARPACK_WHICHINV`:

Invalid WHICH parameter.

`IGRAPH_ARPACK_BMATINV`:

Invalid BMAT parameter.

`IGRAPH_ARPACK_WORKLSMALL`:

WORKL is too small.

`IGRAPH_ARPACK_TRIDERR`:

LAPACK error in tridiagonal eigenvalue calculation.

`IGRAPH_ARPACK_ZEROSTART`:

Starting vector is zero.

`IGRAPH_ARPACK_MODEINV`:

MODE is invalid.

`IGRAPH_ARPACK_MODEBMAT`:

MODE and BMAT are not compatible.

`IGRAPH_ARPACK_ISHIFT`:

ISHIFT must be 0 or 1.

`IGRAPH_ARPACK_NEVBE`:

NEV and WHICH='BE' are incompatible.

`IGRAPH_ARPACK_NOFACT`:

Could not build an Arnoldi factorization.

`IGRAPH_ARPACK_FAILED`:

No eigenvalues to sufficient accuracy.

`IGRAPH_ARPACK_HOWMNY`:

HOWMNY is invalid.

`IGRAPH_ARPACK_HOWMNYS`:

HOWMNY='S' is not implemented.

`IGRAPH_ARPACK_EVDIFF`:

Different number of converged Ritz values.

`IGRAPH_ARPACK_SHUR`:

Error from calculation of a real Schur form.

`IGRAPH_ARPACK_LAPACK`:

LAPACK (dtrevc) error for calculating eigenvectors.

`IGRAPH_ARPACK_UNKNOWN`:

Unknown ARPACK error.

`IGRAPH_ENEGLOOP`:

Negative loop detected while calculating shortest paths.

`IGRAPH_EINTERNAL`:

Internal error, likely a bug in igraph.

`IGRAPH_EDIVZERO`:

Big integer division by zero.

`IGARPH_GLP_EBOUND`:

GLPK error (GLP_EBOUND).

`IGARPH_GLP_EROOT`:

GLPK error (GLP_EROOT).

`IGARPH_GLP_ENOPFS`:

GLPK error (GLP_ENOPFS).

`IGARPH_GLP_ENODFS`:

GLPK error (GLP_ENODFS).

`IGARPH_GLP_EFAIL`:

GLPK error (GLP_EFAIL).

`IGARPH_GLP_EMIPGAP`:

GLPK error (GLP_EMIPGAP).

`IGARPH_GLP_ETMLIM`:

GLPK error (GLP_ETMLIM).

`IGARPH_GLP_ESTOP`:

GLPK error (GLP_ESTOP).

`IGRAPH_EATTRIBUTES`:

Attribute handler error. The user is not expected to find this; it is signalled if some igraph function is not using the attribute handler interface properly.

`IGRAPH_EATTRCOMBINE`:

Unimplemented attribute combination method for the given attribute type.

`IGRAPH_ELAPACK`:

A LAPACK call resulted an error.

`IGRAPH_EDRL`:

Internal error in the DrL layout generator.

`IGRAPH_EOVERFLOW`:

Integer or double overflow.

`IGRAPH_EGLP`:

Internal GLPK error.

`IGRAPH_EUNDERFLOW`:

Integer or double underflow.

5.3.2. `igraph_strerror` — Textual description of an error.

```
const char* igraph_strerror(const int igraph_errno);
```

This is a simple utility function, it gives a short general textual description for an **igraph** error code.

Arguments:

`igraph_errno`:

The **igraph** error code.

Returns:

pointer to the textual description of the error code.

5.4. Warning messages

Igraph also supports warning messages in addition to error messages. Warning messages typically do not terminate the program, but they are usually crucial to the user.

Igraph warning are handled similarly to errors. There is a separate warning handler function that is called whenever an igraph function triggers a warning. This handler can be set by the `igraph_set_warning_handler()` function. There are two predefined simple warning handlers, `igraph_warning_handler_ignore()` and `igraph_warning_handler_print()`, the latter being the default.

To trigger a warning, igraph functions typically use the `IGRAPH_WARNING()` macro, the `igraph_warning()` function, or if more flexibility is needed, `igraph_warningf()`.

5.4.1. `igraph_warning_handler_t` — Type of igraph warning handler functions

```
typedef igraph_error_handler_t igraph_warning_handler_t;
```

Currently it is defined to have the same type as `igraph_error_handler_t`, although the last (error code) argument is not used.

5.4.2. `igraph_set_warning_handler` — Install a warning handler

```
igraph_warning_handler_t*
igraph_set_warning_handler(igraph_warning_handler_t* new_handler);
```

Install the supplied warning handler function.

Arguments:

new_handler:

The new warning handler function to install. Supply a null pointer here to uninstall the current warning handler, without installing a new one.

Returns:

The current warning handler function.

5.4.3. IGRAPH_WARNING — Trigger a warning.

```
#define IGRAPH_WARNING(reason)
```

This is the usual way of triggering a warning from an igraph function. It calls `igraph_warning()`.

Arguments:

reason:

The warning message.

5.4.4. igraph_warning — Trigger a warning

```
int igraph_warning(const char *reason, const char *file, int line,
     int igraph_errno);
```

Call this function if you want to trigger a warning from within a function that uses igraph.

Arguments:

reason:

Textual description of the warning.

file:

The source file in which the warning was noticed.

line:

The number of line in the source file which triggered the warning.

igraph_errno:

Warnings could have potentially error codes as well, but this is currently not used in igraph.

Returns:

The supplied error code.

5.4.5. `igraph_warningf` — Trigger a warning, more flexible printf-like syntax

```
int igraph_warningf(const char *reason, const char *file, int line,
      int igraph_errno, ...);
```

This function is similar to `igraph_warning()`, but uses a printf-like syntax. It substitutes the additional arguments into the *reason* template string and calls `igraph_warning()`.

Arguments:

reason:

Textual description of the warning, a template string with the same syntax as the standard printf C library function.

file:

The source file in which the warning was noticed.

line:

The number of line in the source file which triggered the warning.

igraph_errno:

Warnings could have potentially error codes as well, but this is currently not used in igraph.

...:

The additional arguments to be substituted into the template string.

Returns:

The supplied error code.

5.4.6. `igraph_warning_handler_ignore` — Ignore all warnings

```
void igraph_warning_handler_ignore (const char *reason, const char *file,
       int line, int igraph_errno);
```

This warning handler function simply ignores all warnings.

Arguments:

reason:

Textual description of the warning.

file:

The source file in which the warning was noticed.

line:

The number of line in the source file which triggered the warning..

igraph_errno:

Warnings could have potentially error codes as well, but this is currently not used in igraph.

5.4.7. `igraph_warning_handler_print` — Print all warning to the standard error

```
void igraph_warning_handler_print (const char *reason, const char *file,
       int line, int igraph_errno);
```

This warning handler function simply prints all warnings to the standard error.

Arguments:

`reason`:

Textual description of the warning.

`file`:

The source file in which the warning was noticed.

`line`:

The number of line in the source file which triggered the warning..

`igraph_errno`:

Warnings could have potentially error codes as well, but this is currently not used in igraph.

5.5. Advanced topics

5.5.1. Writing error handlers

The contents of the rest of this chapter might be useful only for those who want to create an interface to **igraph** from another language. Most readers can safely skip to the next chapter.

You can write and install error handlers simply by defining a function of type `igraph_error_handler_t` and calling `igraph_set_error_handler()`. This feature is useful for interface writers, as **igraph** will have the chance to signal errors the appropriate way, eg. the R interface defines an error handler which calls the `error()` function, as required by R, while the Python interface has an error handler which raises an exception according to the Python way.

If you want to write an error handler, your error handler should call `IGRAPH_FINALLY_FREE()` to deallocate all temporary memory to prevent memory leaks.

5.5.1.1. `igraph_set_error_handler` — Set a new error handler.

```
igraph_error_handler_t*
igraph_set_error_handler(igraph_error_handler_t* new_handler);
```

Installs a new error handler. If called with 0, it installs the default error handler (which is currently `igraph_error_handler_abort`).

Arguments:

new_handler:

The error handler function to install.

Returns:

The old error handler function. This should be saved and restored if *new_handler* is not needed any more.

5.5.2. Error handling internals

If an error happens, the functions in the library call the `IGRAPH_ERROR` macro with a textual description of the error and an **igraph** error code. This macro calls (through the `igraph_error()` function) the installed error handler. Another useful macro is `IGRAPH_CHECK()`. This checks the return value of its argument, which is normally a function call, and calls `IGRAPH_ERROR` if it is not `IGRAPH_SUCCESS`.

5.5.2.1. IGRAPH_ERROR — Trigger an error.

```
#define IGRAPH_ERROR(reason,igraph_errno)
```

igraph functions usually use this macro when they notice an error. It calls `igraph_error()` with the proper parameters and if that returns the macro returns the "calling" function as well, with the error code. If for some (suspicious) reason you want to call the error handler without returning from the current function, call `igraph_error()` directly.

Arguments:

reason:

Textual description of the error. This should be something more descriptive than the text associated with the error code. Eg. if the error code is `IGRAPH_EINVAL`, its associated text (see `igraph_strerror()`) is "Invalid value" and this string should explain which parameter was invalid and maybe why.

igraph_errno:

The **igraph** error code.

5.5.2.2. igraph_error — Trigger an error.

```
int igraph_error(const char *reason, const char *file, int line,
   int igraph_errno);
```

igraph functions usually call this function (most often via the `IGRAPH_ERROR` macro) if they notice an error. It calls the currently installed error handler function with the supplied arguments.

Arguments:

reason:

Textual description of the error.

file:

The source file in which the error was noticed.

line:

The number of line in the source file which triggered the error.

igraph_errno:

The **igraph** error code.

Returns:

the error code (if it returns)

See also:

igraph_errorf().

5.5.2.3. `igraph_errorf` — Trigger an error, printf-like version.

```
int igraph_errorf(const char *reason, const char *file, int line,
    int igraph_errno, ...);
```

Arguments:

`reason`:

Textual description of the error, interpreted as a printf format string.

`file`:

The source file in which the error was noticed.

`line`:

The line in the source file which triggered the error.

`igraph_errno`:

The **igraph** error code.

`...`:

Additional parameters, the values to substitute into the format string.

See also:

igraph_error().

5.5.2.4. `IGRAPH_CHECK` — Check the return value of a function call.

```
#define IGRAPH_CHECK(a)
```

Arguments:

a:

An expression, usually a function call.

Executes the expression and checks its value. If this is not `IGRAPH_SUCCESS`, it calls `IGRAPH_ERROR` with the value as the error code. Here is an example usage:

```
IGRAPH_CHECK(vector_push_back(&v, 100));
```

There is only one reason to use this macro when writing **igraph** functions. If the user installs an error handler which returns to the auxiliary calling code (like `igraph_error_handler_ignore` and `igraph_error_handler_printignore`), and the **igraph** function signalling the error is called from another **igraph** function then we need to make sure that the error is propagated back to the auxiliary (ie. non-igraph) calling function. This is achieved by using `IGRAPH_CHECK` on every **igraph** call which can return an error code.

5.5.3. Deallocating memory

If a function runs into an error (and the program is not aborted) the error handler should deallocate all temporary memory. This is done by storing the address and the destroy function of all temporary objects in a stack. The `IGRAPH_FINALLY` function declares an object as temporary by placing its address in the stack. If an **igraph** function returns with success it calls `IGRAPH_FINALLY_CLEAN()` with the number of objects to remove from the stack. If an error happens however, the error handler should call `IGRAPH_FINALLY_FREE()` to deallocate each object added to the stack. This means that the temporary objects allocated in the calling function (and etc.) will be freed as well.

5.5.3.1. `IGRAPH_FINALLY` — Register an object for deallocation.

```
#define IGRAPH_FINALLY(func,ptr)
```

Arguments:

func:

The address of the function which is normally called to destroy the object.

ptr:

Pointer to the object itself.

This macro places the address of an object, together with the address of its destructor in a stack. This stack is used if an error happens to deallocate temporarily allocated objects to prevent memory leaks.

5.5.3.2. `IGRAPH_FINALLY_CLEAN` — Signal clean deallocation of objects.

```
void IGRAPH_FINALLY_CLEAN(int num);
```

Removes the specified number of objects from the stack of temporarily allocated objects. Most often this is called just before returning from a function.

Arguments:

`num`:

The number of objects to remove from the bookkeeping stack.

5.5.3.3. `IGRAPH_FINALLY_FREE` — Deallocate all registered objects.

```
void IGRAPH_FINALLY_FREE(void);
```

Calls the destroy function for all objects in the stack of temporarily allocated objects. This is usually called only from an error handler. It is *not* appropriate to use it instead of destroying each unneeded object of a function, as it destroys the temporary objects of the caller function (and so on) as well.

5.5.4. Writing igraph functions with proper error handling

There are some simple rules to keep in order to have functions behaving well in erroneous situations. First, check the arguments of the functions and call `IGRAPH_ERROR` if they are invalid. Second, call `IGRAPH_FINALLY` on each dynamically allocated object and call `IGRAPH_FINALLY_CLEAN()` with the proper argument before returning. Third, use `IGRAPH_CHECK` on all **igraph** function calls which can generate errors.

The size of the stack used for this bookkeeping is fixed, and small. If you want to allocate several objects, write a destroy function which can deallocate all of these. See the `adjlist.c` file in the **igraph** source for an example.

For some functions these mechanisms are simply not flexible enough. These functions should define their own error handlers and restore the error handler before they return.

5.5.5. Error handling and threads

It is likely that the **igraph** error handling method is *not* thread-safe, mainly because of the static global stack which is used to store the address of the temporarily allocated objects. This issue might be addressed in a later version of **igraph**.

Chapter 6. Memory (de)allocation

6.1. `igraph_free` — Deallocate memory that was allocated by igraph functions

```
int igraph_free(void *p);
```

Some igraph functions return a pointer vector (igraph_vector_ptr_t) containing pointers to other igraph or other data types. These data types are dynamically allocated and have to be deallocated manually, if the user does not need them any more. This can be done by calling igraph_free on them.

Here is a complete example on how to use `igraph_free` properly.

```
#include <igraph.h>

int main(void)
{
   igraph_t graph;
   igraph_vector_ptr_t seps;
   long int i;

   igraph_famous(&graph, "tutte");
   igraph_vector_ptr_init(&seps, 0);
   igraph_minimum_size_separators(&graph, &seps);

   for (i=0; i<igraph_vector_ptr_size(&seps); i++) {
     igraph_vector_t *v=VECTOR(seps)[i];
     igraph_vector_print(v);
     igraph_vector_destroy(v);
     igraph_free(v);
   }

   igraph_vector_ptr_destroy(&seps);
   igraph_destroy(&graph);
   return 0;
}
```

Arguments:

`p`:

Pointer to the piece of memory to be deallocated.

Returns:

Error code, currently always zero, meaning success.

Time complexity: platform dependent, ideally it should be O(1).

Chapter 7. Data structure library: vector, matrix, other data types

7.1. About template types

Some of the container types listed in this section are defined for many base types. This is similar to templates in C++ and generics in Ada, but it is implemented via precompiler macros since the C language cannot handle it. Here is the list of template types and the all base types they currently support:

vector

Vector is currently defined for igraph_real_t, long int (long), char (char), igraph_bool_t (bool). The default is igraph_real_t.

matrix

Matrix is currently defined for igraph_real_t, long int (long), char (char), igraph_bool_t (bool). The default is igraph_real_t.

array3

Array3 is currently defined for igraph_real_t, long int (long), char (char), igraph_bool_t (bool). The default is igraph_real_t.

stack

Stack is currently defined for igraph_real_t, long int (long), char (char), igraph_bool_t (bool). The default is igraph_real_t.

double-ended queue

Dqueue is currently defined for igraph_real_t, long int (long), char (char), igraph_bool_t (bool). The default is igraph_real_t.

heap

Heap is currently defined for igraph_real_t, long int (long), char (char). In addition both maximum and minimum heaps are available. The default is the igraph_real_t maximum heap.

The name of the base element (in parens) is added to the function names, except for te default type.

Some examples:

- igraph_vector_t is a vector of igraph_real_t elements. Its functions are `igraph_vector_init`, `igraph_vector_destroy`, `igraph_vector_sort`, etc.
- igraph_vector_bool_t is a vector of igraph_bool_t elements, initialize it with `igraph_vector_bool_init`, destroy it with `igraph_vector_bool_destroy`, etc.
- igraph_heap_t is a maximum heap with igraph_real_t elements. The corresponding functions are `igraph_heap_init`, `igraph_heap_pop`, etc.
- igraph_heap_min_t is a minimum heap with igraph_real_t elements. The corresponding functions are called `igraph_heap_min_init`, `igraph_heap_min_pop`, etc.
- igraph_heap_long_t is a maximum heap with long int elements. Its function have the `igraph_heap_long_` prefix.
- igraph_heap_min_long_t is a minimum heap containing long int elements. Its functions have the `igraph_heap_min_long_` prefix.

Note that the VECTOR and the MATRIX macros can be used on *all* vector and matrix types.

7.2. Vectors

7.2.1. About igraph_vector_t objects

The igraph_vector_t data type is a simple and efficient interface to arrays containing numbers. It is something similar as (but much simpler than) the vector template in the C++ standard library.

Vectors are used extensively in **igraph**, all functions which expect or return a list of numbers use igraph_vector_t to achieve this.

The igraph_vector_t type usually uses O(n) space to store n elements. Sometimes it uses more, this is because vectors can shrink, but even if they shrink, the current implementation does not free a single bit of memory.

The elements in an igraph_vector_t object are indexed from zero, we follow the usual C convention here.

The elements of a vector always occupy a single block of memory, the starting address of this memory block can be queried with the `VECTOR` macro. This way, vector objects can be used with standard mathematical libraries, like the GNU Scientific Library.

7.2.2. Constructors and Destructors

igraph_vector_t objects have to be initialized before using them, this is analogous to calling a constructor on them. There are a number of igraph_vector_t constructors, for your convenience. `igraph_vector_init()` is the basic constructor, it creates a vector of the given length, filled with zeros. `igraph_vector_copy()` creates a new identical copy of an already existing and initialized vector. `igraph_vector_init_copy()` creates a vector by copying a regular C array. `igraph_vector_init_seq()` creates a vector containing a regular sequence with increment one.

`igraph_vector_view()` is a special constructor, it allows you to handle a regular C array as a vector without copying its elements.

If a igraph_vector_t object is not needed any more, it should be destroyed to free its allocated memory by calling the igraph_vector_t destructor, `igraph_vector_destroy()`.

Note that vectors created by `igraph_vector_view()` are special, you mustn't call `igraph_vector_destroy()` on these.

7.2.2.1. `igraph_vector_init` — Initializes a vector object (constructor).

```
int igraph_vector_init      (igraph_vector_t* v, int long size);
```

Every vector needs to be initialized before it can be used, and there are a number of initialization functions or otherwise called constructors. This function constructs a vector of the given size and initializes each entry to 0. Note that `igraph_vector_null()` can be used to set each element of a vector to zero. However, if you want a vector of zeros, it is much faster to use this function than to create a vector and then invoke `igraph_vector_null()`.

Every vector object initialized by this function should be destroyed (ie. the memory allocated for it should be freed) when it is not needed anymore, the `igraph_vector_destroy()` function is responsible for this.

Arguments:

`v`:

Pointer to a not yet initialized vector object.

`size`:

The size of the vector.

Returns:

error code: `IGRAPH_ENOMEM` if there is not enough memory.

Time complexity: operating system dependent, the amount of “time” required to allocate O(n) elements, n is the number of elements.

7.2.2.2. `igraph_vector_init_copy` — Initializes a vector from an ordinary C array (constructor).

```
int igraph_vector_init_copy(igraph_vector_t *v,
          igraph_real_t *data, long int length);
```

Arguments:

`v`:

Pointer to an uninitialized vector object.

`data`:

A regular C array.

`length`:

The length of the C array.

Returns:

Error code: `IGRAPH_ENOMEM` if there is not enough memory.

Time complexity: operating system specific, usually O(*length*).

7.2.2.3. `igraph_vector_init_seq` — Initializes a vector with a sequence.

```
int igraph_vector_init_seq(igraph_vector_t *v,
         igraph_real_t from, igraph_real_t to);
```

The vector will contain the numbers *from*, *from*+1, ..., *to*.

Arguments:

v:

Pointer to an uninitialized vector object.

from:

The lower limit in the sequence (inclusive).

to:

The upper limit in the sequence (inclusive).

Returns:

Error code: `IGRAPH_ENOMEM`: out of memory.

Time complexity: O(n), the number of elements in the vector.

7.2.2.4. `igraph_vector_copy` — Initializes a vector from another vector object (constructor).

```
int igraph_vector_copy(igraph_vector_t *to,
     const igraph_vector_t *from);
```

The contents of the existing vector object will be copied to the new one.

Arguments:

`to`:

Pointer to a not yet initialized vector object.

`from`:

The original vector object to copy.

Returns:

Error code: `IGRAPH_ENOMEM` if there is not enough memory.

Time complexity: operating system dependent, usually O(n), n is the size of the vector.

7.2.2.5. `igraph_vector_destroy` — Destroys a vector object.

```
void igraph_vector_destroy   (igraph_vector_t* v);
```

All vectors initialized by `igraph_vector_init()` should be properly destroyed by this function. A destroyed vector needs to be reinitialized by `igraph_vector_init()`, `igraph_vector_init_copy()` or another constructor.

Arguments:

`v`:

Pointer to the (previously initialized) vector object to destroy.

Time complexity: operating system dependent.

7.2.3. Initializing elements

7.2.3.1. `igraph_vector_null` — Sets each element in the vector to zero.

```
void igraph_vector_null      (igraph_vector_t* v);
```

Note that `igraph_vector_init()` sets the elements to zero as well, so it makes no sense to call this function on a just initialized vector. Thus if you want to construct a vector of zeros, then you should use `igraph_vector_init()`.

Arguments:

`v`:

The vector object.

Time complexity: O(n), the size of the vector.

7.2.3.2. `igraph_vector_fill` — Fill a vector with a constant element

```
void igraph_vector_fill      (igraph_vector_t* v, igraph_real_t e);
```

Sets each element of the vector to the supplied constant.

Arguments:

`vector`:

The vector to work on.

`e`:

The element to fill with.

Time complexity: O(n), the size of the vector.

7.2.4. Accessing elements

The simplest way to access an element of a vector is to use the `VECTOR` macro. This macro can be used both for querying and setting igraph_vector_t elements. If you need a function, `igraph_vector_e()` queries and `igraph_vector_set()` sets an element of a vector. `igraph_vector_e_ptr()` returns the address of an element.

`igraph_vector_tail()` returns the last element of a non-empty vector. There is no `igraph_vector_head()` function however, as it is easy to write `VECTOR(v)[0]` instead.

7.2.4.1. `VECTOR` — Accessing an element of a vector.

```
#define VECTOR(v)
```

Usage:

```
VECTOR(v)[0]
```

to access the first element of the vector, you can also use this in assignments, like:

```
VECTOR(v)[10]=5;
```

Note that there are no range checks right now. This functionality might be redefined later as a real function instead of a `#define`.

Arguments:

v:

The vector object.

Time complexity: O(1).

7.2.4.2. `igraph_vector_e` — Access an element of a vector.

```
igraph_real_t igraph_vector_e         (const igraph_vector_t* v, long int pos);
```

Arguments:

v:

The igraph_vector_t object.

pos:

The position of the element, the index of the first element is zero.

Returns:

The desired element.

See also:

`igraph_vector_e_ptr()` and the `VECTOR` macro.

Time complexity: O(1).

7.2.4.3. `igraph_vector_e_ptr` — Get the address of an element of a vector

```
igraph_real_t* igraph_vector_e_ptr  (const igraph_vector_t* v, long int pos);
```

Arguments:

v:

The igraph_vector_t object.

pos:

The position of the element, the position of the first element is zero.

Returns:

Pointer to the desired element.

See also:

`igraph_vector_e()` and the `VECTOR` macro.

Time complexity: O(1).

7.2.4.4. `igraph_vector_set` — Assignment to an element of a vector.

```
void igraph_vector_set       (igraph_vector_t* v,
     long int pos, igraph_real_t value);
```

Arguments:

v:

The igraph_vector_t element.

pos:

Position of the element to set.

value:

New value of the element.

See also:

`igraph_vector_e()`.

7.2.4.5. `igraph_vector_tail` — Returns the last element in a vector.

```
igraph_real_t igraph_vector_tail(const igraph_vector_t *v);
```

It is an error to call this function on an empty vector, the result is undefined.

Arguments:

v:

The vector object.

Returns:

The last element.

Time complexity: O(1).

7.2.5. Vector views

7.2.5.1. `igraph_vector_view` — Handle a regular C array as a igraph_vector_t.

```
const igraph_vector_t*igraph_vector_view (const igraph_vector_t *v,
       const igraph_real_t *data,
       long int length);
```

This is a special igraph_vector_t constructor. It allows to handle a regular C array as a igraph_vector_t temporarily. Be sure that you *don't* ever call the destructor (`igraph_vector_destroy()`) on objects created by this constructor.

Arguments:

v:

Pointer to an uninitialized igraph_vector_t object.

data:

Pointer, the C array.

length:

The length of the C array.

Returns:

Pointer to the vector object, the same as the *v* parameter, for convenience.

Time complexity: O(1)

7.2.6. Copying vectors

7.2.6.1. `igraph_vector_copy_to` — Copies the contents of a vector to a C array.

```
void igraph_vector_copy_to(const igraph_vector_t *v, igraph_real_t *to);
```

The C array should have sufficient length.

Arguments:

v:

The vector object.

to:

The C array.

Time complexity: O(n), n is the size of the vector.

7.2.6.2. `igraph_vector_update` — Update a vector from another one.

```
int igraph_vector_update(igraph_vector_t *to,
       const igraph_vector_t *from);
```

After this operation the contents of *to* will be exactly the same *from*. *to* will be resized if it was originally shorter or longer than *from*.

Arguments:

to:

The vector to update.

from:

The vector to update from.

Returns:

Error code.

Time complexity: O(n), the number of elements in *from*.

7.2.6.3. `igraph_vector_append` — Append a vector to another one.

```
int igraph_vector_append(igraph_vector_t *to,
       const igraph_vector_t *from);
```

The target vector will be resized (except *from* is empty).

Arguments:

to:

The vector to append to.

from:

The vector to append, it is kept unchanged.

Returns:

Error code.

Time complexity: O(n), the number of elements in the new vector.

7.2.6.4. `igraph_vector_swap` — Swap elements of two vectors.

```
int igraph_vector_swap(igraph_vector_t *v1, igraph_vector_t *v2);
```

The two vectors must have the same length, otherwise an error happens.

Arguments:

v1:

The first vector.

v2:

The second vector.

Returns:

Error code.

Time complexity: O(n), the length of the vectors.

7.2.7. Exchanging elements

7.2.7.1. `igraph_vector_swap_elements` — Swap two elements in a vector.

```
int igraph_vector_swap_elements(igraph_vector_t *v,
       long int i, long int j);
```

Note that currently no range checking is performed.

Arguments:

v:

The input vector.

i:

Index of the first element.

j:

index of the second element. (Might be the same as the first.)

Returns:

Error code, currently always `IGRAPH_SUCCESS`.

Time complexity: O(1).

7.2.7.2. `igraph_vector_reverse` — Reverse the elements of a vector.

```
int igraph_vector_reverse(igraph_vector_t *v);
```

The first element will be last, the last element will be first, etc.

Arguments:

v:

The input vector.

Returns:

Error code, currently always `IGRAPH_SUCCESS`.

Time complexity: O(n), the number of elements.

7.2.7.3. `igraph_vector_shuffle` — Shuffles a vector in-place using the Fisher-Yates method

```
int igraph_vector_shuffle(igraph_vector_t *v);
```

The Fisher-Yates shuffle ensures that every implementation is equally probable when using a proper randomness source. Of course this does not apply to pseudo-random generators as the cycle of these generators is less than the number of possible permutations of the vector if the vector is long enough.

Arguments:

`v`:

The vector object.

Returns:

Error code, currently always `IGRAPH_SUCCESS`.

Time complexity: O(n), n is the number of elements in the vector.

References:

`(Fisher & Yates 1963)`

R. A. Fisher and F. Yates. *Statistical Tables for Biological, Agricultural and Medical Research.* Oliver and Boyd, 6th edition, 1963, page 37.

`(Knuth 1998)`

D. E. Knuth. *Seminumerical Algorithms,* volume 2 of *The Art of Computer Programming.* Addison-Wesley, 3rd edition, 1998, page 145.

Example 7-1. File examples/simple/igraph_fisher_yates_shuffle.c

7.2.8. Vector operations

7.2.8.1. `igraph_vector_add_constant` — Add a constant to the vector.

```
void igraph_vector_add_constant(igraph_vector_t *v, igraph_real_t plus);
```

`plus` is added to every element of `v`. Note that overflow might happen.

Arguments:

`v`:

The input vector.

`plus`:

The constant to add.

Time complexity: O(n), the number of elements.

7.2.8.2. `igraph_vector_scale` — Multiply all elements of a vector by a constant

```
void igraph_vector_scale(igraph_vector_t *v, igraph_real_t by);
```

Arguments:

`v`:

The vector.

`by`:

The constant.

Returns:

Error code. The current implementation always returns with success.

Added in version 0.2.

Time complexity: O(n), the number of elements in a vector.

7.2.8.3. `igraph_vector_add` — Add two vectors.

```
int igraph_vector_add(igraph_vector_t *v1,
    const igraph_vector_t *v2);
```

Add the elements of *v2* to *v1*, the result is stored in *v1*. The two vectors must have the same length.

Arguments:

v1:

The first vector, the result will be stored here.

v2:

The second vector, its contents will be unchanged.

Returns:

Error code.

Time complexity: O(n), the number of elements.

7.2.8.4. `igraph_vector_sub` — Subtract a vector from another one.

```
int igraph_vector_sub(igraph_vector_t *v1,
    const igraph_vector_t *v2);
```

Subtract the elements of *v2* from *v1*, the result is stored in *v1*. The two vectors must have the same length.

Arguments:

v1:

The first vector, to subtract from. The result is stored here.

v2:

The vector to subtract, it will be unchanged.

Returns:

Error code.

Time complexity: O(n), the length of the vectors.

7.2.8.5. `igraph_vector_mul` — Multiply two vectors.

```
int igraph_vector_mul(igraph_vector_t *v1,
    const igraph_vector_t *v2);
```

v1 will be multiplied by *v2*, elementwise. The two vectors must have the same length.

Arguments:

v1:

The first vector, the result will be stored here.

v2:

The second vector, it is left unchanged.

Returns:

Error code.

Time complexity: O(n), the number of elements.

7.2.8.6. `igraph_vector_div` — Divide a vector by another one.

```
int igraph_vector_div(igraph_vector_t *v1,
    const igraph_vector_t *v2);
```

v1 is divided by *v2*, elementwise. They must have the same length. If the base type of the vector can generate divide by zero errors then please make sure that *v2* contains no zero if you want to avoid trouble.

Arguments:

v1:

The dividend. The result is also stored here.

v2:

The divisor, it is left unchanged.

Returns:

Error code.

Time complexity: O(n), the length of the vectors.

7.2.9. Vector comparisons

7.2.9.1. `igraph_vector_all_e` — Are all elements equal?

```
igraph_bool_t igraph_vector_all_e(const igraph_vector_t *lhs,
          const igraph_vector_t *rhs);
```

Arguments:

lhs:

The first vector.

rhs:

The second vector.

Returns:

Positive integer (=true) if the elements in the *lhs* are all equal to the corresponding elements in *rhs*. Returns `0` (=false) if the lengths of the vectors don't match.

Time complexity: O(n), the length of the vectors.

7.2.9.2. `igraph_vector_all_l` — Are all elements less?

```
igraph_bool_t igraph_vector_all_l(const igraph_vector_t *lhs,
         const igraph_vector_t *rhs);
```

Arguments:

`lhs`:

The first vector.

`rhs`:

The second vector.

Returns:

Positive integer (=true) if the elements in the *`lhs`* are all less than the corresponding elements in *`rhs`*. Returns `0` (=false) if the lengths of the vectors don't match.

Time complexity: O(n), the length of the vectors.

7.2.9.3. `igraph_vector_all_g` — Are all elements greater?

```
igraph_bool_t igraph_vector_all_g(const igraph_vector_t *lhs,
         const igraph_vector_t *rhs);
```

Arguments:

`lhs`:

The first vector.

rhs:

The second vector.

Returns:

Positive integer (=true) if the elements in the *lhs* are all greater than the corresponding elements in *rhs*. Returns 0 (=false) if the lengths of the vectors don't match.

Time complexity: O(n), the length of the vectors.

7.2.9.4. igraph_vector_all_le — Are all elements less or equal?

```
igraph_bool_t
igraph_vector_all_le(const igraph_vector_t *lhs,
          const igraph_vector_t *rhs);
```

Arguments:

lhs:

The first vector.

rhs:

The second vector.

Returns:

Positive integer (=true) if the elements in the *lhs* are all less than or equal to the corresponding elements in *rhs*. Returns 0 (=false) if the lengths of the vectors don't match.

Time complexity: O(n), the length of the vectors.

7.2.9.5. igraph_vector_all_ge — Are all elements greater or equal?

```
igraph_bool_t
```

```
igraph_vector_all_ge(const igraph_vector_t *lhs,
         const igraph_vector_t *rhs);
```

Arguments:

lhs:

The first vector.

rhs:

The second vector.

Returns:

Positive integer (=true) if the elements in the *lhs* are all greater than or equal to the corresponding elements in *rhs*. Returns 0 (=false) if the lengths of the vectors don't match.

Time complexity: O(n), the length of the vectors.

7.2.10. Finding minimum and maximum

7.2.10.1. `igraph_vector_min` — Smallest element of a vector.

```
igraph_real_t igraph_vector_min(const igraph_vector_t* v);
```

The vector must be non-empty.

Arguments:

v:

The input vector.

Returns:

The smallest element of *v*.

Time complexity: O(n), the number of elements.

7.2.10.2. `igraph_vector_max` — Gives the maximum element of the vector.

```
igraph_real_t igraph_vector_max(const igraph_vector_t* v);
```

If the size of the vector is zero, an arbitrary number is returned.

Arguments:

v:

The vector object.

Returns:

The maximum element.

Time complexity: O(n), n is the size of the vector.

7.2.10.3. `igraph_vector_which_min` — Index of the smallest element.

```
long int igraph_vector_which_min(const igraph_vector_t* v);
```

The vector must be non-empty. If the smallest element is not unique, then the index of the first is returned.

Arguments:

v:

The input vector.

Returns:

Index of the smallest element.

Time complexity: O(n), the number of elements.

7.2.10.4. `igraph_vector_which_max` — Gives the position of the maximum element of the vector.

```
long int igraph_vector_which_max(const igraph_vector_t* v);
```

If the size of the vector is zero, -1 is returned.

Arguments:

v:

The vector object.

Returns:

The position of the first maximum element.

Time complexity: O(n), n is the size of the vector.

7.2.10.5. `igraph_vector_minmax` — Minimum and maximum elements of a vector.

```
int igraph_vector_minmax(const igraph_vector_t *v,
       igraph_real_t *min, igraph_real_t *max);
```

Handy if you want to have both the smallest and largest element of a vector. The vector is only traversed once. The vector must by non-empty.

Arguments:

v:

The input vector. It must contain at least one element.

min:

Pointer to a base type variable, the minimum is stored here.

max:

Pointer to a base type variable, the maximum is stored here.

Returns:

Error code.

Time complexity: O(n), the number of elements.

7.2.10.6. `igraph_vector_which_minmax` — Index of the minimum and maximum elements

```
int igraph_vector_which_minmax(const igraph_vector_t *v,
      long int *which_min, long int *which_max);
```

Handy if you need the indices of the smallest and largest elements. The vector is traversed only once. The vector must to non-empty.

Arguments:

v:

The input vector. It must contain at least one element.

which_min:

The index of the minimum element will be stored here.

which_max:

The index of the maximum element will be stored here.

Returns:

Error code.

Time complexity: O(n), the number of elements.

7.2.11. Vector properties

7.2.11.1. `igraph_vector_empty` — Decides whether the size of the vector is zero.

```
igraph_bool_t igraph_vector_empty     (const igraph_vector_t* v);
```

Arguments:

v:

The vector object.

Returns:

Non-zero number if the size of the vector is not zero and zero otherwise.

Time complexity: O(1).

7.2.11.2. `igraph_vector_size` — Gives the size (=length) of the vector.

```
long int igraph_vector_size      (const igraph_vector_t* v);
```

Arguments:

v:

The vector object

Returns:

The size of the vector.

Time complexity: O(1).

7.2.11.3. `igraph_vector_capacity` — Returns the allocated capacity of the vector

```
long int igraph_vector_capacity(const igraph_vector_t*v);
```

Note that this might be different from the size of the vector (as queried by `igraph_vector_size()`, and specifies how many elements the vector can hold, without reallocation.

Arguments:

`v`:

Pointer to the (previously initialized) vector object to query.

Returns:

The allocated capacity.

See also:

`igraph_vector_size()`.

Time complexity: O(1).

7.2.11.4. `igraph_vector_sum` — Calculates the sum of the elements in the vector.

```
igraph_real_t igraph_vector_sum(const igraph_vector_t *v);
```

For the empty vector 0.0 is returned.

Arguments:

v:

The vector object.

Returns:

The sum of the elements.

Time complexity: O(n), the size of the vector.

7.2.11.5. `igraph_vector_prod` — Calculates the product of the elements in the vector.

```
igraph_real_t igraph_vector_prod(const igraph_vector_t *v);
```

For the empty vector one (1) is returned.

Arguments:

v:

The vector object.

Returns:

The product of the elements.

Time complexity: O(n), the size of the vector.

7.2.11.6. `igraph_vector_isininterval` — Checks if all elements of a vector are in the given

```
igraph_bool_t igraph_vector_isininterval(const igraph_vector_t *v,
        igraph_real_t low,
        igraph_real_t high);
```

interval.

Arguments:

`v`:

The vector object.

`low`:

The lower limit of the interval (inclusive).

`high`:

The higher limit of the interval (inclusive).

Returns:

True (positive integer) if all vector elements are in the interval, false (zero) otherwise.

Time complexity: O(n), the number of elements in the vector.

7.2.11.7. `igraph_vector_maxdifference` — The maximum absolute difference of *`m1`* and *`m2`*

```
igraph_real_t igraph_vector_maxdifference(const igraph_vector_t *m1,
        const igraph_vector_t *m2);
```

The element with the largest absolute value in *`m1`* - *`m2`* is returned. Both vectors must be non-empty, but they not need to have the same length, the extra elements in the longer vector are ignored.

Arguments:

m1:

The first vector.

m2:

The second vector.

Returns:

The maximum absolute difference of *m1* and *m2*.

Time complexity: O(n), the number of elements in the shorter vector.

7.2.12. Searching for elements

7.2.12.1. `igraph_vector_contains` — Linear search in a vector.

```
igraph_bool_t igraph_vector_contains(const igraph_vector_t *v,
            igraph_real_t e);
```

Check whether the supplied element is included in the vector, by linear search.

Arguments:

v:

The input vector.

e:

The element to look for.

Returns:

`TRUE` if the element is found and `FALSE` otherwise.

Time complexity: O(n), the length of the vector.

7.2.12.2. `igraph_vector_search` — Search from a given position

```
igraph_bool_t igraph_vector_search(const igraph_vector_t *v,
          long int from, igraph_real_t what,
          long int *pos);
```

The supplied element *what* is searched in vector *v*, starting from element index *from*. If found then the index of the first instance (after *from*) is stored in *pos*.

Arguments:

v:

The input vector.

from:

The index to start searching from. No range checking is performed.

what:

The element to find.

pos:

If not `NULL` then the index of the found element is stored here.

Returns:

Boolean, `TRUE` if the element was found, `FALSE` otherwise.

Time complexity: O(m), the number of elements to search, the length of the vector minus the *from* argument.

7.2.12.3. `igraph_vector_binsearch` — Finds an element by binary searching a sorted vector.

```
igraph_bool_t igraph_vector_binsearch(const igraph_vector_t *v,
      igraph_real_t what, long int *pos);
```

It is assumed that the vector is sorted. If the specified element (`what`) is not in the vector, then the position of where it should be inserted (to keep the vector sorted) is returned.

Arguments:

`v`:

The igraph_vector_t object.

`what`:

The element to search for.

`pos`:

Pointer to a long int. This is set to the position of an instance of `what` in the vector if it is present. If `v` does not contain `what` then `pos` is set to the position to which it should be inserted (to keep the the vector sorted of course).

Returns:

Positive integer (true) if `what` is found in the vector, zero (false) otherwise.

Time complexity: O(log(n)), n is the number of elements in `v`.

7.2.12.4. `igraph_vector_binsearch2` — Binary search, without returning the index.

```
igraph_bool_t igraph_vector_binsearch2(const igraph_vector_t *v,
       igraph_real_t what);
```

It is assumed that the vector is sorted.

Arguments:

`v`:

The igraph_vector_t object.

`what`:

The element to search for.

Returns:

Positive integer (true) if *what* is found in the vector, zero (false) otherwise.

Time complexity: O(log(n)), n is the number of elements in *v*.

7.2.13. Resizing operations

7.2.13.1. `igraph_vector_clear` — Removes all elements from a vector.

```
void igraph_vector_clear     (igraph_vector_t* v);
```

This function simply sets the size of the vector to zero, it does not free any allocated memory. For that you have to call `igraph_vector_destroy()`.

Arguments:

v:

The vector object.

Time complexity: O(1).

7.2.13.2. `igraph_vector_reserve` — Reserves memory for a vector.

```
int igraph_vector_reserve   (igraph_vector_t* v, long int size);
```

igraph vectors are flexible, they can grow and shrink. Growing however occasionally needs the data in the vector to be copied. In order to avoid this, you can call this function to reserve space for future growth of the vector.

Note that this function does *not* change the size of the vector. Let us see a small example to clarify things: if you reserve space for 100 elements and the size of your vector was (and still is) 60, then you can surely add additional 40 elements to your vector before it will be copied.

Arguments:

v:

The vector object.

size:

The new *allocated* size of the vector.

Returns:

Error code: IGRAPH_ENOMEM if there is not enough memory.

Time complexity: operating system dependent, should be around O(n), n is the new allocated size of the vector.

7.2.13.3. igraph_vector_resize — Resize the vector.

```
int igraph_vector_resize(igraph_vector_t* v, long int newsize);
```

Note that this function does not free any memory, just sets the size of the vector to the given one. It can on the other hand allocate more memory if the new size is larger than the previous one. In this case the newly appeared elements in the vector are *not* set to zero, they are uninitialized.

Arguments:

v:

The vector object

newsize:

The new size of the vector.

Returns:

Error code, `IGRAPH_ENOMEM` if there is not enough memory. Note that this function *never* returns an error if the vector is made smaller.

See also:

`igraph_vector_reserve()` for allocating memory for future extensions of a vector. `igraph_vector_resize_min()` for deallocating the unnneded memory for a vector.

Time complexity: O(1) if the new size is smaller, operating system dependent if it is larger. In the latter case it is usually around O(n), n is the new size of the vector.

7.2.13.4. `igraph_vector_resize_min` — Deallocate the unused memory of a vector.

```
int igraph_vector_resize_min(igraph_vector_t*v);
```

Note that this function involves additional memory allocation and may result an out-of-memory error.

Arguments:

v:

Pointer to an initialized vector.

Returns:

Error code.

See also:

`igraph_vector_resize()`, `igraph_vector_reserve()`.

Time complexity: operating system dependent.

7.2.13.5. `igraph_vector_push_back` — Appends one element to a vector.

```
int igraph_vector_push_back (igraph_vector_t* v, igraph_real_t e);
```

This function resizes the vector to be one element longer and sets the very last element in the vector to *e*.

Arguments:

v:

The vector object.

e:

The element to append to the vector.

Returns:

Error code: `IGRAPH_ENOMEM`: not enough memory.

Time complexity: operating system dependent. What is important is that a sequence of n subsequent calls to this function has time complexity O(n), even if there hadn't been any space reserved for the new elements by `igraph_vector_reserve()`. This is implemented by a trick similar to the C++ vector class: each time more memory is allocated for a vector, the size of the additionally allocated memory is the same as the vector's current length. (We assume here that the time complexity of memory allocation is at most linear.)

7.2.13.6. `igraph_vector_pop_back` — Removes and returns the last element of a vector.

```
igraph_real_t igraph_vector_pop_back(igraph_vector_t* v);
```

It is an error to call this function with an empty vector.

Arguments:

v:

The vector object.

Returns:

The removed last element.

Time complexity: O(1).

7.2.13.7. `igraph_vector_insert` — Inserts a single element into a vector.

```
int igraph_vector_insert(igraph_vector_t *v, long int pos,
       igraph_real_t value);
```

Note that this function does not do range checking. Insertion will shift the elements from the position given to the end of the vector one position to the right, and the new element will be inserted in the empty space created at the given position. The size of the vector will increase by one.

Arguments:

v:

The vector object.

pos:

The position where the new element is to be inserted.

value:

The new element to be inserted.

7.2.13.8. `igraph_vector_remove` — Removes a single element from a vector.

```
void igraph_vector_remove(igraph_vector_t *v, long int elem);
```

Note that this function does not do range checking.

Arguments:

v:

The vector object.

elem:

The position of the element to remove.

Time complexity: O(n-elem), n is the number of elements in the vector.

7.2.13.9. `igraph_vector_remove_section` — Deletes a section from a vector.

```
void igraph_vector_remove_section(igraph_vector_t *v,
         long int from, long int to);
```

Note that this function does not do range checking. The result is undefined if you supply invalid limits.

Arguments:

v:

The vector object.

from:

The position of the first element to remove.

to:

The position of the first element *not* to remove.

Time complexity: O(n-from), n is the number of elements in the vector.

7.2.14. Sorting

7.2.14.1. `igraph_vector_sort` — Sorts the elements of the vector into ascending order.

```
void igraph_vector_sort(igraph_vector_t *v);
```

This function uses the built-in sort function of the C library.

Arguments:

v:

Pointer to an initialized vector object.

Time complexity: should be O(nlogn) for n elements.

7.2.15. Set operations on sorted vectors

7.2.15.1. `igraph_vector_intersect_sorted` — Calculates the intersection of two sorted vectors

```
int igraph_vector_intersect_sorted(const igraph_vector_t *v1,
    const igraph_vector_t *v2, igraph_vector_t *result);
```

The elements that are contained in both vectors are stored in the result vector. All three vectors must be initialized.

Instead of the naive intersection which takes O(n), this function uses the set intersection method of Ricardo Baeza-Yates, which is more efficient when one of the vectors is significantly smaller than the other, and gives similar performance on average when the two vectors are equal.

The algorithm keeps the multiplicities of the elements: if an element appears k1 times in the first vector and k2 times in the second, the result will include that element min(k1, k2) times.

Reference: Baeza-Yates R: A fast set intersection algorithm for sorted sequences. In: Lecture Notes in Computer Science, vol. 3109/2004, pp. 400--408, 2004. Springer Berlin/Heidelberg. ISBN: 978-3-540-22341-2.

Arguments:

v1:

the first vector

v2:

the second vector

result:

the result vector, which will also be sorted.

Time complexity: O(m log(n)) where m is the size of the smaller vector and n is the size of the larger one.

7.2.15.2. `igraph_vector_difference_sorted` — Calculates the difference between two sorted vectors (considered as sets)

```
int igraph_vector_difference_sorted(const igraph_vector_t *v1,
    const igraph_vector_t *v2, igraph_vector_t *result);
```

The elements that are contained in only the first vector but not the second are stored in the result vector. All three vectors must be initialized.

Arguments:

v1:

the first vector

v2:

the second vector

result:

the result vector

7.2.16. Pointer vectors (igraph_vector_ptr_t)

The igraph_vector_ptr_t data type is very similar to the igraph_vector_t type, but it stores generic pointers instead of real numbers.

This type has the same space complexity as igraph_vector_t, and most implemented operations work the same way as for igraph_vector_t.

This type is mostly used to pass to or receive from a set of graphs to some **igraph** functions, such as `igraph_decompose()`, which decomposes a graph to connected components.

The same `VECTOR` macro used for ordinary vectors can be used for pointer vectors as well, please note that a typeless generic pointer will be provided by this macro and you may need to cast it to a specific pointer before starting to work with it.

Pointer vectors may have an associated item destructor function which takes a pointer and returns nothing. The item destructor will be called on each item in the pointer vector when it is destroyed by `igraph_vector_ptr_destroy()` or `igraph_vector_ptr_destroy_all()`, or when its elements are freed by `igraph_vector_ptr_free_all()`. Note that the semantics of an item destructor does not coincide with C++ destructors; for instance, when a pointer vector is resized to a smaller size, the extra items will *not* be destroyed automatically! Nevertheless, item destructors may become handy in many cases; for instance, a vector of graphs generated by `igraph_decompose()` can be destroyed with a single call to `igraph_vector_ptr_destroy_all()` if the item destructor is set to `igraph_destroy()`.

7.2.16.1. `igraph_vector_ptr_init` — Initialize a pointer vector (constructor).

```
int igraph_vector_ptr_init      (igraph_vector_ptr_t* v, int long size);
```

This is the constructor of the pointer vector data type. All pointer vectors constructed this way should be destroyed via calling `igraph_vector_ptr_destroy()`.

Arguments:

v:

Pointer to an uninitialized igraph_vector_ptr_t object, to be created.

size:

Integer, the size of the pointer vector.

Returns:

Error code: `IGRAPH_ENOMEM` if out of memory

Time complexity: operating system dependent, the amount of “time” required to allocate *size* elements.

7.2.16.2. `igraph_vector_ptr_copy` — Copy a pointer vector (constructor).

```
int igraph_vector_ptr_copy(igraph_vector_ptr_t *to, const igraph_vector_ptr_t *from);
```

This function creates a pointer vector by copying another one. This is shallow copy, only the pointers in the vector will be copied.

It is potentially dangerous to copy a pointer vector with an associated item destructor. The copied vector will inherit the item destructor, which may cause problems when both vectors are destroyed as the items might get destroyed twice. Make sure you know what you are doing when copying a pointer vector with an item destructor, or unset the item destructor on one of the vectors later.

Arguments:

to:

Pointer to an uninitialized pointer vector object.

from:

A pointer vector object.

Returns:

Error code: `IGRAPH_ENOMEM` if out of memory

Time complexity: O(n) if allocating memory for n elements can be done in O(n) time.

7.2.16.3. `igraph_vector_ptr_destroy` — Destroys a pointer vector.

```
void igraph_vector_ptr_destroy   (igraph_vector_ptr_t* v);
```

The destructor for pointer vectors.

Arguments:

v:

Pointer to the pointer vector to destroy.

Time complexity: operating system dependent, the “time” required to deallocate O(n) bytes, n is the number of elements allocated for the pointer vector (not necessarily the number of elements in the vector).

7.2.16.4. `igraph_vector_ptr_free_all` — Frees all the elements of a pointer vector.

```
void igraph_vector_ptr_free_all   (igraph_vector_ptr_t* v);
```

If an item destructor is set for this pointer vector, this function will first call the destructor on all elements of the vector and then free all the elements using free(). If an item destructor is not set, the elements will simply be freed.

Arguments:

v:

Pointer to the pointer vector whose elements will be freed.

Time complexity: operating system dependent, the “time” required to call the destructor n times and then deallocate O(n) pointers, each pointing to a memory area of arbitrary size. n is the number of elements in the pointer vector.

7.2.16.5. `igraph_vector_ptr_destroy_all` — Frees all the elements and destroys the pointer vector.

```
void igraph_vector_ptr_destroy_all   (igraph_vector_ptr_t* v);
```

This function is equivalent to `igraph_vector_ptr_free_all()` followed by `igraph_vector_ptr_destroy()`.

Arguments:

v:

Pointer to the pointer vector to destroy.

Time complexity: operating system dependent, the "time" required to deallocate O(n) pointers, each pointing to a memory area of arbitrary size, plus the "time" required to deallocate O(n) bytes, n being the number of elements allocated for the pointer vector (not necessarily the number of elements in the vector).

7.2.16.6. `igraph_vector_ptr_size` — Gives the number of elements in the pointer vector.

```
long int igraph_vector_ptr_size      (const igraph_vector_ptr_t* v);
```

Arguments:

v:

The pointer vector object.

Returns:

The size of the object, ie. the number of pointers stored.

Time complexity: O(1).

7.2.16.7. igraph_vector_ptr_clear — Removes all elements from a pointer vector.

```
void igraph_vector_ptr_clear     (igraph_vector_ptr_t* v);
```

This function resizes a pointer to vector to zero length. Note that the pointed objects are *not* deallocated, you should call free() on them, or make sure that their allocated memory is freed in some other way, you'll get memory leaks otherwise. If you have set up an item destructor earlier, the destructor will be called on every element.

Note that the current implementation of this function does *not* deallocate the memory required for storing the pointers, so making a pointer vector smaller this way does not give back any memory. This behavior might change in the future.

Arguments:

v:

The pointer vector to clear.

Time complexity: O(1).

7.2.16.8. igraph_vector_ptr_push_back — Appends an element to the back of a pointer vector.

```
int igraph_vector_ptr_push_back (igraph_vector_ptr_t* v, void* e);
```

Arguments:

v:

The pointer vector.

e:

The new element to include in the pointer vector.

Returns:

Error code.

See also:

igraph_vector_push_back() for the corresponding operation of the ordinary vector type.

Time complexity: O(1) or O(n), n is the number of elements in the vector. The pointer vector implementation ensures that n subsequent push_back operations need O(n) time to complete.

7.2.16.9. `igraph_vector_ptr_e` — Access an element of a pointer vector.

```
void* igraph_vector_ptr_e         (const igraph_vector_ptr_t* v, long int pos);
```

Arguments:

v:

Pointer to a pointer vector.

pos:

The index of the pointer to return.

Returns:

The pointer at *pos* position.

Time complexity: O(1).

7.2.16.10. `igraph_vector_ptr_set` — Assign to an element of a pointer vector.

```
void igraph_vector_ptr_set       (igraph_vector_ptr_t* v, long int pos, void* value);
```

Arguments:

`v`:

Pointer to a pointer vector.

`pos`:

The index of the pointer to update.

`value`:

The new pointer to set in the vector.

Time complexity: O(1).

7.2.16.11. `igraph_vector_ptr_resize` — Resizes a pointer vector.

```
int igraph_vector_ptr_resize(igraph_vector_ptr_t* v, long int newsize);
```

Note that if a vector is made smaller the pointed object are not deallocated by this function and the item destructor is not called on the extra elements.

Arguments:

`v`:

A pointer vector.

`newsize`:

The new size of the pointer vector.

Returns:

Error code.

Time complexity: O(1) if the vector if made smaller. Operating system dependent otherwise, the amount of “time” needed to allocate the memory for the vector elements.

7.2.16.12. igraph_vector_ptr_get_item_destructor — Gets the current item destructor for this pointer vector.

```
igraph_finally_func_t* igraph_vector_ptr_get_item_destructor(const igraph_vector_ptr_t *v);
```

The item destructor is a function which will be called on every non-null pointer stored in this vector when `igraph_vector_ptr_destroy()`, igraph_vector_ptr_destroy_all() or `igraph_vector_ptr_free_all()` is called.

Returns:

The current item destructor.

Time complexity: O(1).

7.2.16.13. igraph_vector_ptr_set_item_destructor — Sets the item destructor for this pointer vector.

```
igraph_finally_func_t* igraph_vector_ptr_set_item_destructor(
        igraph_vector_ptr_t *v, igraph_finally_func_t *func);
```

The item destructor is a function which will be called on every non-null pointer stored in this vector when `igraph_vector_ptr_destroy()`, igraph_vector_ptr_destroy_all() or `igraph_vector_ptr_free_all()` is called.

Returns:

The old item destructor.

Time complexity: O(1).

7.2.16.14. IGRAPH_VECTOR_PTR_SET_ITEM_DESTRUCTOR — Sets the item destructor for this pointer vector (macro version).

```
#define IGRAPH_VECTOR_PTR_SET_ITEM_DESTRUCTOR(v,
```

This macro is expanded to `igraph_vector_ptr_set_item_destructor()`, the only difference is that the second argument is automatically cast to an `igraph_finally_func_t*`. The cast is necessary in most cases as the destructor functions we use (such as `igraph_vector_destroy()`) take a pointer to some concrete igraph data type, while `igraph_finally_func_t` expects `void*`

7.3. Matrices

7.3.1. About igraph_matrix_t objects

This type is just an interface to igraph_vector_t.

The igraph_matrix_t type usually stores n elements in O(n) space, but not always. See the documentation of the vector type.

7.3.2. Matrix constructors and destructors

7.3.2.1. `igraph_matrix_init` — Initializes a matrix.

```
int igraph_matrix_init(igraph_matrix_t *m, long int nrow, long int ncol);
```

Every matrix needs to be initialized before using it. This is done by calling this function. A matrix has to be destroyed if it is not needed any more; see `igraph_matrix_destroy()`.

Arguments:

m:

Pointer to a not yet initialized matrix object to be initialized.

nrow:

The number of rows in the matrix.

ncol:

The number of columns in the matrix.

Returns:

Error code.

Time complexity: usually O(n), n is the number of elements in the matrix.

7.3.2.2. `igraph_matrix_copy` — Copies a matrix.

```
int igraph_matrix_copy(igraph_matrix_t *to, const igraph_matrix_t *from);
```

Creates a matrix object by copying from an existing matrix.

Arguments:

`to`:

Pointer to an uninitialized matrix object.

`from`:

The initialized matrix object to copy.

Returns:

Error code, `IGRAPH_ENOMEM` if there isn't enough memory to allocate the new matrix.

Time complexity: O(n), the number of elements in the matrix.

7.3.2.3. `igraph_matrix_destroy` — Destroys a matrix object.

```
void igraph_matrix_destroy(igraph_matrix_t *m);
```

This function frees all the memory allocated for a matrix object. The destroyed object needs to be reinitialized before using it again.

Arguments:

m:

The matrix to destroy.

Time complexity: operating system dependent.

7.3.3. Initializing elements

7.3.3.1. `igraph_matrix_null` — Sets all elements in a matrix to zero.

```
void igraph_matrix_null(igraph_matrix_t *m);
```

Arguments:

m:

Pointer to an initialized matrix object.

Time complexity: O(n), n is the number of elements in the matrix.

7.3.3.2. `igraph_matrix_fill` — Fill with an element.

```
void igraph_matrix_fill(igraph_matrix_t *m, igraph_real_t e);
```

Set the matrix to a constant matrix.

Arguments:

m:

The input matrix.

e:

The element to set.

Time complexity: O(mn), the number of elements.

7.3.4. Copying matrices

7.3.4.1. `igraph_matrix_copy_to` — Copies a matrix to a regular C array.

```
void igraph_matrix_copy_to(const igraph_matrix_t *m, igraph_real_t *to);
```

The matrix is copied columnwise, as this is the format most programs and languages use. The C array should be of sufficient size; there are (of course) no range checks.

Arguments:

m:

Pointer to an initialized matrix object.

to:

Pointer to a C array; the place to copy the data to.

Returns:

Error code.

Time complexity: O(n), n is the number of elements in the matrix.

7.3.4.2. `igraph_matrix_update` — Update from another matrix.

```
int igraph_matrix_update(igraph_matrix_t *to,
       const igraph_matrix_t *from);
```

This function replicates *from* in the matrix *to*. Note that *to* must be already initialized.

Arguments:

to:

The result matrix.

from:

The matrix to replicate; it is left unchanged.

Returns:

Error code.

Time complexity: O(mn), the number of elements.

7.3.4.3. `igraph_matrix_swap` — Swap two matrices.

```
int igraph_matrix_swap(igraph_matrix_t *m1, igraph_matrix_t *m2);
```

The contents of the two matrices will be swapped. They must have the same dimensions.

Arguments:

m1:

The first matrix.

m2:

The second matrix.

Returns:

Error code.

Time complexity: O(mn), the number of elements in the matrices.

7.3.5. Accessing elements of a matrix

7.3.5.1. MATRIX — Accessing an element of a matrix.

```
#define MATRIX(m,i,j)
```

Note that there are no range checks right now. This functionality might be redefines as a proper function later.

Arguments:

m:

The matrix object.

i:

The index of the row, starting with zero.

j:

The index of the column, starting with zero.

Time complexity: O(1).

7.3.5.2. igraph_matrix_e — Extract an element from a matrix.

```
igraph_real_t igraph_matrix_e(const igraph_matrix_t *m,
          long int row, long int col);
```

Use this if you need a function for some reason and cannot use the `MATRIX` macro. Note that no range checking is performed.

Arguments:

m:

The input matrix.

row:

The row index.

col:

The column index.

Returns:

The element in the given row and column.

Time complexity: O(1).

7.3.5.3. `igraph_matrix_e_ptr` — Pointer to an element of a matrix.

```
igraph_real_t* igraph_matrix_e_ptr(const igraph_matrix_t *m,
        long int row, long int col);
```

The function returns a pointer to an element. No range checking is performed.

Arguments:

m:

The input matrix.

row:

The row index.

col:

The column index.

Returns:

Pointer to the element in the given row and column.

Time complexity: O(1).

7.3.5.4. `igraph_matrix_set` — Set an element.

```
void igraph_matrix_set(igraph_matrix_t* m, long int row, long int col,
     igraph_real_t value);
```

Set an element of a matrix. No range checking is performed.

Arguments:

m:

The input matrix.

row:

The row index.

col:

The column index.

value:

The new value of the element.

Time complexity: O(1).

7.3.6. Operations on rows and columns

7.3.6.1. `igraph_matrix_get_row` — Extract a row.

```
int igraph_matrix_get_row(const igraph_matrix_t *m,
        igraph_vector_t *res, long int index);
```

Extract a row from a matrix and return it as a vector.

Arguments:

m:

The input matrix.

res:

Pointer to an initialized vector; it will be resized if needed.

index:

The index of the row to select.

Returns:

Error code.

Time complexity: O(n), the number of columns in the matrix.

7.3.6.2. igraph_matrix_get_col — Select a column.

```
int igraph_matrix_get_col(const igraph_matrix_t *m,
        igraph_vector_t *res,
        long int index);
```

Extract a column of a matrix and return it as a vector.

Arguments:

m:

The input matrix.

res:

The result will we stored in this vector. It should be initialized and will be resized as needed.

index:

The index of the column to select.

Returns:

Error code.

Time complexity: O(n), the number of rows in the matrix.

7.3.6.3. `igraph_matrix_set_row` — Set a row from a vector.

```
int igraph_matrix_set_row(igraph_matrix_t *m,
        const igraph_vector_t *v, long int index);
```

Sets the elements of a row with the given vector. This has the effect of setting row `index` to have the elements in the vector `v`. The length of the vector and the number of columns in the matrix must match, otherwise an error is triggered.

Arguments:

m:

The input matrix.

v:

The vector containing the new elements of the row.

index:

Index of the row to set.

Returns:

Error code.

Time complexity: O(n), the number of columns in the matrix.

7.3.6.4. `igraph_matrix_set_col` — Set a column from a vector.

```
int igraph_matrix_set_col(igraph_matrix_t *m,
        const igraph_vector_t *v, long int index);
```

Sets the elements of a column with the given vector. In effect, column `index` will be set with elements from the vector `v`. The length of the vector and the number of rows in the matrix must match, otherwise an error is triggered.

Arguments:

m:

The input matrix.

v:

The vector containing the new elements of the column.

index:

Index of the column to set.

Returns:

Error code.

Time complexity: O(m), the number of rows in the matrix.

7.3.6.5. `igraph_matrix_swap_rows` — Swap two rows.

```
int igraph_matrix_swap_rows(igraph_matrix_t *m,
            long int i, long int j);
```

Swap two rows in the matrix.

Arguments:

m:

The input matrix.

i:

The index of the first row.

j:

The index of the second row.

Returns:

Error code.

Time complexity: O(n), the number of columns.

7.3.6.6. `igraph_matrix_swap_cols` — Swap two columns.

```
int igraph_matrix_swap_cols(igraph_matrix_t *m,
          long int i, long int j);
```

Swap two columns in the matrix.

Arguments:

m:

The input matrix.

i:

The index of the first column.

j:

The index of the second column.

Returns:

Error code.

Time complexity: O(m), the number of rows.

7.3.6.7. `igraph_matrix_select_rows` — Select some rows of a matrix.

```
int igraph_matrix_select_rows(const igraph_matrix_t *m,
     igraph_matrix_t *res,
     const igraph_vector_t *rows);
```

This function selects some rows of a matrix and returns them in a new matrix. The result matrix should be initialized before calling the function.

Arguments:

m:

The input matrix.

res:

The result matrix. It should be initialized and will be resized as needed.

rows:

Vector; it contains the row indices (starting with zero) to extract. Note that no range checking is performed.

Returns:

Error code.

Time complexity: O(nm), n is the number of rows, m the number of columns of the result matrix.

7.3.6.8. `igraph_matrix_select_cols` — Select some columns of a matrix.

```
int igraph_matrix_select_cols(const igraph_matrix_t *m,
     igraph_matrix_t *res,
     const igraph_vector_t *cols);
```

This function selects some columns of a matrix and returns them in a new matrix. The result matrix should be initialized before calling the function.

Arguments:

m:

The input matrix.

res:

The result matrix. It should be initialized and will be resized as needed.

cols:

Vector; it contains the column indices (starting with zero) to extract. Note that no range checking is performed.

Returns:

Error code.

Time complexity: O(nm), n is the number of rows, m the number of columns of the result matrix.

7.3.6.9. `igraph_matrix_select_rows_cols` — Select some rows and columns of a matrix.

```
int igraph_matrix_select_rows_cols(const igraph_matrix_t *m,
     igraph_matrix_t *res,
     const igraph_vector_t *rows,
     const igraph_vector_t *cols);
```

This function selects some rows and columns of a matrix and returns them in a new matrix. The result matrix should be initialized before calling the function.

Arguments:

m:

The input matrix.

res:

The result matrix. It should be initialized and will be resized as needed.

rows:

Vector; it contains the row indices (starting with zero) to extract. Note that no range checking is performed.

cols:

Vector; it contains the column indices (starting with zero) to extract. Note that no range checking is performed.

Returns:

Error code.

Time complexity: O(nm), n is the number of rows, m the number of columns of the result matrix.

7.3.7. Matrix operations

7.3.7.1. `igraph_matrix_add_constant` — Add a constant to every element.

```
void igraph_matrix_add_constant(igraph_matrix_t *m, igraph_real_t plus);
```

Arguments:

m:

The input matrix.

plud:

The constant to add.

Time complexity: O(mn), the number of elements.

7.3.7.2. `igraph_matrix_scale` — Multiplies each element of the matrix by a constant.

```
void igraph_matrix_scale(igraph_matrix_t *m, igraph_real_t by);
```

Arguments:

m:

The matrix.

by:

The constant.

Added in version 0.2.

Time complexity: O(n), the number of elements in the matrix.

7.3.7.3. `igraph_matrix_add` — Add two matrices.

```
int igraph_matrix_add(igraph_matrix_t *m1,
    const igraph_matrix_t *m2);
```

Add *m2* to *m1*, and store the result in *m1*. The dimensions of the matrices must match.

Arguments:

m1:

The first matrix; the result will be stored here.

m2:

The second matrix; it is left unchanged.

Returns:

Error code.

Time complexity: O(mn), the number of elements.

7.3.7.4. `igraph_matrix_sub` — Difference of two matrices.

```
int igraph_matrix_sub(igraph_matrix_t *m1,
    const igraph_matrix_t *m2);
```

Subtract *m2* from *m1* and store the result in *m1*. The dimensions of the two matrices must match.

Arguments:

m1:

The first matrix; the result is stored here.

m2:

The second matrix; it is left unchanged.

Returns:

Error code.

Time complexity: O(mn), the number of elements.

7.3.7.5. `igraph_matrix_mul_elements` — Elementwise multiplication.

```
int igraph_matrix_mul_elements(igraph_matrix_t *m1,
      const igraph_matrix_t *m2);
```

Multiply *m1* by *m2* elementwise and store the result in *m1*. The dimensions of the two matrices must match.

Arguments:

m1:

The first matrix; the result is stored here.

m2:

The second matrix; it is left unchanged.

Returns:

Error code.

Time complexity: O(mn), the number of elements.

7.3.7.6. `igraph_matrix_div_elements` — Elementwise division.

```
int igraph_matrix_div_elements(igraph_matrix_t *m1,
      const igraph_matrix_t *m2);
```

Divide *m1* by *m2* elementwise and store the result in *m1*. The dimensions of the two matrices must match.

Arguments:

m1:

The dividend. The result is store here.

m2:

The divisor. It is left unchanged.

Returns:

Error code.

Time complexity: O(mn), the number of elements.

7.3.7.7. `igraph_matrix_sum` — Sum of elements.

```
igraph_real_t igraph_matrix_sum(const igraph_matrix_t *m);
```

Returns the sum of the elements of a matrix.

Arguments:

m:

The input matrix.

Returns:

The sum of the elements.

Time complexity: O(mn), the number of elements in the matrix.

7.3.7.8. `igraph_matrix_prod` — Product of the elements.

```
igraph_real_t igraph_matrix_prod(const igraph_matrix_t *m);
```

Note this function can result in overflow easily, even for not too big matrices.

Arguments:

m:

The input matrix.

Returns:

The product of the elements.

Time complexity: O(mn), the number of elements.

7.3.7.9. `igraph_matrix_rowsum` — Rowwise sum.

```
int igraph_matrix_rowsum(const igraph_matrix_t *m,
       igraph_vector_t *res);
```

Calculate the sum of the elements in each row.

Arguments:

m:

The input matrix.

res:

Pointer to an initialized vector; the result is stored here. It will be resized if necessary.

Returns:

Error code.

Time complexity: O(mn), the number of elements in the matrix.

7.3.7.10. `igraph_matrix_colsum` — Columnwise sum.

```
int igraph_matrix_colsum(const igraph_matrix_t *m,
       igraph_vector_t *res);
```

Calculate the sum of the elements in each column.

Arguments:

m:

The input matrix.

res:

Pointer to an initialized vector; the result is stored here. It will be resized if necessary.

Returns:

Error code.

Time complexity: O(mn), the number of elements in the matrix.

7.3.7.11. `igraph_matrix_transpose` — Transpose a matrix.

```
int igraph_matrix_transpose(igraph_matrix_t *m);
```

Calculate the transpose of a matrix. Note that the function reallocates the memory used for the matrix.

Arguments:

m:

The input (and output) matrix.

Returns:

Error code.

Time complexity: O(mn), the number of elements in the matrix.

7.3.8. Matrix comparisons

7.3.8.1. `igraph_matrix_all_e` — Are all elements equal?

```
igraph_bool_t igraph_matrix_all_e(const igraph_matrix_t *lhs,
        const igraph_matrix_t *rhs);
```

Arguments:

`lhs`:

The first matrix.

`rhs`:

The second matrix.

Returns:

Positive integer (=true) if the elements in the *`lhs`* are all equal to the corresponding elements in *`rhs`*. Returns `0` (=false) if the dimensions of the matrices don't match.

Time complexity: O(nm), the size of the matrices.

7.3.8.2. `igraph_matrix_all_l` — Are all elements less?

```
igraph_bool_t igraph_matrix_all_l(const igraph_matrix_t *lhs,
        const igraph_matrix_t *rhs);
```

Arguments:

lhs:

The first matrix.

rhs:

The second matrix.

Returns:

Positive integer (=true) if the elements in the *lhs* are all less than the corresponding elements in *rhs*. Returns `0` (=false) if the dimensions of the matrices don't match.

Time complexity: O(nm), the size of the matrices.

7.3.8.3. `igraph_matrix_all_g` — Are all elements greater?

```
igraph_bool_t igraph_matrix_all_g(const igraph_matrix_t *lhs,
          const igraph_matrix_t *rhs);
```

Arguments:

lhs:

The first matrix.

rhs:

The second matrix.

Returns:

Positive integer (=true) if the elements in the *lhs* are all greater than the corresponding elements in *rhs*. Returns `0` (=false) if the dimensions of the matrices don't match.

Time complexity: O(nm), the size of the matrices.

7.3.8.4. `igraph_matrix_all_le` — Are all elements less or equal?

```
igraph_bool_t
igraph_matrix_all_le(const igraph_matrix_t *lhs,
          const igraph_matrix_t *rhs);
```

Arguments:

lhs:

The first matrix.

rhs:

The second matrix.

Returns:

Positive integer (=true) if the elements in the *lhs* are all less than or equal to the corresponding elements in *rhs*. Returns `0` (=false) if the dimensions of the matrices don't match.

Time complexity: O(nm), the size of the matrices.

7.3.8.5. `igraph_matrix_all_ge` — Are all elements greater or equal?

```
igraph_bool_t
igraph_matrix_all_ge(const igraph_matrix_t *lhs,
          const igraph_matrix_t *rhs);
```

Arguments:

lhs:

The first matrix.

rhs:

The second matrix.

Returns:

Positive integer (=true) if the elements in the *lhs* are all greater than or equal to the corresponding elements in *rhs*. Returns `0` (=false) if the dimensions of the matrices don't match.

Time complexity: O(nm), the size of the matrices.

7.3.9. Combining matrices

7.3.9.1. `igraph_matrix_rbind` — Combine two matrices rowwise.

```
int igraph_matrix_rbind(igraph_matrix_t *to,
      const igraph_matrix_t *from);
```

This function places the rows of *from* below the rows of `to` and stores the result in *to*. The number of columns in the two matrices must match.

Arguments:

to:

The upper matrix; the result is also stored here.

from:

The lower matrix. It is left unchanged.

Returns:

Error code.

Time complexity: O(mn), the number of elements in the newly created matrix.

7.3.9.2. `igraph_matrix_cbind` — Combine matrices columnwise.

```
int igraph_matrix_cbind(igraph_matrix_t *to,
      const igraph_matrix_t *from);
```

This function places the columns of *from* on the right of *to*, and stores the result in *to*.

Arguments:

to:

The left matrix; the result is stored here too.

from:

The right matrix. It is left unchanged.

Returns:

Error code.

Time complexity: O(mn), the number of elements on the new matrix.

7.3.10. Finding minimum and maximum

7.3.10.1. igraph_matrix_min — Minimum element.

```
igraph_real_t igraph_matrix_min(const igraph_matrix_t *m);
```

Returns the smallest element of a non-empty matrix.

Arguments:

m:

The input matrix.

Returns:

The smallest element.

Time complexity: O(mn), the number of elements.

7.3.10.2. `igraph_matrix_max` — Returns the maximal element of a matrix.

```
igraph_real_t igraph_matrix_max(const igraph_matrix_t *m);
```

Arguments:

m:

The matrix object.

Returns:

The maximum element. For empty matrix the returned value is undefined.

Added in version 0.2.

Time complexity: O(n), the number of elements in the matrix.

7.3.10.3. `igraph_matrix_which_min` — Indices of the minimum.

```
int igraph_matrix_which_min(const igraph_matrix_t *m,
          long int *i, long int *j);
```

Gives the indices of the (first) smallest element in a non-empty matrix.

Arguments:

m:

The matrix.

i:

Pointer to a long int. The row index of the minimum is stored here.

j:

Pointer to a long int. The column index of the minimum is stored here.

Returns:

Error code.

Time complexity: O(mn), the number of elements.

7.3.10.4. `igraph_matrix_which_max` — Indices of the maximum.

```
int igraph_matrix_which_max(const igraph_matrix_t *m,
          long int *i, long int *j);
```

Gives the indices of the (first) largest element in a non-empty matrix.

Arguments:

m:

The matrix.

i:

Pointer to a long int. The row index of the maximum is stored here.

j:

Pointer to a long int. The column index of the maximum is stored here.

Returns:

Error code.

Time complexity: O(mn), the number of elements.

7.3.10.5. `igraph_matrix_minmax` — Minimum and maximum

```
int igraph_matrix_minmax(const igraph_matrix_t *m,
       igraph_real_t *min, igraph_real_t *max);
```

The maximum and minimum elements of a non-empty matrix.

Arguments:

m:

The input matrix.

min:

Pointer to a base type. The minimum is stored here.

max:

Pointer to a base type. The maximum is stored here.

Returns:

Error code.

Time complexity: O(mn), the number of elements.

7.3.10.6. `igraph_matrix_which_minmax` — Indices of the minimum and maximum

```
int igraph_matrix_which_minmax(const igraph_matrix_t *m,
      long int *imin, long int *jmin,
      long int *imax, long int *jmax);
```

Find the positions of the smallest and largest elements of a non-empty matrix.

Arguments:

m:

The input matrix.

imin:

Pointer to a long int, the row index of the minimum is stored here.

jmin:

Pointer to a long int, the column index of the minimum is stored here.

imax:

Pointer to a long int, the row index of the maximum is stored here.

jmax:

Pointer to a long int, the column index of the maximum is stored here.

Returns:

Error code.

Time complexity: O(mn), the number of elements.

7.3.11. Matrix properties

7.3.11.1. `igraph_matrix_empty` — Check for an empty matrix.

```
igraph_bool_t igraph_matrix_empty(const igraph_matrix_t *m);
```

It is possible to have a matrix with zero rows or zero columns, or even both. This functions checks for these.

Arguments:

m:

The input matrix.

Returns:

Boolean, `TRUE` if the matrix contains zero elements, and `FALSE` otherwise.

Time complexity: O(1).

7.3.11.2. `igraph_matrix_isnull` — Check for a null matrix.

```
igraph_bool_t igraph_matrix_isnull(const igraph_matrix_t *m);
```

Checks whether all elements are zero.

Arguments:

m:

The input matrix.

Returns:

Boolean, `TRUE` is *m* contains only zeros and `FALSE` otherwise.

Time complexity: O(mn), the number of elements.

7.3.11.3. `igraph_matrix_size` — The number of elements in a matrix.

```
long int igraph_matrix_size(const igraph_matrix_t *m);
```

Arguments:

m:

Pointer to an initialized matrix object.

Returns:

The size of the matrix.

Time complexity: O(1).

7.3.11.4. `igraph_matrix_capacity` — Returns the number of elements allocated for a matrix.

```
long int igraph_matrix_capacity(const igraph_matrix_t *m);
```

Note that this might be different from the size of the matrix (as queried by `igraph_matrix_size()`, and specifies how many elements the matrix can hold, without reallocation.

Arguments:

v:

Pointer to the (previously initialized) matrix object to query.

Returns:

The allocated capacity.

See also:

`igraph_matrix_size()`, `igraph_matrix_nrow()`, `igraph_matrix_ncol()`.

Time complexity: O(1).

7.3.11.5. `igraph_matrix_nrow` — The number of rows in a matrix.

```
long int igraph_matrix_nrow(const igraph_matrix_t *m);
```

Arguments:

m:

Pointer to an initialized matrix object.

Returns:

The number of rows in the matrix.

Time complexity: O(1).

7.3.11.6. `igraph_matrix_ncol` — The number of columns in a matrix.

```
long int igraph_matrix_ncol(const igraph_matrix_t *m);
```

Arguments:

m:

Pointer to an initialized matrix object.

Returns:

The number of columns in the matrix.

Time complexity: O(1).

7.3.11.7. `igraph_matrix_is_symmetric` — Check for symmetric matrix.

```
igraph_bool_t igraph_matrix_is_symmetric(const igraph_matrix_t *m);
```

A non-square matrix is not symmetric by definition.

Arguments:

m:

The input matrix.

Returns:

Boolean, `TRUE` if the matrix is square and symmetric, `FALSE` otherwise.

Time complexity: O(mn), the number of elements. O(1) for non-square matrices.

7.3.11.8. `igraph_matrix_maxdifference` — Maximum absolute difference between two matrices.

```
igraph_real_t igraph_matrix_maxdifference(const igraph_matrix_t *m1,
        const igraph_matrix_t *m2);
```

Calculate the maximum absolute difference of two matrices. Both matrices must be non-empty. If their dimensions differ then a warning is given and the comparison is performed by vectors columnwise from both matrices. The remaining elements in the larger vector are ignored.

Arguments:

m1:

The first matrix.

m2:

The second matrix.

Returns:

The element with the largest absolute value in `m1` - `m2`.

Time complexity: O(mn), the elements in the smaller matrix.

7.3.12. Searching for elements

7.3.12.1. `igraph_matrix_contains` — Search for an element.

```
igraph_bool_t igraph_matrix_contains(const igraph_matrix_t *m,
            igraph_real_t e);
```

Search for the given element in the matrix.

Arguments:

m:

The input matrix.

e:

The element to search for.

Returns:

Boolean, `TRUE` if the matrix contains *e*, `FALSE` otherwise.

Time complexity: O(mn), the number of elements.

7.3.12.2. `igraph_matrix_search` — Search from a given position.

```
igraph_bool_t igraph_matrix_search(const igraph_matrix_t *m,
          long int from, igraph_real_t what,
          long int *pos,
          long int *row, long int *col);
```

Search for an element in a matrix and start the search from the given position. The search is performed columnwise.

Arguments:

m:

The input matrix.

from:

The position to search from, the positions are enumerated columnwise.

what:

The element to search for.

pos:

Pointer to a long int. If the element is found, then this is set to the position of its first appearance.

row:

Pointer to a long int. If the element is found, then this is set to its row index.

col:

Pointer to a long int. If the element is found, then this is set to its column index.

Returns:

Boolean, `TRUE` if the element is found, `FALSE` otherwise.

Time complexity: O(mn), the number of elements.

7.3.13. Resizing operations

7.3.13.1. `igraph_matrix_resize` — Resizes a matrix.

```
int igraph_matrix_resize(igraph_matrix_t *m, long int nrow, long int ncol);
```

This function resizes a matrix by adding more elements to it. The matrix contains arbitrary data after resizing it. That is, after calling this function you cannot expect that element (i,j) in the matrix remains the same as before.

Arguments:

m:

Pointer to an already initialized matrix object.

nrow:

The number of rows in the resized matrix.

ncol:

The number of columns in the resized matrix.

Returns:

Error code.

Time complexity: O(1) if the matrix gets smaller, usually O(n) if it gets larger, n is the number of elements in the resized matrix.

7.3.13.2. `igraph_matrix_resize_min` — Deallocates unused memory for a matrix.

```
int igraph_matrix_resize_min(igraph_matrix_t *m);
```

Note that this function might fail if there is not enough memory available.

Also note, that this function leaves the matrix intact, i.e. it does not destroy any of the elements. However, usually it involves copying the matrix in memory.

Arguments:

m:

Pointer to an initialized matrix.

Returns:

Error code.

See also:

`igraph_matrix_resize()`.

Time complexity: operating system dependent.

7.3.13.3. `igraph_matrix_add_rows` — Adds rows to a matrix.

```
int igraph_matrix_add_rows(igraph_matrix_t *m, long int n);
```

Arguments:

m:

The matrix object.

n:

The number of rows to add.

Returns:

Error code, `IGRAPH_ENOMEM` if there isn't enough memory for the operation.

Time complexity: linear with the number of elements of the new, resized matrix.

7.3.13.4. `igraph_matrix_add_cols` — Adds columns to a matrix.

```
int igraph_matrix_add_cols(igraph_matrix_t *m, long int n);
```

Arguments:

m:

The matrix object.

n:

The number of columns to add.

Returns:

Error code, `IGRAPH_ENOMEM` if there is not enough memory to perform the operation.

Time complexity: linear with the number of elements of the new, resized matrix.

7.3.13.5. `igraph_matrix_remove_row` — Remove a row.

```
int igraph_matrix_remove_row(igraph_matrix_t *m, long int row);
```

A row is removed from the matrix.

Arguments:

m:

The input matrix.

row:

The index of the row to remove.

Returns:

Error code.

Time complexity: O(mn), the number of elements in the matrix.

7.3.13.6. `igraph_matrix_remove_col` — Removes a column from a matrix.

```
int igraph_matrix_remove_col(igraph_matrix_t *m, long int col);
```

Arguments:

m:

The matrix object.

col:

The column to remove.

Returns:

Error code, always returns with success.

Time complexity: linear with the number of elements of the new, resized matrix.

7.4. Sparse matrices

7.4.1. About igraph_spmatrix_t objects

The igraph_spmatrix_t type stores a sparse matrix with the assumption that the number of nonzero elements in the matrix scales linearly with the row or column count of the matrix (so most of the elements are zero). Of course it can store an arbitrary real matrix, but if most of the elements are nonzero, one should use igraph_matrix_t instead.

The elements are stored in column compressed format, so the elements in the same column are stored adjacent in the computer's memory. The storage requirement for a sparse matrix is O(n) where n is the number of nonzero elements. Actually it can be a bit larger, see the documentation of the vector type for an explanation.

7.4.2. Sparse matrix constructors and destructors.

7.4.2.1. `igraph_spmatrix_init` — Initializes a sparse matrix.

```
int igraph_spmatrix_init(igraph_spmatrix_t *m, long int nrow, long int ncol);
```

Every sparse matrix needs to be initialized before using it, this is done by calling this function. A matrix has to be destroyed if it is not needed any more, see `igraph_spmatrix_destroy()`.

Arguments:

m:

Pointer to a not yet initialized sparse matrix object to be initialized.

nrow:

The number of rows in the matrix.

ncol:

The number of columns in the matrix.

Returns:

Error code.

Time complexity: operating system dependent.

7.4.2.2. `igraph_spmatrix_copy` — Copies a sparse matrix.

```
int igraph_spmatrix_copy(igraph_spmatrix_t *to, const igraph_spmatrix_t *from);
```

Creates a sparse matrix object by copying another one.

Arguments:

to:

Pointer to an uninitialized sparse matrix object.

from:

The initialized sparse matrix object to copy.

Returns:

Error code, `IGRAPH_ENOMEM` if there isn't enough memory to allocate the new sparse matrix.

Time complexity: O(n), the number of elements in the matrix.

7.4.2.3. `igraph_spmatrix_destroy` — Destroys a sparse matrix object.

```
void igraph_spmatrix_destroy(igraph_spmatrix_t *m);
```

This function frees all the memory allocated for a sparse matrix object. The destroyed object needs to be reinitialized before using it again.

Arguments:

m:

The matrix to destroy.

Time complexity: operating system dependent.

7.4.3. Accessing elements of a sparse matrix

7.4.3.1. `igraph_spmatrix_e` — Accessing an element of a sparse matrix.

```
igraph_real_t igraph_spmatrix_e(const igraph_spmatrix_t *m,
                                long int row, long int col);
```

Note that there are no range checks right now.

Arguments:

m:

The matrix object.

row:

The index of the row, starting with zero.

col:

The index of the column, starting with zero.

Time complexity: O(log n), where n is the number of nonzero elements in the requested column.

7.4.3.2. `igraph_spmatrix_set` — Setting an element of a sparse matrix.

```
int igraph_spmatrix_set(igraph_spmatrix_t *m, long int row, long int col,
            igraph_real_t value);
```

Note that there are no range checks right now.

Arguments:

m:

The matrix object.

row:

The index of the row, starting with zero.

col:

The index of the column, starting with zero.

value:

The new value.

Time complexity: O(log n), where n is the number of nonzero elements in the requested column.

7.4.3.3. `igraph_spmatrix_add_e` — Adding a real value to an element of a sparse matrix.

```
int igraph_spmatrix_add_e(igraph_spmatrix_t *m, long int row, long int col,
            igraph_real_t value);
```

Note that there are no range checks right now. This is implemented to avoid double lookup of a given element in the matrix by using `igraph_spmatrix_e()` and `igraph_spmatrix_set()` consecutively.

Arguments:

m:

The matrix object.

row:

The index of the row, starting with zero.

col:

The index of the column, starting with zero.

value:

The value to add.

Time complexity: O(log n), where n is the number of nonzero elements in the requested column.

7.4.4. Iterating over the non-zero elements of a sparse matrix

The igraph_spmatrix_iter_t type represents an iterator that can be used to step over the non-zero elements of a sparse matrix in columnwise order efficiently. In general, you shouldn't modify the elements of the matrix while iterating over it; doing so will probably invalidate the iterator, but there are no checks to prevent you from doing this.

To access the row index of the current element of the iterator, use its `ri` field. Similarly, the `ci` field stores the column index of the current element and the `value` field stores the value of the element.

7.4.4.1. `igraph_spmatrix_iter_create` — Creates a sparse matrix iterator corresponding to the given matrix.

```
int igraph_spmatrix_iter_create(igraph_spmatrix_iter_t *mit, const igraph_spmatrix_t *m);
```

Arguments:

mit:

pointer to the matrix iterator being initialized

m:

pointer to the matrix we will be iterating over

Returns:

Error code. The current implementation is always successful.

Time complexity: O(1).

7.4.4.2. `igraph_spmatrix_iter_reset` — Resets a sparse matrix iterator.

```
int igraph_spmatrix_iter_reset(igraph_spmatrix_iter_t *mit);
```

After resetting, the iterator will point to the first nonzero element (if any).

Arguments:

mit:

pointer to the matrix iterator being reset

Returns:

Error code. The current implementation is always successful.

Time complexity: O(1).

7.4.4.3. `igraph_spmatrix_iter_next` — Moves a sparse matrix iterator to the next nonzero element.

```
int igraph_spmatrix_iter_next(igraph_spmatrix_iter_t *mit);
```

You should call this function only if `igraph_spmatrix_iter_end()` returns FALSE (0).

Arguments:

mit:

pointer to the matrix iterator being moved

Returns:

Error code. The current implementation is always successful.

Time complexity: O(1).

7.4.4.4. `igraph_spmatrix_iter_end` — Checks whether there are more elements in the iterator.

```
igraph_bool_t igraph_spmatrix_iter_end(igraph_spmatrix_iter_t *mit);
```

You should call this function before calling `igraph_spmatrix_iter_next()` to make sure you have more elements in the iterator.

Arguments:

mit:

pointer to the matrix iterator being checked

Returns:

TRUE (1) if there are more elements in the iterator, FALSE (0) otherwise.

Time complexity: O(1).

7.4.4.5. `igraph_spmatrix_iter_destroy` — Frees the memory used by the iterator.

```
void igraph_spmatrix_iter_destroy(igraph_spmatrix_iter_t *mit);
```

The current implementation does not allocate any memory upon creation, so this function does nothing. However, since there is no guarantee that future implementations will not allocate any memory in `igraph_spmatrix_iter_create()`, you are still required to call this function whenever you are done with the iterator.

Arguments:

mit:

pointer to the matrix iterator being destroyed

Time complexity: O(1).

7.4.5. Matrix query operations

7.4.5.1. `igraph_spmatrix_size` — The number of elements in a sparse matrix.

```
long int igraph_spmatrix_size(const igraph_spmatrix_t *m);
```

Arguments:

m:

Pointer to an initialized sparse matrix object.

Returns:

The size of the matrix.

Time complexity: O(1).

7.4.5.2. `igraph_spmatrix_nrow` — The number of rows in a sparse matrix.

```
long int igraph_spmatrix_nrow(const igraph_spmatrix_t *m);
```

Arguments:

m:

Pointer to an initialized sparse matrix object.

Returns:

The number of rows in the matrix.

Time complexity: O(1).

7.4.5.3. `igraph_spmatrix_ncol` — The number of columns in a sparse matrix.

```
long int igraph_spmatrix_ncol(const igraph_spmatrix_t *m);
```

Arguments:

m:

Pointer to an initialized sparse matrix object.

Returns:

The number of columns in the sparse matrix.

Time complexity: O(1).

7.4.5.4. `igraph_spmatrix_count_nonzero` — The number of non-zero elements in a sparse matrix.

```
long int igraph_spmatrix_count_nonzero(const igraph_spmatrix_t *m);
```

Arguments:

m:

Pointer to an initialized sparse matrix object.

Returns:

The size of the matrix.

Time complexity: O(1).

7.4.5.5. `igraph_spmatrix_max` — Returns the maximum element of a matrix.

```
igraph_real_t igraph_spmatrix_max(const igraph_spmatrix_t *m,
    igraph_real_t *ridx, igraph_real_t *cidx);
```

If the matrix is empty, zero is returned.

Arguments:

m:

the matrix object.

ridx:

the row index of the maximum element if not `NULL`.

cidx:

the column index of the maximum element if not `NULL`.

Time complexity: O(n), the number of nonzero elements in the matrix.

7.4.5.6. `igraph_spmatrix_rowsums` — Calculates the row sums of the matrix.

```
int igraph_spmatrix_rowsums(const igraph_spmatrix_t *m, igraph_vector_t *res);
```

Arguments:

m:

The matrix.

res:

An initialized `igraph_vector_t`, the result will be stored here. The vector will be resized as needed.

Time complexity: O(n), the number of nonzero elements in the matrix.

7.4.5.7. `igraph_spmatrix_colsums` — Calculates the column sums of the matrix.

```
int igraph_spmatrix_colsums(const igraph_spmatrix_t *m, igraph_vector_t *res);
```

Arguments:

m:

The matrix.

res:

An initialized `igraph_vector_t`, the result will be stored here. The vector will be resized as needed.

Time complexity: O(n), the number of nonzero elements in the matrix.

7.4.6. Matrix operations

7.4.6.1. `igraph_spmatrix_scale` — Multiplies each element of the sparse matrix by a constant.

```
void igraph_spmatrix_scale(igraph_spmatrix_t *m, igraph_real_t by);
```

Arguments:

m:

The matrix.

by:

The constant.

Time complexity: O(n), the number of elements in the matrix.

7.4.6.2. `igraph_spmatrix_add_rows` — Adds rows to a sparse matrix.

```
int igraph_spmatrix_add_rows(igraph_spmatrix_t *m, long int n);
```

Arguments:

m:

The sparse matrix object.

n:

The number of rows to add.

Returns:

Error code.

Time complexity: O(1).

7.4.6.3. `igraph_spmatrix_add_cols` — Adds columns to a sparse matrix.

```
int igraph_spmatrix_add_cols(igraph_spmatrix_t *m, long int n);
```

Arguments:

m:

The sparse matrix object.

n:

The number of columns to add.

Returns:

Error code.

Time complexity: O(1).

7.4.6.4. `igraph_spmatrix_resize` — Resizes a sparse matrix.

```
int igraph_spmatrix_resize(igraph_spmatrix_t *m, long int nrow, long int ncol);
```

This function resizes a sparse matrix by adding more elements to it. The matrix retains its data even after resizing it, except for the data which lies outside the new boundaries (if the new size is smaller).

Arguments:

`m`:

Pointer to an already initialized sparse matrix object.

`nrow`:

The number of rows in the resized matrix.

`ncol`:

The number of columns in the resized matrix.

Returns:

Error code.

Time complexity: O(n). n is the number of elements in the old matrix.

7.4.7. Printing sparse matrices

7.4.7.1. `igraph_spmatrix_print` — Prints a sparse matrix.

```
int igraph_spmatrix_print(const igraph_spmatrix_t* matrix);
```

Prints a sparse matrix to the standard output. Only the non-zero entries are printed.

Returns:

Error code.

Time complexity: O(n), the number of non-zero elements.

7.4.7.2. `igraph_spmatrix_fprint` — Prints a sparse matrix to the given file.

```
int igraph_spmatrix_fprint(const igraph_spmatrix_t* matrix, FILE *file);
```

Prints a sparse matrix to the given file. Only the non-zero entries are printed.

Returns:

Error code.

Time complexity: O(n), the number of non-zero elements.

7.5. Sparse matrices, another kind

7.5.1. About sparse matrices

The `igraph_sparsemat_t` data type stores sparse matrices, i.e. matrices in which the majority of the elements are zero.

The data type is essentially a wrapper to some of the functions in the CXSparse library, by Tim Davis, see http://www.cise.ufl.edu/research/sparse/CXSparse/

Matrices can be stored in two formats: triplet and column-compressed. The triplet format is intended for sparse matrix initialization, as it is easy to add new (non-zero) elements to it. Most of the computations are done on sparse matrices in column-compressed format, after the user has converted the triplet matrix to column-compressed, via `igraph_sparsemat_compress()`.

Both formats are dynamic, in the sense that new elements can be added to them, possibly resulting the allocation of more memory.

Row and column indices follow the C convention and are zero-based.

Example 7-2. File examples/simple/igraph_sparsemat.c

Example 7-3. File examples/simple/igraph_sparsemat2.c

Example 7-4. File examples/simple/igraph_sparsemat3.c

Example 7-5. File examples/simple/igraph_sparsemat4.c

Example 7-6. File examples/simple/igraph_sparsemat5.c

Example 7-7. File examples/simple/igraph_sparsemat6.c

Example 7-8. File examples/simple/igraph_sparsemat7.c

Example 7-9. File examples/simple/igraph_sparsemat8.c

7.5.2. Creating sparse matrix objects

7.5.2.1. `igraph_sparsemat_init` — Initialize a sparse matrix, in triplet format

```
int igraph_sparsemat_init(igraph_sparsemat_t *A, int rows, int cols, int nzmax);
```

This is the most common way to create a sparse matrix, together with the `igraph_sparsemat_entry()` function, which can be used to add the non-zero elements one by one. Once done, the user can call `igraph_sparsemat_compress()` to convert the matrix to column-compressed, to allow computations with it.

The user must call `igraph_sparsemat_destroy()` on the matrix to deallocate the memory, once the matrix is no more needed.

Arguments:

`A`:

Pointer to a not yet initialized sparse matrix.

`rows`:

The number of rows in the matrix.

`cols`:

The number of columns.

`nzmax`:

The maximum number of non-zero elements in the matrix. It is not compulsory to get this right, but it is useful for the allocation of the proper amount of memory.

Returns:

Error code.

Time complexity: TODO.

7.5.2.2. `igraph_sparsemat_copy` — Copy a sparse matrix

```
int igraph_sparsemat_copy(igraph_sparsemat_t *to,
     const igraph_sparsemat_t *from);
```

Create a sparse matrix object, by copying another one. The source matrix can be either in triplet or column-compressed format.

Exactly the same amount of memory will be allocated to the copy matrix, as it is currently for the original one.

Arguments:

`to`:

Pointer to an uninitialized sparse matrix, the copy will be created here.

`from`:

The sparse matrix to copy.

Returns:

Error code.

Time complexity: O(n+nzmax), the number of columns plus the maximum number of non-zero elements.

7.5.2.3. `igraph_sparsemat_realloc` — Allocate more (or less) memory for a sparse matrix

```
int igraph_sparsemat_realloc(igraph_sparsemat_t *A, int nzmax);
```

Sparse matrices automatically allocate more memory, as needed. To control memory allocation, the user can call this function, to allocate memory for a given number of non-zero elements.

Arguments:

A:

The sparse matrix, it can be in triplet or column-compressed format.

nzmax:

The new maximum number of non-zero elements.

Returns:

Error code.

Time complexity: TODO.

7.5.2.4. `igraph_sparsemat_destroy` — Deallocate memory used by a sparse matrix

```
void igraph_sparsemat_destroy(igraph_sparsemat_t *A);
```

One destroyed, the sparse matrix must be initialized again, before calling any other operation on it.

Arguments:

A:

The sparse matrix to destroy.

Time complexity: O(1).

7.5.2.5. `igraph_sparsemat_eye` — Create a sparse identity matrix

```
int igraph_sparsemat_eye(igraph_sparsemat_t *A, int n, int nzmax,
    igraph_real_t value,
    igraph_bool_t compress);
```

Arguments:

A:

An uninitialized sparse matrix, the result is stored here.

n:

The number of rows and number of columns in the matrix.

nzmax:

The maximum number of non-zero elements, this essentially gives the amount of memory that will be allocated for matrix elements.

value:

The value to store in the diagonal.

compress:

Whether to create a column-compressed matrix. If false, then a triplet matrix is created.

Returns:

Error code.

Time complexity: O(n).

7.5.2.6. `igraph_sparsemat_diag` — Create a sparse diagonal matrix

```
int igraph_sparsemat_diag(igraph_sparsemat_t *A, int nzmax,
     const igraph_vector_t *values,
     igraph_bool_t compress);
```

Arguments:

A:

An uninitialized sparse matrix, the result is stored here.

nzmax:

The maximum number of non-zero elements, this essentially gives the amount of memory that will be allocated for matrix elements.

values:

The values to store in the diagonal, the size of the matrix defined by the length of this vector.

compress:

Whether to create a column-compressed matrix. If false, then a triplet matrix is created.

Returns:

Error code.

Time complexity: O(n), the length of the diagonal vector.

7.5.3. Query properties of a sparse matrix

7.5.3.1. `igraph_sparsemat_index` — Index a sparse matrix, extract a submatrix, or a single element

```
int igraph_sparsemat_index(const igraph_sparsemat_t *A,
      const igraph_vector_int_t *p,
      const igraph_vector_int_t *q,
      igraph_sparsemat_t *res,
      igraph_real_t *constres);
```

This function serves two purposes. First, it can extract submatrices from a sparse matrix. Second, as a special case, it can extract a single element from a sparse matrix.

Arguments:

A:

The input matrix, it must be in column-compressed format.

p:

An integer vector, or a null pointer. The selected row index or indices. A null pointer selects all rows.

q:

An integer vector, or a null pointer. The selected column index or indices. A null pointer selects all columns.

res:

Pointer to an uninitialized sparse matrix, or a null pointer. If not a null pointer, then the selected submatrix is stored here.

constres:

Pointer to a real variable or a null pointer. If not a null pointer, then the first non-zero element in the selected submatrix is stored here, if there is one. Otherwise zero is stored here. This behavior is handy if one wants to select a single entry from the matrix.

Returns:

Error code.

Time complexity: TODO.

7.5.3.2. `igraph_sparsemat_nrow` — Number of rows

```
long int igraph_sparsemat_nrow(const igraph_sparsemat_t *A);
```

Arguments:

A:

The input matrix, in triplet or column-compressed format.

Returns:

The number of rows in the *A* matrix.

Time complexity: O(1).

7.5.3.3. `igraph_sparsemat_ncol` — Number of columns.

```
long int igraph_sparsemat_ncol(const igraph_sparsemat_t *A);
```

Arguments:

A:

The input matrix, in triplet or column-compressed format.

Returns:

The number of columns in the `A` matrix.

Time complexity: O(1).

7.5.3.4. `igraph_sparsemat_type` — Type of a sparse matrix (triplet or column-compressed)

```
igraph_sparsemat_type_t igraph_sparsemat_type(const igraph_sparsemat_t *A);
```

Gives whether a sparse matrix is stored in the triplet format or in column-compressed format.

Arguments:

A:

The input matrix.

Returns:

Either `IGRAPH_SPARSEMAT_CC` or `IGRAPH_SPARSEMAT_TRIPLET`.

Time complexity: O(1).

7.5.3.5. `igraph_sparsemat_is_triplet` — Is this sparse matrix in triplet format?

```
igraph_bool_t igraph_sparsemat_is_triplet(const igraph_sparsemat_t *A);
```

Decides whether a sparse matrix is in triplet format.

Arguments:

A:

The input matrix.

Returns:

One if the input matrix is in triplet format, zero otherwise.

Time complexity: O(1).

7.5.3.6. `igraph_sparsemat_is_cc` — Is this sparse matrix in column-compressed format?

```
igraph_bool_t igraph_sparsemat_is_cc(const igraph_sparsemat_t *A);
```

Decides whether a sparse matrix is in column-compressed format.

Arguments:

A:

The input matrix.

Returns:

One if the input matrix is in column-compressed format, zero otherwise.

Time complexity: O(1).

7.5.4. Operations on sprase matrices

7.5.4.1. `igraph_sparsemat_entry` — Add an element to a sparse matrix

```
int igraph_sparsemat_entry(igraph_sparsemat_t *A, int row, int col,
      igraph_real_t elem);
```

This function can be used to add the entries to a sparse matrix, after initializing it with `igraph_sparsemat_init()`.

Arguments:

`A`:

The input matrix, it must be in triplet format.

`row`:

The row index of the entry to add.

`col`:

The column index of the entry to add.

`elem`:

The value of the entry.

Returns:

Error code.

Time complexity: TODO.

7.5.4.2. `igraph_sparsemat_fkeep` — Filter the elements of a sparse matrix

```
int igraph_sparsemat_fkeep(igraph_sparsemat_t *A,
      int (*fkeep)(int, int, igraph_real_t, void*),
```

```
void *other);
```

This function can be used to filter the (non-zero) elements of a sparse matrix. For all entries, it calls the supplied function and depending on the return values either keeps, or deleted the element from the matrix.

Arguments:

A:

The input matrix, in column-compressed format.

fkeep:

The filter function. It must take four arguments: the first is an `int`, the row index of the entry, the second is another `int`, the column index. The third is `igraph_real_t`, the value of the entry. The fourth element is a `void` pointer, the *other* argument is passed here. The function must return an `int`. If this is zero, then the entry is deleted, otherwise it is kept.

other:

A `void` pointer that is passed to the filtering function.

Returns:

Error code.

Time complexity: TODO.

7.5.4.3. `igraph_sparsemat_dropzeros` — Drop the zero elements from a sparse matrix

```
int igraph_sparsemat_dropzeros(igraph_sparsemat_t *A);
```

As a result of matrix operations, some of the entries in a sparse matrix might be zero. This function removes these entries.

Arguments:

A:

The input matrix, it must be in column-compressed format.

Returns:

Error code.

Time complexity: TODO.

7.5.4.4. `igraph_sparsemat_droptol` — Drop the almost zero elements of a sparse matrix

```
int igraph_sparsemat_droptol(igraph_sparsemat_t *A, igraph_real_t tol);
```

This function is similar to `igraph_sparsemat_dropzeros()`, but it also drops entries that are closer to zero than the given tolerance threshold.

Arguments:

A:

The input matrix, it must be in column-compressed format.

tol:

Real number, giving the tolerance threshold.

Returns:

Error code.

Time complexity: TODO.

7.5.4.5. `igraph_sparsemat_scale` — Scale a sparse matrix

```
int igraph_sparsemat_scale(igraph_sparsemat_t *A, igraph_real_t by);
```

Multiplies all elements of a sparse matrix, by the given scalar.

Arguments:

A:

The input matrix.

by:

The scaling factor.

Returns:

Error code.

Time complexity: O(nz), the number of non-zero elements in the matrix.

7.5.4.6. `igraph_sparsemat_permute` — Permute the rows and columns of a sparse matrix

```
int igraph_sparsemat_permute(const igraph_sparsemat_t *A,
        const igraph_vector_int_t *p,
        const igraph_vector_int_t *q,
        igraph_sparsemat_t *res);
```

Arguments:

A:

The input matrix, it must be in column-compressed format.

p:

Integer vector, giving the permutation of the rows.

q:

Integer vector, the permutation of the columns.

res:

Pointer to an uninitialized sparse matrix, the result is stored here.

Returns:

Error code.

Time complexity: O(m+n+nz), the number of rows plus the number of columns plus the number of non-zero elements in the matrix.

7.5.4.7. `igraph_sparsemat_transpose` — Transpose a sparse matrix

```
int igraph_sparsemat_transpose(const igraph_sparsemat_t *A,
          igraph_sparsemat_t *res,
          int values);
```

Arguments:

A:

The input matrix, column-compressed or triple format.

res:

Pointer to an uninitialized sparse matrix, the result is stored here.

values:

If this is non-zero, the matrix transpose is calculated the normal way. If it is zero, then only the pattern of the input matrix is stored in the result, the values are not.

Returns:

Error code.

Time complexity: TODO.

7.5.4.8. `igraph_sparsemat_add` — Sum of two sparse matrices

```
int igraph_sparsemat_add(const igraph_sparsemat_t *A,
    const igraph_sparsemat_t *B,
    igraph_real_t alpha,
    igraph_real_t beta,
    igraph_sparsemat_t *res);
```

Arguments:

A:

The first input matrix, in column-compressed format.

B:

The second input matrix, in column-compressed format.

alpha:

Real scalar, *A* is multiplied by *alpha* before the addition.

beta:

Real scalar, *B* is multiplied by *beta* before the addition.

res:

Pointer to an uninitialized sparse matrix, the result is stored here.

Returns:

Error code.

Time complexity: TODO.

7.5.4.9. `igraph_sparsemat_multiply` — Matrix multiplication

```
int igraph_sparsemat_multiply(const igraph_sparsemat_t *A,
         const igraph_sparsemat_t *B,
         igraph_sparsemat_t *res);
```

Multiplies two sparse matrices.

Arguments:

A:

The first input matrix (left hand side), in column-compressed format.

B:

The second input matrix (right hand side), in column-compressed format.

res:

Pointer to an uninitialized sparse matrix, the result is stored here.

Returns:

Error code.

Time complexity: TODO.

7.5.4.10. `igraph_sparsemat_gaxpy` — Matrix-vector product, added to another vector.

```
int igraph_sparsemat_gaxpy(const igraph_sparsemat_t *A,
      const igraph_vector_t *x,
      igraph_vector_t *res);
```

Arguments:

A:

The input matrix, in column-compressed format.

x:

The input vector, its size must match the number of columns in *A*.

res:

This vector is added to the matrix-vector product and it is overwritten by the result.

Returns:

Error code.

Time complexity: TODO.

7.5.4.11. `igraph_sparsemat_add_rows` — Add rows to a sparse matrix

```
int igraph_sparsemat_add_rows(igraph_sparsemat_t *A, long int n);
```

The current matrix elements are retained and all elements in the new rows are zero.

Arguments:

A:

The input matrix, in triplet or column-compressed format.

n:

The number of rows to add.

Returns:

Error code.

Time complexity: O(1).

7.5.4.12. `igraph_sparsemat_add_cols` — Add columns to a sparse matrix

```
int igraph_sparsemat_add_cols(igraph_sparsemat_t *A, long int n);
```

The current matrix elements are retained, and all elements in the new columns are zero.

Arguments:

A:

The input matrix, in triplet or column-compressed format.

n:

The number of columns to add.

Returns:

Error code.

Time complexity: TODO.

7.5.4.13. `igraph_sparsemat_resize` — Resize a sparse matrix

```
int igraph_sparsemat_resize(igraph_sparsemat_t *A, long int nrow,
       long int ncol, int nzmax);
```

This function resizes a sparse matrix. The resized sparse matrix will be empty.

Arguments:

A:

The initialized sparse matrix to resize.

nrow:

The new number of rows.

ncol:

The new number of columns.

nzmax:

The new maximum number of elements.

Returns:

Error code.

Time complexity: O(nzmax), the maximum number of non-zero elements.

7.5.5. Operations that change the internal representation

7.5.5.1. `igraph_sparsemat_compress` — Compress a sparse matrix, i.e. convert it to column-compress format

```
int igraph_sparsemat_compress(const igraph_sparsemat_t *A,
         igraph_sparsemat_t *res);
```

Almost all sparse matrix operations require that the matrix is in column-compressed format.

Arguments:

`A`:

The input matrix, it must be in triplet format.

`res`:

Pointer to an uninitialized sparse matrix object, the compressed version of `A` is stored here.

Returns:

Error code.

Time complexity: TODO.

7.5.5.2. `igraph_sparsemat_dupl` — Remove duplicate elements from a sparse matrix

```
int igraph_sparsemat_dupl(igraph_sparsemat_t *A);
```

It is possible that a column-compressed sparse matrix stores a single matrix entry in multiple pieces. The entry is then the sum of all its pieces. (Some functions create matrices like this.) This function eliminates the multiple pieces.

Arguments:

A:

The input matrix, in column-compressed format.

Returns:

Error code.

Time complexity: TODO.

7.5.6. Decompositions and solving linear systems

7.5.6.1. `igraph_sparsemat_symblu` — Symbolic LU decomposition

```
int igraph_sparsemat_symblu(long int order, const igraph_sparsemat_t *A,
       igraph_sparsemat_symbolic_t *dis);
```

LU decomposition of sparse matrices involves two steps, the first is calling this function, and then `igraph_sparsemat_lu()`.

Arguments:

order:

The ordering to use: 0 means natural ordering, 1 means minimum degree ordering of A+A', 2 is minimum degree ordering of A'A after removing the dense rows from A, and 3 is the minimum degree ordering of A'A.

A:

The input matrix, in column-compressed format.

dis:

The result of the symbolic analysis is stored here. Once not needed anymore, it must be destroyed by calling `igraph_sparsemat_symbolic_destroy()`.

Returns:

Error code.

Time complexity: TODO.

7.5.6.2. `igraph_sparsemat_symbqr` — Symbolic QR decomposition

```
int igraph_sparsemat_symbqr(long int order, const igraph_sparsemat_t *A,
       igraph_sparsemat_symbolic_t *dis);
```

QR decomposition of sparse matrices involves two steps, the first is calling this function, and then `igraph_sparsemat_qr()`.

Arguments:

order:

The ordering to use: 0 means natural ordering, 1 means minimum degree ordering of A+A', 2 is minimum degree ordering of A'A after removing the dense rows from A, and 3 is the minimum degree ordering of A'A.

A:

The input matrix, in column-compressed format.

dis:

The result of the symbolic analysis is stored here. Once not needed anymore, it must be destroyed by calling `igraph_sparsemat_symbolic_destroy()`.

Returns:

Error code.

Time complexity: TODO.

7.5.6.3. `igraph_sparsemat_lsolve` — Solve a lower-triangular linear system

```
int igraph_sparsemat_lsolve(const igraph_sparsemat_t *L,
       const igraph_vector_t *b,
```

```
        igraph_vector_t *res);
```

Solve the Lx=b linear equation system, where the L coefficient matrix is square and lower-triangular, with a zero-free diagonal.

Arguments:

L:

The input matrix, in column-compressed format.

b:

The right hand side of the linear system.

res:

An initialized vector, the result is stored here.

Returns:

Error code.

Time complexity: TODO.

7.5.6.4. `igraph_sparsemat_ltsolve` — Solve an upper-triangular linear system

```
int igraph_sparsemat_ltsolve(const igraph_sparsemat_t *L,
        const igraph_vector_t *b,
        igraph_vector_t *res);
```

Solve the L'x=b linear equation system, where the L matrix is square and lower-triangular, with a zero-free diagonal.

Arguments:

L:

The input matrix, in column-compressed format.

b:

The right hand side of the linear system.

res:

An initialized vector, the result is stored here.

Returns:

Error code.

Time complexity: TODO.

7.5.6.5. `igraph_sparsemat_usolve` — Solve an upper-triangular linear system

```
int igraph_sparsemat_usolve(const igraph_sparsemat_t *U,
       const igraph_vector_t *b,
       igraph_vector_t *res);
```

Solves the Ux=b upper triangular system.

Arguments:

U:

The input matrix, in column-compressed format.

b:

The right hand side of the linear system.

res:

An initialized vector, the result is stored here.

Returns:

Error code.

Time complexity: TODO.

7.5.6.6. `igraph_sparsemat_utsolve` — Solve a lower-triangular linear system

```
int igraph_sparsemat_utsolve(const igraph_sparsemat_t *U,
        const igraph_vector_t *b,
        igraph_vector_t *res);
```

This is the same as `igraph_sparsemat_usolve()`, but U'x=b is solved, where the apostrophe denotes the transpose.

Arguments:

U:

The input matrix, in column-compressed format.

b:

The right hand side of the linear system.

res:

An initialized vector, the result is stored here.

Returns:

Error code.

Time complexity: TODO.

7.5.6.7. `igraph_sparsemat_cholsol` — Solve a symmetric linear system via Cholesky decomposition

```
int igraph_sparsemat_cholsol(const igraph_sparsemat_t *A,
        const igraph_vector_t *b,
        igraph_vector_t *res,
        int order);
```

Solve Ax=b, where A is a symmetric positive definite matrix.

Arguments:

A:

The input matrix, in column-compressed format.

v:

The right hand side.

res:

An initialized vector, the result is stored here.

order:

An integer giving the ordering method to use for the factorization. Zero is the natural ordering; if it is one, then the fill-reducing minimum-degree ordering of A+A' is used.

Returns:

Error code.

Time complexity: TODO.

7.5.6.8. `igraph_sparsemat_lusol` — Solve a linear system via LU decomposition

```
int igraph_sparsemat_lusol(const igraph_sparsemat_t *A,
      const igraph_vector_t *b,
      igraph_vector_t *res,
      int order,
      igraph_real_t tol);
```

Solve Ax=b, via LU factorization of A.

Arguments:

A:

The input matrix, in column-compressed format.

b:

The right hand side of the equation.

res:

An initialized vector, the result is stored here.

order:

The ordering method to use, zero means the natural ordering, one means the fill-reducing minimum-degree ordering of A+A', two means the ordering of A'*A, after removing the dense rows from A. Three means the ordering of A'*A.

tol:

Real number, the tolerance limit to use for the numeric LU factorization.

Returns:

Error code.

Time complexity: TODO.

7.5.6.9. `igraph_sparsemat_lu` — LU decomposition of a sparse matrix

```
int igraph_sparsemat_lu(const igraph_sparsemat_t *A,
   const igraph_sparsemat_symbolic_t *dis,
   igraph_sparsemat_numeric_t *din, double tol);
```

Performs numeric sparse LU decomposition of a matrix.

Arguments:

A:

The input matrix, in column-compressed format.

dis:

The symbolic analysis for LU decomposition, coming from a call to the `igraph_sparsemat_symblu()` function.

din:

The numeric decomposition, the result is stored here. It can be used to solve linear systems with changing right hand side vectors, by calling `igraph_sparsemat_luresol()`. Once not needed any more, it must be destroyed by calling `igraph_sparsemat_symbolic_destroy()` on it.

tol:

The tolerance for the numeric LU decomposition.

Returns:

Error code.

Time complexity: TODO.

7.5.6.10. `igraph_sparsemat_qr` — QR decomposition of a sparse matrix

```
int igraph_sparsemat_qr(const igraph_sparsemat_t *A,
   const igraph_sparsemat_symbolic_t *dis,
   igraph_sparsemat_numeric_t *din);
```

Numeric QR decomposition of a sparse matrix.

Arguments:

A:

The input matrix, in column-compressed format.

dis:

The result of the symbolic QR analysis, from the function `igraph_sparsemat_symbqr()`.

din:

The result of the decomposition is stored here, it can be used to solve many linear systems with the same coefficient matrix and changing right hand sides, using the `igraph_sparsemat_qrresol()` function. Once not needed any more, one should call `igraph_sparsemat_numeric_destroy()` on it to free the allocated memory.

Returns:

Error code.

Time complexity: TODO.

7.5.6.11. `igraph_sparsemat_luresol` — Solve linear system using a precomputed LU decomposition

```
int igraph_sparsemat_luresol(const igraph_sparsemat_symbolic_t *dis,
        const igraph_sparsemat_numeric_t *din,
        const igraph_vector_t *b,
        igraph_vector_t *res);
```

Uses the LU decomposition of a matrix to solve linear systems.

Arguments:

`dis`:

The symbolic analysis of the coefficient matrix, the result of `igraph_sparsemat_symblu()`.

`din`:

The LU decomposition, the result of a call to `igraph_sparsemat_lu()`.

`b`:

A vector that defines the right hand side of the linear equation system.

`res`:

An initialized vector, the solution of the linear system is stored here.

Returns:

Error code.

Time complexity: TODO.

7.5.6.12. `igraph_sparsemat_qrresol` — Solve a linear system using a precomputed QR decomposition

```
int igraph_sparsemat_qrresol(const igraph_sparsemat_symbolic_t *dis,
        const igraph_sparsemat_numeric_t *din,
        const igraph_vector_t *b,
        igraph_vector_t *res);
```

Solves a linear system using a QR decomposition of its coefficient matrix.

Arguments:

dis:

Symbolic analysis of the coefficient matrix, the result of `igraph_sparsemat_symbqr()`.

din:

The QR decomposition of the coefficient matrix, the result of `igraph_sparsemat_qr()`.

b:

Vector, giving the right hand side of the linear equation system.

res:

An initialized vector, the solution is stored here. It is resized as needed.

Returns:

Error code.

Time complexity: TODO.

7.5.6.13. `igraph_sparsemat_symbolic_destroy` — Deallocate memory for a symbolic decomposition

```
void igraph_sparsemat_symbolic_destroy(igraph_sparsemat_symbolic_t *dis);
```

Frees the memory allocated by `igraph_sparsemat_symbqr()` or `igraph_sparsemat_symblu()`.

Arguments:

dis:

The symbolic analysis.

Time complexity: O(1).

7.5.6.14. `igraph_sparsemat_numeric_destroy` — Deallocate memory for a numeric decomposition

```
void igraph_sparsemat_numeric_destroy(igraph_sparsemat_numeric_t *din);
```

Frees the memoty allocated by `igraph_sparsemat_qr()` or `igraph_sparsemat_lu()`.

Arguments:

`din`:

The LU or QR decomposition.

Time complexity: O(1).

7.5.7. Eigenvalues and eigenvectors

7.5.7.1. `igraph_sparsemat_arpack_rssolve` — Eigenvalues and eigenvectors of a symmetric sparse matrix via ARPACK

```
int igraph_sparsemat_arpack_rssolve(const igraph_sparsemat_t *A,
        igraph_arpack_options_t *options,
        igraph_arpack_storage_t *storage,
        igraph_vector_t *values,
        igraph_matrix_t *vectors,
        igraph_sparsemat_solve_t solvemethod);
```

Arguments:

`The`:

input matrix, must be column-compressed.

`options`:

It is passed to `igraph_arpack_rssolve()`. See `igraph_arpack_options_t` for the details. If `mode` is 1, then ARPACK uses regular mode, if `mode` is 3, then shift and invert mode is used and the `sigma` structure member defines the shift.

storage:

Storage for ARPACK. See `igraph_arpack_rssolve()` and `igraph_arpack_storage_t` for details.

values:

An initialized vector or a null pointer, the eigenvalues are stored here.

vectors:

An initialised matrix, or a null pointer, the eigenvectors are stored here, in the columns.

solvemethod:

The method to solve the linear system, if `mode` is 3, i.e. the shift and invert mode is used. Possible values:

`IGRAPH_SPARSEMAT_SOLVE_LU`

The linear system is solved using LU decomposition.

`IGRAPH_SPARSEMAT_SOLVE_QR`

The linear system is solved using QR decomposition.

Returns:

Error code.

Time complexity: TODO.

7.5.7.2. `igraph_sparsemat_arpack_rnsolve` — Eigenvalues and eigenvectors of a nonsymmetric sparse matrix via ARPACK

```
int igraph_sparsemat_arpack_rnsolve(const igraph_sparsemat_t *A,
        igraph_arpack_options_t *options,
        igraph_arpack_storage_t *storage,
        igraph_matrix_t *values,
        igraph_matrix_t *vectors);
```

Eigenvalues and/or eigenvectors of a nonsymmetric sparse matrix.

Arguments:

A:

The input matrix, in column-compressed mode.

options:

ARPACK options, it is passed to `igraph_arpack_rnsolve()`. See also `igraph_arpack_options_t` for details.

storage:

Storage for ARPACK, this is passed to `igraph_arpack_rnsolve()`. See `igraph_arpack_storage_t` for details.

values:

An initialized matrix, or a null pointer. If not a null pointer, then the eigenvalues are stored here, the first column is the real part, the second column is the imaginary part.

vectors:

An initialized matrix, or a null pointer. If not a null pointer, then the eigenvectors are stored here, please see `igraph_arpack_rnsolve()` for the format.

Returns:

Error code.

Time complexity: TODO.

7.5.8. Conversion to other data types

7.5.8.1. `igraph_sparsemat` — Create an igraph graph from a sparse matrix

```
int igraph_sparsemat(igraph_t *graph, const igraph_sparsemat_t *A,
       igraph_bool_t directed);
```

One edge is created for each non-zero entry in the matrix. If you have a symmetric matrix, and want to create an undirected graph, then delete the entries in the upper diagonal first, or call `igraph_simplify()` on the result graph to eliminate the multiple edges.

Arguments:

graph:

Pointer to an uninitialized igraph_t object, the graphs is stored here.

A:

The input matrix, in triplet or column-compressed format.

directed:

Boolean scalar, whether to create a directed graph.

Returns:

Error code.

Time complexity: TODO.

7.5.8.2. `igraph_get_sparsemat` — Convert an igraph graph to a sparse matrix

```
int igraph_get_sparsemat(const igraph_t *graph, igraph_sparsemat_t *res);
```

If the graph is undirected, then a symmetric matrix is created.

Arguments:

graph:

The input graph.

res:

Pointer to an uninitialized sparse matrix. The result will be stored here.

Returns:

Error code.

Time complexity: TODO.

7.5.8.3. `igraph_matrix_as_sparsemat` — Convert a dense matrix to a sparse matrix

```
int igraph_matrix_as_sparsemat(igraph_sparsemat_t *res,
          const igraph_matrix_t *mat,
          igraph_real_t tol);
```

Arguments:

res:

An uninitialized sparse matrix, the result is stored here.

mat:

The dense input matrix.

tol:

Real scalar, the tolerance. Values closer than *tol* to zero are considered as zero, and will not be included in the sparse matrix.

Returns:

Error code.

Time complexity: O(mn), the number of elements in the dense matrix.

7.5.8.4. `igraph_sparsemat_as_matrix` — Convert a sparse matrix to a dense matrix

```
int igraph_sparsemat_as_matrix(igraph_matrix_t *res,
          const igraph_sparsemat_t *spmat);
```

Arguments:

res:

Pointer to an initialized matrix, the result is stored here. It will be resized to the required size.

spmat:

The input sparse matrix, in triplet or column-compressed format.

Returns:

Error code.

Time complexity: O(mn), the number of elements in the dense matrix.

7.5.9. Writing to a file, or to the screen

7.5.9.1. `igraph_sparsemat_print` — Print a sparse matrix to a file

```
int igraph_sparsemat_print(const igraph_sparsemat_t *A,
      FILE *outstream);
```

Only the non-zero entries are printed. This function serves more as a debugging utility, as currently there is no function that could read back the printed matrix from the file.

Arguments:

A:

The input matrix, triplet or column-compressed format.

outstream:

The stream to print it to.

Returns:

Error code.

Time complexity: O(nz) for triplet matrices, O(n+nz) for column-compressed matrices. nz is the number of non-zero elements, n is the number columns in the matrix.

7.6. Stacks

7.6.1. `igraph_stack_init` — Initializes a stack.

```
int igraph_stack_init       (igraph_stack_t* s, long int size);
```

The initialized stack is always empty.

Arguments:

`s`:

Pointer to an uninitialized stack.

`size`:

The number of elements to allocate memory for.

Returns:

Error code.

Time complexity: O(*`size`*).

7.6.2. `igraph_stack_destroy` — Destroys a stack object.

```
void igraph_stack_destroy    (igraph_stack_t* s);
```

Deallocate the memory used for a stack. It is possible to reinitialize a destroyed stack again by `igraph_stack_init()`.

Arguments:

s:

The stack to destroy.

Time complexity: O(1).

7.6.3. `igraph_stack_reserve` — Reserve memory.

```
int igraph_stack_reserve    (igraph_stack_t* s, long int size);
```

Reverse memory for future use. The actual size of the stack is unchanged.

Arguments:

s:

The stack object.

size:

The number of elements to reserve memory for. If it is not bigger than the current size then nothing happens.

Returns:

Error code.

Time complexity: should be around O(n), the new allocated size of the stack.

7.6.4. `igraph_stack_empty` — Decides whether a stack object is empty.

```
igraph_bool_t igraph_stack_empty      (igraph_stack_t* s);
```

Arguments:

s:

The stack object.

Returns:

Boolean, `TRUE` if the stack is empty, `FALSE` otherwise.

Time complexity: O(1).

7.6.5. `igraph_stack_size` — Returns the number of elements in a stack.

```
long int igraph_stack_size       (const igraph_stack_t* s);
```

Arguments:

s:

The stack object.

Returns:

The number of elements in the stack.

Time complexity: O(1).

7.6.6. `igraph_stack_clear` — Removes all elements from a stack.

```
void igraph_stack_clear      (igraph_stack_t* s);
```

Arguments:

s:

The stack object.

Time complexity: O(1).

7.6.7. `igraph_stack_push` — Places an element on the top of a stack.

```
int igraph_stack_push(igraph_stack_t* s, igraph_real_t elem);
```

The capacity of the stack is increased, if needed.

Arguments:

s:

The stack object.

elem:

The element to push.

Returns:

Error code.

Time complexity: O(1) is no reallocation is needed, O(n) otherwise, but it is ensured that n push operations are performed in O(n) time.

7.6.8. `igraph_stack_pop` — Removes and returns an element from the top of a stack.

```
igraph_real_t igraph_stack_pop        (igraph_stack_t* s);
```

The stack must contain at least one element, call `igraph_stack_empty()` to make sure of this.

Arguments:

s:

The stack object.

Returns:

The removed top element.

Time complexity: O(1).

7.6.9. `igraph_stack_top` — Query top element.

```
igraph_real_t igraph_stack_top        (const igraph_stack_t* s);
```

Returns the top element of the stack, without removing it. The stack must be non-empty.

Arguments:

s:

The stack.

Returns:

The top element.

Time complexity: O(1).

7.7. Double-ended queues

This is the classic data type of the double ended queue. Most of the time it is used if a First-In-First-Out (FIFO) behavior is needed. See the operations below.

Example 7-10. File examples/simple/dqueue.c

7.7.1. `igraph_dqueue_init` — Initialize a double ended queue (deque).

```
int igraph_dqueue_init (igraph_dqueue_t* q, long int size);
```

The queue will be always empty.

Arguments:

q:

Pointer to an uninitialized deque.

size:

How many elements to allocate memory for.

Returns:

Error code.

Time complexity: O(*size*).

7.7.2. `igraph_dqueue_destroy` — Destroy a double ended queue.

```
void igraph_dqueue_destroy (igraph_dqueue_t* q);
```

Arguments:

q:

The queue to destroy

Time complexity: O(1).

7.7.3. `igraph_dqueue_empty` — Decide whether the queue is empty.

```
igraph_bool_t igraph_dqueue_empty (const igraph_dqueue_t* q);
```

Arguments:

q:

The queue.

Returns:

Boolean, `TRUE` if *q* contains at least one element, `FALSE` otherwise.

Time complexity: O(1).

7.7.4. `igraph_dqueue_full` — Check whether the queue is full.

```
igraph_bool_t igraph_dqueue_full (igraph_dqueue_t* q);
```

If a queue is full the next igraph_dqueue_push() operation will allocate more memory.

Arguments:

q:

The queue.

Returns:

`TRUE` if *q* is full, `FALSE` otherwise.

Time complecity: O(1).

7.7.5. `igraph_dqueue_clear` — Remove all elements from the queue.

```
void igraph_dqueue_clear   (igraph_dqueue_t* q);
```

Arguments:

q:

The queue

Time complexity: O(1).

7.7.6. `igraph_dqueue_size` — Number of elements in the queue.

```
long int igraph_dqueue_size (const igraph_dqueue_t* q);
```

Arguments:

q:

The queue.

Returns:

Integer, the number of elements currently in the queue.

Time complexity: O(1).

7.7.7. `igraph_dqueue_head` — Head of the queue.

```
igraph_real_t igraph_dqueue_head (const igraph_dqueue_t* q);
```

The queue must contain at least one element.

Arguments:

q:

The queue.

Returns:

The first element in the queue.

Time complexity: O(1).

7.7.8. `igraph_dqueue_back` — Tail of the queue.

```
igraph_real_t igraph_dqueue_back (const igraph_dqueue_t* q);
```

The queue must contain at least one element.

Arguments:

q:

The queue.

Returns:

The last element in the queue.

Time complexity: O(1).

7.7.9. `igraph_dqueue_pop` — Remove the head.

```
igraph_real_t igraph_dqueue_pop (igraph_dqueue_t* q);
```

Removes and returns the first element in the queue. The queue must be non-empty.

Arguments:

q:

The input queue.

Returns:

The first element in the queue.

Time complexity: O(1).

7.7.10. `igraph_dqueue_pop_back` — Remove the tail

```
igraph_real_t igraph_dqueue_pop_back (igraph_dqueue_t* q);
```

Removes and returns the last element in the queue. The queue must be non-empty.

Arguments:

q:

The queue.

Returns:

The last element in the queue.

Time complexity: O(1).

7.7.11. `igraph_dqueue_push` — Appends an element.

```
int igraph_dqueue_push (igraph_dqueue_t* q, igraph_real_t elem);
```

Append an element to the end of the queue.

Arguments:

`q`:

The queue.

`elem`:

The element to append.

Returns:

Error code.

Time complexity: O(1) if no memory allocation is needed, O(n), the number of elements in the queue otherwise. But not that by allocating always twice as much memory as the current size of the queue we ensure that n push operations can always be done in at most O(n) time. (Assuming memory allocation is at most linear.)

7.8. Maximum and minimum heaps

7.8.1. `igraph_heap_init` — Initializes an empty heap object.

```
int igraph_heap_init(igraph_heap_t* h, long int alloc_size);
```

Creates an empty heap, but allocates size for some elements.

Arguments:

h:

Pointer to an uninitialized heap object.

alloc_size:

Number of elements to allocate memory for.

Returns:

Error code.

Time complexity: O(*alloc_size*), assuming memory allocation is a linear operation.

7.8.2. `igraph_heap_init_array` — Build a heap from an array.

```
int igraph_heap_init_array(igraph_heap_t *h, igraph_real_t* data, long int len);
```

Initializes a heap object from an array, the heap is also built of course (constructor).

Arguments:

h:

Pointer to an uninitialized heap object.

data:

Pointer to an array of base data type.

len:

The length of the array at *data*.

Returns:

Error code.

Time complexity: O(n), the number of elements in the heap.

7.8.3. `igraph_heap_destroy` — Destroys an initialized heap object.

```
void igraph_heap_destroy(igraph_heap_t* h);
```

Arguments:

h:

The heap object.

Time complexity: O(1).

7.8.4. `igraph_heap_empty` — Decides whether a heap object is empty.

```
igraph_bool_t igraph_heap_empty(igraph_heap_t* h);
```

Arguments:

h:

The heap object.

Returns:

`TRUE` if the heap is empty, `FALSE` otherwise.

TIme complexity: O(1).

7.8.5. `igraph_heap_push` — Add an element.

```
int igraph_heap_push(igraph_heap_t* h, igraph_real_t elem);
```

Adds an element to the heap.

Arguments:

h:

The heap object.

elem:

The element to add.

Returns:

Error code.

Time complexity: O(log n), n is the number of elements in the heap if no reallocation is needed, O(n) otherwise. It is ensured that n push operations are performed in O(n log n) time.

7.8.6. `igraph_heap_top` — Top element.

```
igraph_real_t igraph_heap_top(igraph_heap_t* h);
```

For maximum heaps this is the largest, for minimum heaps the smallest element of the heap.

Arguments:

h:

The heap object.

Returns:

The top element.

Time complexity: O(1).

7.8.7. `igraph_heap_delete_top` — Return and removes the top element

```
igraph_real_t igraph_heap_delete_top(igraph_heap_t* h);
```

Removes and returns the top element of the heap. For maximum heaps this is the largest, for minimum heaps the smallest element.

Arguments:

h:

The heap object.

Returns:

The top element.

Time complexity: O(log n), n is the number of elements in the heap.

7.8.8. `igraph_heap_size` — Number of elements

```
long int igraph_heap_size(igraph_heap_t* h);
```

Gives the number of elements in a heap.

Arguments:

h:

The heap object.

Returns:

The number of elements in the heap.

Time complexity: O(1).

7.8.9. `igraph_heap_reserve` — Allocate more memory

```
int igraph_heap_reserve(igraph_heap_t* h, long int size);
```

Allocates memory for future use. The size of the heap is unchanged. If the heap is larger than the *size* parameter then nothing happens.

Arguments:

h:

The heap object.

size:

The number of elements to allocate memory for.

Returns:

Error code.

Time complexity: O(*size*) if *size* is larger than the current number of elements. O(1) otherwise.

7.9. String vectors

The igraph_strvector_t type is a vector of strings. The current implementation is very simple and not too efficient. It works fine for not too many strings, e.g. the list of attribute names is returned in a string vector by `igraph_cattribute_list()`. Do not expect great performance from this type.

Example 7-11. File examples/simple/igraph_strvector.c

7.9.1. `igraph_strvector_init` — Initialize

```
int igraph_strvector_init(igraph_strvector_t *sv, long int len);
```

Reserves memory for the string vector, a string vector must be first initialized before calling other functions on it. All elements of the string vector are set to the empty string.

Arguments:

`sv`:

Pointer to an initialized string vector.

`len`:

The (initial) length of the string vector.

Returns:

Error code.

Time complexity: O(*`len`*).

7.9.2. `igraph_strvector_copy` — Initialization by copying.

```
int igraph_strvector_copy(igraph_strvector_t *to,
     const igraph_strvector_t *from);
```

Initializes a string vector by copying another string vector.

Arguments:

to:

Pointer to an uninitialized string vector.

from:

The other string vector, to be copied.

Returns:

Error code.

Time complexity: O(l), the total length of the strings in *from*.

7.9.3. `igraph_strvector_destroy` — Free allocated memory

```
void igraph_strvector_destroy(igraph_strvector_t *sv);
```

Destroy a string vector. It may be reinitialized with `igraph_strvector_init()` later.

Arguments:

sv:

The string vector.

Time complexity: O(l), the total length of the strings, maybe less depending on the memory manager.

7.9.4. `STR` — Indexing string vectors

```
#define STR(sv,i)
```

This is a macro which allows to query the elements of a string vector in simpler way than `igraph_strvector_get()`. Note this macro cannot be used to set an element, for that use `igraph_strvector_set()`.

Arguments:

sv:

The string vector

i:

The the index of the element.

Returns:

The element at position *i*.

Time complexity: O(1).

7.9.5. `igraph_strvector_get` — Indexing

```
void igraph_strvector_get(const igraph_strvector_t *sv, long int idx,
     char **value);
```

Query an element of a string vector. See also the `STR` macro for an easier way.

Arguments:

sv:

The input string vector.

idx:

The index of the element to query.

Pointer:

to a char*, the address of the string is stored here.

Time complexity: O(1).

7.9.6. `igraph_strvector_set` — Set an element

```
int igraph_strvector_set(igraph_strvector_t *sv, long int idx,
    const char *value);
```

The provided *value* is copied into the *idx* position in the string vector.

Arguments:

sv:

The string vector.

idx:

The position to set.

value:

The new value.

Returns:

Error code.

Time complexity: O(l), the length of the new string. Maybe more, depending on the memory management, if reallocation is needed.

7.9.7. `igraph_strvector_set2` — Sets an element

```
int igraph_strvector_set2(igraph_strvector_t *sv, long int idx,
     const char *value, int len);
```

This is almost the same as `igraph_strvector_set`, but the new value is not a zero terminated string, but its length is given.

Arguments:

sv:

The string vector.

idx:

The position to set.

value:

The new value.

len:

The length of the new value.

Returns:

Error code.

Time complexity: O(l), the length of the new string. Maybe more, depending on the memory management, if reallocation is needed.

7.9.8. `igraph_strvector_remove` — Removes a single element from a string vector.

```
void igraph_strvector_remove(igraph_strvector_t *v, long int elem);
```

The string will be one shorter.

Arguments:

The:

string vector.

elem:

The index of the element to remove.

Time complexity: O(n), the length of the string.

7.9.9. `igraph_strvector_append` — Concatenate two string vectors.

```
int igraph_strvector_append(igraph_strvector_t *to,
       const igraph_strvector_t *from);
```

Arguments:

to:

The first string vector, the result is stored here.

from:

The second string vector, it is kept unchanged.

Returns:

Error code.

Time complexity: O(n+l2), n is the number of strings in the new string vector, l2 is the total length of strings in the *from* string vector.

7.9.10. `igraph_strvector_clear` — Remove all elements

```
void igraph_strvector_clear(igraph_strvector_t *sv);
```

After this operation the string vector will be empty.

Arguments:

sv:

The string vector.

Time complexity: O(l), the total length of strings, maybe less, depending on the memory manager.

7.9.11. `igraph_strvector_resize` — Resize

```
int igraph_strvector_resize(igraph_strvector_t* v, long int newsize);
```

If the new size is bigger then empty strings are added, if it is smaller then the unneeded elements are removed.

Arguments:

v:

The string vector.

newsize:

The new size.

Returns:

Error code.

Time complexity: O(n), the number of strings if the vector is made bigger, O(l), the total length of the deleted strings if it is made smaller, maybe less, depending on memory management.

7.9.12. `igraph_strvector_size` — Gives the size of a string vector.

```
long int igraph_strvector_size(const igraph_strvector_t *sv);
```

Arguments:

sv:

The string vector.

Returns:

The length of the string vector.

Time complexity: O(1).

7.9.13. `igraph_strvector_add` — Adds an element to the back of a string vector.

```
int igraph_strvector_add(igraph_strvector_t *v, const char *value);
```

Arguments:

`v`:

The string vector.

`value`:

The string to add, it will be copied.

Returns:

Error code.

Time complexity: O(n+l), n is the total number of strings, l is the length of the new string.

7.10. Adjacency lists

Sometimes it is easier to work with a graph which is in adjacency list format: a list of vectors; each vector contains the neighbor vertices or incident edges of a given vertex. Typically, this representation is good if we need to iterate over the neighbors of all vertices many times. E.g. when finding the shortest paths between every pairs of vertices or calculating closeness centrality for all the vertices.

The igraph_adjlist_t stores the adjacency lists of a graph. After creation it is independent of the original graph, it can be modified freely with the usual vector operations, the graph is not affected. E.g. the adjacency list can be used to rewire the edges of a graph efficiently. If one used the straightforward `igraph_delete_edges()` and `igraph_add_edges()` combination for this that needs O(|V|+|E|) time for every single deletion and insertion operation, it is thus very slow if many edges are rewired. Extracting the graph into an adjacency list, do all the rewiring operations on the vectors of the adjacency list and then creating a new graph needs (depending on how exactly the rewiring is done) typically O(|V|+|E|) time for the whole rewiring process.

Lazy adjacency lists are a bit different. When creating a lazy adjacency list, the neighbors of the vertices are not queried, only some memory is allocated for the vectors. When `igraph_lazy_adjlist_get()` is called for vertex v the first time, the neighbors of v are queried and stored in a vector of the adjacency list, so they don't need to be queried again. Lazy adjacency lists are handy if you have an at least linear operation (because initialization is generally linear in terms of number of vertices), but you don't know how many vertices you will visit during the computation.

Example 7-12. File examples/simple/adjlist.c

7.10.1. Adjacent vertices

7.10.1.1. `igraph_adjlist_init` — Initialize an adjacency list of vertices from a given graph

```
int igraph_adjlist_init(const igraph_t *graph, igraph_adjlist_t *al,
     igraph_neimode_t mode);
```

Create a list of vectors containing the neighbors of all vertices in a graph. The adjacency list is independent of the graph after creation, e.g. the graph can be destroyed and modified, the adjacency list contains the state of the graph at the time of its initialization.

Arguments:

graph:

The input graph.

al:

Pointer to an uninitialized igraph_adjlist_t object.

mode:

Constant specifying whether outgoing (`IGRAPH_OUT`), incoming (`IGRAPH_IN`), or both (`IGRAPH_ALL`) types of neighbors to include in the adjacency list. It is ignored for undirected networks.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

7.10.1.2. `igraph_adjlist_init_empty` — Initialize an empty adjacency list

```
int igraph_adjlist_init_empty(igraph_adjlist_t *al, igraph_integer_t no_of_nodes);
```

Creates a list of vectors, one for each vertex. This is useful when you are *constructing* a graph using an adjacency list representation as it does not require your graph to exist yet.

Arguments:

`no_of_nodes`:

The number of vertices

`al`:

Pointer to an uninitialized igraph_adjlist_t object.

Returns:

Error code.

Time complexity: O(|V|), linear in the number of vertices.

7.10.1.3. `igraph_adjlist_init_complementer` — Adjacency lists for the complementer graph

```
int igraph_adjlist_init_complementer(const igraph_t *graph,
            igraph_adjlist_t *al,
            igraph_neimode_t mode,
            igraph_bool_t loops);
```

This function creates adjacency lists for the complementer of the input graph. In the complementer graph all edges are present which are not present in the original graph. Multiple edges in the input graph are ignored.

Arguments:

graph:

The input graph.

al:

Pointer to a not yet initialized adjacency list.

mode:

Constant specifying whether outgoing (`IGRAPH_OUT`), incoming (`IGRAPH_IN`), or both (`IGRAPH_ALL`) types of neighbors (in the complementer graph) to include in the adjacency list. It is ignored for undirected networks.

loops:

Whether to consider loop edges.

Returns:

Error code.

Time complexity: O(|V|^2+|E|), quadratic in the number of vertices.

7.10.1.4. `igraph_adjlist_destroy` — Deallocate memory

```
void igraph_adjlist_destroy(igraph_adjlist_t *al);
```

Free all memory allocated for an adjacency list.

Arguments:

al:

The adjacency list to destroy.

Time complexity: depends on memory management.

7.10.1.5. `igraph_adjlist_get` — Query a vector in an adjlist

```
#define igraph_adjlist_get(al,no)
```

Returns a pointer to an igraph_vector_t object from an adjacency list. The vector can be modified as desired.

Arguments:

al:

The adjacency list object.

no:

The vertex of which the vertex of adjacent vertices are returned.

Returns:

Pointer to the igraph_vector_t object.

Time complexity: O(1).

7.10.1.6. `igraph_adjlist_clear` — Removes all edges from an adjacency list.

```
void igraph_adjlist_clear(igraph_adjlist_t *al);
```

Arguments:

al:

The adjacency list. Time complexity: depends on memory management, typically O(n), where n is the total number of elements in the adjacency list.

7.10.1.7. `igraph_adjlist_sort` — Sort each vector in an adjacency list.

```
void igraph_adjlist_sort(igraph_adjlist_t *al);
```

Sorts every vector of the adjacency list.

Arguments:

`al`:

The adjacency list.

Time complexity: O(n log n), n is the total number of elements in the adjacency list.

7.10.1.8. `igraph_adjlist_simplify` — Simplify

```
int igraph_adjlist_simplify(igraph_adjlist_t *al);
```

Simplify an adjacency list, ie. remove loop and multiple edges.

Arguments:

`al`:

The adjacency list.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of edges and vertices.

7.10.2. Incident edges

7.10.2.1. igraph_inclist_init — Initialize an incidence list of edges

```
int igraph_inclist_init(const igraph_t *graph,
         igraph_inclist_t *il,
         igraph_neimode_t mode);
```

Create a list of vectors containing the incident edges for all vertices. The incidence list is independent of the graph after creation, subsequent changes of the graph object do not update the incidence list, and changes to the incidence list do not update the graph.

Arguments:

graph:

The input graph.

il:

Pointer to an uninitialized incidence list.

mode:

Constant specifying whether incoming edges (IGRAPH_IN), outgoing edges (IGRAPH_OUT) or both (IGRAPH_ALL) to include in the incidence lists of directed graphs. It is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

7.10.2.2. igraph_inclist_destroy — Frees all memory allocated for an incidence list.

```
void igraph_inclist_destroy(igraph_inclist_t *il);
```

Arguments:

eal:

The incidence list to destroy.

Time complexity: depends on memory management.

7.10.2.3. `igraph_inclist_get` — Query a vector in an incidence list

```
#define igraph_inclist_get(il,no)
```

Returns a pointer to an igraph_vector_t object from an incidence list containing edge ids. The vector can be modified, resized, etc. as desired.

Arguments:

graph:

il The incidence list.

no:

The vertex for which the incident edges are returned.

Returns:

Pointer to an igraph_vector_t object.

Time complexity: O(1).

7.10.2.4. `igraph_inclist_clear` — Removes all edges from an incidence list.

```
void igraph_inclist_clear(igraph_inclist_t *il);
```

Arguments:

il:

The incidence list. Time complexity: depends on memory management, typically O(n), where n is the total number of elements in the incidence list.

7.10.3. Lazy adjacency list for vertices

7.10.3.1. `igraph_lazy_adjlist_init` — Constructor

```
int igraph_lazy_adjlist_init(const igraph_t *graph,
          igraph_lazy_adjlist_t *al,
          igraph_neimode_t mode,
          igraph_lazy_adlist_simplify_t simplify);
```

Create a lazy adjacency list for vertices. This function only allocates some memory for storing the vectors of an adjacency list, but the neighbor vertices are not queried, only at the `igraph_lazy_adjlist_get()` calls.

Arguments:

graph:

The input graph.

al:

Pointer to an uninitialized adjacency list object.

mode:

Constant, it gives whether incoming edges (`IGRAPH_IN`), outgoing edges (`IGRPAH_OUT`) or both types of edges (`IGRAPH_ALL`) are considered. It is ignored for undirected graphs.

simplify:

Constant, it gives whether to simplify the vectors in the adjacency list (`IGRAPH_SIMPLIFY`) or not (`IGRAPH_DONT_SIMPLIFY`).

Returns:

Error code.

Time complexity: O(|V|), the number of vertices, possibly, but depends on the underlying memory management too.

7.10.3.2. `igraph_lazy_adjlist_destroy` — Deallocate memory

```
void igraph_lazy_adjlist_destroy(igraph_lazy_adjlist_t *al);
```

Free all allocated memory for a lazy adjacency list.

Arguments:

`al`:

The adjacency list to deallocate.

Time complexity: depends on the memory management.

7.10.3.3. `igraph_lazy_adjlist_get` — Query neighbor vertices

```
#define igraph_lazy_adjlist_get(al,no)
```

If the function is called for the first time for a vertex then the result is stored in the adjacency list and no further query operations are needed when the neighbors of the same vertex are queried again.

Arguments:

`al`:

The lazy adjacency list.

`no`:

The vertex id to query.

Returns:

Pointer to a vector. It is allowed to modify it and modification does not affect the original graph.

Time complexity: O(d), the number of neighbor vertices for the first time, O(1) for subsequent calls.

7.10.3.4. `igraph_lazy_adjlist_clear` — Removes all edges from a lazy adjacency list.

```
void igraph_lazy_adjlist_clear(igraph_lazy_adjlist_t *al);
```

Arguments:

`al`:

The lazy adjacency list. Time complexity: depends on memory management, typically O(n), where n is the total number of elements in the adjacency list.

7.10.4. Lazy incidence list for edges

7.10.4.1. `igraph_lazy_inclist_init` — Initializes a lazy incidence list of edges

```
int igraph_lazy_inclist_init(const igraph_t *graph,
       igraph_lazy_inclist_t *al,
       igraph_neimode_t mode);
```

Create a lazy incidence list for edges. This function only allocates some memory for storing the vectors of an incidence list, but the incident edges are not queried, only when `igraph_lazy_inclist_get()` is called.

Arguments:

`graph`:

The input graph.

`al`:

Pointer to an uninitialized incidence list.

mode:

Constant, it gives whether incoming edges (`IGRAPH_IN`), outgoing edges (`IGRPAH_OUT`) or both types of edges (`IGRAPH_ALL`) are considered. It is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|V|), the number of vertices, possibly. But it also depends on the underlying memory management.

7.10.4.2. `igraph_lazy_inclist_destroy` — Deallocates memory

```
void igraph_lazy_inclist_destroy(igraph_lazy_inclist_t *il);
```

Frees all allocated memory for a lazy incidence list.

Arguments:

al:

The incidence list to deallocate.

Time complexity: depends on memory management.

7.10.4.3. `igraph_lazy_inclist_get` — Query incident edges

```
#define igraph_lazy_inclist_get(al,no)
```

If the function is called for the first time for a vertex, then the result is stored in the incidence list and no further query operations are needed when the incident edges of the same vertex are queried again.

Arguments:

al:

The lazy incidence list object.

no:

The vertex id to query.

Returns:

Pointer to a vector. It is allowed to modify it and modification does not affect the original graph.

Time complexity: O(d), the number of incident edges for the first time, O(1) for subsequent calls with the same *no* argument.

7.10.4.4. `igraph_lazy_inclist_clear` — Removes all edges from a lazy incidence list.

```
void igraph_lazy_inclist_clear(igraph_lazy_inclist_t *il);
```

Arguments:

il:

The lazy incidence list. Time complexity: depends on memory management, typically O(n), where n is the total number of elements in the incidence list.

7.10.5. Deprecated functions

7.10.5.1. `igraph_adjedgelist_init` — Initialize an incidence list of edges

```
int igraph_adjedgelist_init(const igraph_t *graph,
                   igraph_inclist_t *il,
                   igraph_neimode_t mode);
```

This function was superseded by `igraph_inclist_init()` in igraph 0.6. Please use `igraph_inclist_init()` instead of this function.

Deprecated in version 0.6.

7.10.5.2. `igraph_adjedgelist_destroy` — Frees all memory allocated for an incidence list.

```
void igraph_adjedgelist_destroy(igraph_inclist_t *il);
```

This function was superseded by `igraph_inclist_destroy()` in igraph 0.6. Please use `igraph_inclist_destroy()` instead of this function.

Deprecated in version 0.6.

7.10.5.3. `igraph_adjedgelist_get` — Query a vector in an incidence list

```
#define igraph_adjedgelist_get(ael,no)
```

This macro was superseded by `igraph_inclist_get()` in igraph 0.6. Please use `igraph_inclist_get()` instead of this macro.

Deprecated in version 0.6.

7.10.5.4. `igraph_lazy_adjedgelist_init` — Initializes a lazy incidence list of edges

```
int igraph_lazy_adjedgelist_init(const igraph_t *graph,
                   igraph_lazy_inclist_t *il,
                   igraph_neimode_t mode);
```

This function was superseded by `igraph_lazy_inclist_init()` in igraph 0.6. Please use `igraph_lazy_inclist_init()` instead of this function.

Deprecated in version 0.6.

7.10.5.5. `igraph_lazy_adjedgelist_destroy` — Frees all memory allocated for an incidence list.

```
void igraph_lazy_adjedgelist_destroy(igraph_lazy_inclist_t *il);
```

This function was superseded by `igraph_lazy_inclist_destroy()` in igraph 0.6. Please use `igraph_lazy_inclist_destroy()` instead of this function.

Deprecated in version 0.6.

7.10.5.6. `igraph_lazy_adjedgelist_get` — Query a vector in a lazy incidence list

```
#define igraph_lazy_adjedgelist_get(al,no)
```

This macro was superseded by `igraph_lazy_inclist_get()` in igraph 0.6. Please use `igraph_lazy_inclist_get()` instead of this macro.

Deprecated in version 0.6.

Chapter 8. Random numbers

8.1. About random numbers in igraph, use cases

Some algorithms in igraph, e.g. the generation of random graphs, require random number generators (RNGs). Prior to version 0.6 igraph did not have a sophisticated way to deal with random number generators at the C level, but this has changed. From version 0.6 different and multiple random number generators are supported.

8.2. The default random number generator

8.2.1. `igraph_rng_default` — Query the default random number generator.

```
igraph_rng_t *igraph_rng_default();
```

Returns:

A pointer to the default random number generator.

See also:

igraph_rng_set_default()

8.2.2. `igraph_rng_set_default` — Set the default igraph random number generator

```
void igraph_rng_set_default(igraph_rng_t *rng);
```

Arguments:

rng:

The random number generator to use as default from now on. Calling `igraph_rng_destroy()` on it, while it is still being used as the default will result craches and/or unpredictable results.

Time complexity: O(1).

8.3. Creating random number generators

8.3.1. `igraph_rng_init` — Initialize a random number generator

```
int igraph_rng_init(igraph_rng_t *rng, const igraph_rng_type_t *type);
```

This function allocates memory for a random number generator, with the given type, and sets its seed to the default.

Arguments:

rng:

Pointer to an uninitialized RNG.

type:

The type of the RNG, please see the documentation for the supported types.

Returns:

Error code.

Time complexity: depends on the type of the generator, but usually it should be O(1).

8.3.2. `igraph_rng_destroy` — Deallocate memory associated with a random number generator

```
void igraph_rng_destroy(igraph_rng_t *rng);
```

Arguments:

rng:

The RNG to destroy. Do not destroy an RNG that is used as the default igraph RNG.

Time complexity: O(1).

8.3.3. `igraph_rng_seed` — Set the seed of a random number generator

```
int igraph_rng_seed(igraph_rng_t *rng, unsigned long int seed);
```

Arguments:

rng:

The RNG.

seed:

The new seed.

Returns:

Error code.

Time complexity: usually O(1), but may depend on the type of the RNG.

8.3.4. `igraph_rng_min` — Query the minimum possible integer for a random number generator

```
unsigned long int igraph_rng_min(igraph_rng_t *rng);
```

Arguments:

rng:

The RNG.

Returns:

The smallest possible integer that can be generated by calling `igraph_rng_get_integer()` on the RNG.

Time complexity: O(1).

8.3.5. `igraph_rng_max` — Query the maximum possible integer for a random number generator

```
unsigned long int igraph_rng_max(igraph_rng_t *rng);
```

Arguments:

rng:

The RNG.

Returns:

The largest possible integer that can be generated by calling `igraph_rng_get_integer()` on the RNG.

Time complexity: O(1).

8.3.6. `igraph_rng_name` — Query the type of a random number generator

```
const char *igraph_rng_name(igraph_rng_t *rng);
```

Arguments:

rng:

The RNG.

Returns:

The name of the type of the generator. Do not deallocate or change the returned string pointer.

Time complexity: O(1).

8.4. Generating random numbers

8.4.1. `igraph_rng_get_integer` — Generate an integer random number from an interval

```
long int igraph_rng_get_integer(igraph_rng_t *rng,
    long int l, long int h);
```

Arguments:

rng:

Pointer to the RNG to use for the generation. Use `igraph_rng_default()` here to use the default igraph RNG.

l:

Lower limit, inclusive, it can be negative as well.

h:

Upper limit, inclusive, it can be negative as well, but it should be at least `l`.

Returns:

The generated random integer.

Time complexity: depends on the generator, but should be usually O(1).

8.4.2. `igraph_rng_get_unif` — Generate real, uniform random numbers from an interval

```
igraph_real_t igraph_rng_get_unif(igraph_rng_t *rng,
      igraph_real_t l, igraph_real_t h);
```

Arguments:

rng:

Pointer to the RNG to use. Use `igraph_rng_default()` here to use the default igraph RNG.

l:

The lower bound, it can be negative.

h:

The upper bound, it can be negative, but it has to be larger than the lower bound.

Returns:

The generated uniformly distributed random number.

Time complexity: depends on the type of the RNG.

8.4.3. `igraph_rng_get_unif01` — Generate real, uniform random number from the unit interval

```
igraph_real_t igraph_rng_get_unif01(igraph_rng_t *rng);
```

Arguments:

rng:

Pointer to the RNG to use. Use `igraph_rng_default()` here to use the default igraph RNG.

Returns:

The generated uniformly distributed random number.

Time complexity: depends on the type of the RNG.

8.4.4. `igraph_rng_get_normal` — Normally distributed random numbers

```
igraph_real_t igraph_rng_get_normal(igraph_rng_t *rng,
        igraph_real_t m, igraph_real_t s);
```

Arguments:

rng:

Pointer to the RNG to use. Use `igraph_rng_default()` here to use the default igraph RNG.

m:

The mean.

s:

Standard deviation.

Returns:

The generated normally distributed random number.

Time complexity: depends on the type of the RNG.

8.4.5. `igraph_rng_get_geom` — Generate geometrically distributed random numbers

```
igraph_real_t igraph_rng_get_geom(igraph_rng_t *rng, igraph_real_t p);
```

Arguments:

rng:

Pointer to the RNG to use. Use `igraph_rng_default()` here to use the default igraph RNG.

p:

The probability of success in each trial. Must be larger than zero and smaller or equal to 1.

Returns:

The generated geometrically distributed random number.

Time complexity: depends on the type of the RNG.

8.4.6. `igraph_rng_get_binom` — Generate binomially distributed random numbers

```
igraph_real_t igraph_rng_get_binom(igraph_rng_t *rng, long int n,
       igraph_real_t p);
```

Arguments:

rng:

Pointer to the RNG to use. Use `igraph_rng_default()` here to use the default igraph RNG.

n:

Number of observations.

p:

Probability of an event.

Returns:

The generated binomially distributed random number.

Time complexity: depends on the type of the RNG.

8.5. Supported random number generators

By default igraph uses the MT19937 generator. Prior to igraph version 0.6, the generator supplied by the standard C library was used. This means the GLIBC2 generator on GNU libc 2 systems, and maybe the RAND generator on others.

8.5.1. `igraph_rngtype_mt19937` — The MT19937 random number generator

```
const igraph_rng_type_t igraph_rngtype_mt19937 = {
  /* name= */      "MT19937",
```

```
  /* min=  */      0,
  /* max=  */      0xffffffffUL,
  /* init= */      igraph_rng_mt19937_init,
  /* destroy= */   igraph_rng_mt19937_destroy,
  /* seed= */      igraph_rng_mt19937_seed,
  /* get= */       igraph_rng_mt19937_get,
  /* get_real= */  igraph_rng_mt19937_get_real,
  /* get_norm= */  0,
  /* get_geom= */  0,
  /* get_binom= */ 0,
  /* get_exp= */   0
};
```

The MT19937 generator of Makoto Matsumoto and Takuji Nishimura is a variant of the twisted generalized feedback shift-register algorithm, and is known as the “Mersenne Twister” generator. It has a Mersenne prime period of 2^19937 - 1 (about 10^6000) and is equi-distributed in 623 dimensions. It has passed the diehard statistical tests. It uses 624 words of state per generator and is comparable in speed to the other generators. The original generator used a default seed of 4357 and choosing s equal to zero in gsl_rng_set reproduces this. Later versions switched to 5489 as the default seed, you can choose this explicitly via igraph_rng_seed instead if you require it. For more information see, Makoto Matsumoto and Takuji Nishimura, “Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator”. ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1 (Jan. 1998), Pages 3–30 The generator igraph_rngtype_mt19937 uses the second revision of the seeding procedure published by the two authors above in 2002. The original seeding procedures could cause spurious artifacts for some seed values. This generator was ported from the GNU Scientific Library.

8.5.2. `igraph_rngtype_glibc2` — The random number generator type introduced in GNU libc 2

```
const igraph_rng_type_t igraph_rngtype_glibc2 = {
  /* name= */      "LIBC",
  /* min=  */      0,
  /* max=  */      RAND_MAX,
  /* init= */      igraph_rng_glibc2_init,
  /* destroy= */   igraph_rng_glibc2_destroy,
  /* seed= */      igraph_rng_glibc2_seed,
  /* get= */       igraph_rng_glibc2_get,
  /* get_real= */  igraph_rng_glibc2_get_real,
  /* get_norm= */  0,
  /* get_geom= */  0,
  /* get_binom= */ 0,
  /* get_exp= */   0
};
```

It is a linear feedback shift register generator with a 128-byte buffer. This generator was the default prior to igraph version 0.6, at least on systems relying on GNU libc. This generator was ported from the GNU Scientific Library.

8.5.3. `igraph_rngtype_rand` — The old BSD rand/stand random number generator

```
const igraph_rng_type_t igraph_rngtype_rand = {
  /* name= */      "RAND",
  /* min=  */      0,
  /* max=  */      0x7fffffffUL,
  /* init= */      igraph_rng_rand_init,
  /* destroy= */   igraph_rng_rand_destroy,
  /* seed= */      igraph_rng_rand_seed,
  /* get= */       igraph_rng_rand_get,
  /* get_real= */  igraph_rng_rand_get_real,
  /* get_norm= */  0,
  /* get_geom= */  0,
  /* get_binom= */ 0,
  /* get_exp= */   0
};
```

The sequence is x_{n+1} = (a x_n + c) mod m with a = 1103515245, c = 12345 and m = 2^31 = 2147483648. The seed specifies the initial value, x_1. The theoretical value of x_{10001} is 1910041713. The period of this generator is 2^31. This generator is not very good -- the low bits of successive numbers are correlated. This generator was ported from the GNU Scientific Library.

8.6. Use cases

8.6.1. Normal (default) use

If the user does not use any of the RNG functions explicitly, but calls some of the randomized igraph functions, then a default RNG is set up the first time an igraph function needs random numbers. The seed of this RNG is the output of the `time(0)` function call, using the `time` function from the standard C library. This ensures that igraph creates a different random graph, each time the C program is called.

The created default generator is stored internally and can be queried with the `igraph_rng_default()` function.

8.6.2. Reproducible simulations

If reproducible results are needed, then the user should set the seed of the default random number generator explicitly, using the `igraph_rng_seed()` function on the default generator, `igraph_rng_default()`. When setting the seed to the same number, igraph generates exactly the same random graph (or series of random graphs).

8.6.3. Changing the default generator

By default igraph uses the `igraph_rng_default()` random number generator. This can be changed any time by calling `igraph_rng_set_default()`, with an already initialized random number generator. Note that the old (replaced) generator is not destroyed, so no memory is deallocated.

8.6.4. Using multiple generators

igraph also provides functions to set up multiple random number generators, using the `igraph_rng_init()` function, and then generating random numbers from them, e.g. with `igraph_rng_get_integer()` and/or `igraph_rng_get_unif()` calls.

Note that initializing a new random number generator is independent of the generator that the igraph functions themselves use. If you want to replace that, then please use `igraph_rng_set_default()`.

8.6.5. Example

Example 8-1. File examples/simple/random_seed.c

Chapter 9. Graph Generators

Graph generators create graphs.

Almost all functions which create graph objects are documented here. The exceptions are `igraph_subgraph()` and alike, these create graphs based on another graph.

9.1. Deterministic Graph Generators

9.1.1. `igraph_create` — Creates a graph with the specified edges.

```
int igraph_create(igraph_t *graph, const igraph_vector_t *edges,
  igraph_integer_t n, igraph_bool_t directed);
```

Arguments:

graph:

An uninitialized graph object.

edges:

The edges to add, the first two elements are the first edge, etc.

n:

The number of vertices in the graph, if smaller or equal to the highest vertex id in the *edges* vector it will be increased automatically. So it is safe to give 0 here.

directed:

Boolean, whether to create a directed graph or not. If yes, then the first edge points from the first vertex id in *edges* to the second, etc.

Returns:

Error code: `IGRAPH_EINVEVECTOR`: invalid edges vector (odd number of vertices). `IGRAPH_EINVVID`: invalid (negative) vertex id.

Time complexity: O(|V|+|E|), |V| is the number of vertices, |E| the number of edges in the graph.

Example 9-1. File examples/simple/igraph_create.c

9.1.2. `igraph_small` — Shorthand to create a short graph, giving the edges as arguments.

```
int igraph_small(igraph_t *graph, igraph_integer_t n, igraph_bool_t directed,
    ...);
```

This function is handy when a relatively small graph needs to be created. Instead of giving the edges as a vector, they are given simply as arguments and a '-1' needs to be given after the last meaningful edge argument.

Note that only graphs which have vertices less than the highest value of the 'int' type can be created this way. If you give larger values then the result is undefined.

Arguments:

`graph`:

Pointer to an uninitialized graph object. The result will be stored here.

`n`:

The number of vertices in the graph; a nonnegative integer.

`directed`:

Logical constant; gives whether the graph should be directed. Supported values are:

`IGRAPH_DIRECTED`

The graph to be created will be *directed.*

`IGRAPH_UNDIRECTED`

The graph to be created will be *undirected.*

`...`:

The additional arguments giving the edges of the graph. Don't forget to supply an additional '-1' after the last (meaningful) argument.

Returns:

Error code.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph to create.

Example 9-2. File examples/simple/igraph_small.c

9.1.3. `igraph_adjacency` — Creates a graph object from an adjacency matrix.

```
int igraph_adjacency(igraph_t *graph, igraph_matrix_t *adjmatrix,
       igraph_adjacency_t mode);
```

The order of the vertices in the matrix is preserved, i.e. the vertex corresponding to the first row/column will be vertex with id 0, the next row is for vertex 1, etc.

Arguments:

`graph`:

Pointer to an uninitialized graph object.

`adjmatrix`:

The adjacency matrix. How it is interpreted depends on the *`mode`* argument.

`mode`:

Constant to specify how the given matrix is interpreted as an adjacency matrix. Possible values (A(i,j) is the element in row i and column j in the adjacency matrix *`adjmatrix`*):

`IGRAPH_ADJ_DIRECTED`

the graph will be directed and an element gives the number of edges between two vertices.

`IGRAPH_ADJ_UNDIRECTED`

this is the same as `IGRAPH_ADJ_MAX`, for convenience.

`IGRAPH_ADJ_MAX`

undirected graph will be created and the number of edges between vertices i and j is max(A(i,j), A(j,i)).

`IGRAPH_ADJ_MIN`

undirected graph will be created with min(A(i,j), A(j,i)) edges between vertices i and j.

`IGRAPH_ADJ_PLUS`

undirected graph will be created with A(i,j)+A(j,i) edges between vertices i and j.

`IGRAPH_ADJ_UPPER`

undirected graph will be created, only the upper right triangle (including the diagonal) is used for the number of edges.

`IGRAPH_ADJ_LOWER`

undirected graph will be created, only the lower left triangle (including the diagonal) is used for creating the edges.

Returns:

Error code, `IGRAPH_NONSQUARE`: non-square matrix.

Time complexity: O(|V||V|), |V| is the number of vertices in the graph.

Example 9-3. File examples/simple/igraph_adjacency.c

9.1.4. `igraph_weighted_adjacency` — Creates a graph object from a weighted adjacency matrix.

```
int igraph_weighted_adjacency(igraph_t *graph, igraph_matrix_t *adjmatrix,
         igraph_adjacency_t mode, const char* attr,
         igraph_bool_t loops);
```

The order of the vertices in the matrix is preserved, i.e. the vertex corresponding to the first row/column will be vertex with id 0, the next row is for vertex 1, etc.

Arguments:

`graph`:

Pointer to an uninitialized graph object.

`adjmatrix`:

The weighted adjacency matrix. How it is interpreted depends on the *`mode`* argument. The common feature is that edges with zero weights are considered nonexistent (however, negative weights are permitted).

`mode`:

Constant to specify how the given matrix is interpreted as an adjacency matrix. Possible values (A(i,j) is the element in row i and column j in the adjacency matrix *`adjmatrix`*):

`IGRAPH_ADJ_DIRECTED`

the graph will be directed and an element gives the weight of the edge between two vertices.

`IGRAPH_ADJ_UNDIRECTED`

this is the same as `IGRAPH_ADJ_MAX`, for convenience.

`IGRAPH_ADJ_MAX`

undirected graph will be created and the weight of the edge between vertices i and j is max(A(i,j), A(j,i)).

`IGRAPH_ADJ_MIN`

undirected graph will be created with edge weight min(A(i,j), A(j,i)) between vertices i and j.

`IGRAPH_ADJ_PLUS`

undirected graph will be created with edge weight A(i,j)+A(j,i) between vertices i and j.

`IGRAPH_ADJ_UPPER`

undirected graph will be created, only the upper right triangle (including the diagonal) is used for the edge weights.

`IGRAPH_ADJ_LOWER`

undirected graph will be created, only the lower left triangle (including the diagonal) is used for the edge weights.

`attr`:

the name of the attribute that will store the edge weights. If `NULL` , it will use `weight` as the attribute name.

`loops`:

Logical scalar, whether to ignore the diagonal elements in the adjacency matrix.

Returns:

Error code, `IGRAPH_NONSQUARE`: non-square matrix.

Time complexity: O(|V||V|), |V| is the number of vertices in the graph.

Example 9-4. File examples/simple/igraph_weighted_adjacency.c

9.1.5. `igraph_adjlist` — Create a graph from an adjacency list

```
int igraph_adjlist(igraph_t *graph, const igraph_adjlist_t *adjlist,
     igraph_neimode_t mode, igraph_bool_t duplicate);
```

An adjacency list is a list of vectors, containing the neighbors of all vertices. For operations that involve many changes to the graph structure, it is recommended that you convert the graph into an adjacency list via `igraph_adjlist_init()`, perform the modifications (these are cheap for an adjacency list) and then recreate the igraph graph via this function.

Arguments:

graph:

Pointer to an uninitialized graph object.

adjlist:

The adjacency list.

mode:

Whether or not to create a directed graph. `IGRAPH_ALL` means an undirected graph, `IGRAPH_OUT` means a directed graph from an out-adjacency list (i.e. each list contains the successors of the corresponding vertices), `IGRAPH_IN` means a directed graph from an in-adjacency list

duplicate:

Logical, for undirected graphs this specified whether each edge is included twice, in the vectors of both adjacent vertices. If this is false (0), then it is assumed that every edge is included only once. This argument is ignored for directed graphs.

Returns:

Error code.

See also:

`igraph_adjlist_init()` for the opposite operation.

Time complexity: O(|V|+|E|).

9.1.6. `igraph_star` — Creates a *star* graph, every vertex connects only to the center.

```
int igraph_star(igraph_t *graph, igraph_integer_t n, igraph_star_mode_t mode,
  igraph_integer_t center);
```

Arguments:

graph:

Pointer to an uninitialized graph object, this will be the result.

n:

Integer constant, the number of vertices in the graph.

mode:

Constant, gives the type of the star graph to create. Possible values:

IGRAPH_STAR_OUT

directed star graph, edges point *from* the center to the other vertices.

IGRAPH_STAR_IN

directed star graph, edges point *to* the center from the other vertices.

IGRAPH_STAR_MUTUAL

directed star graph with mutual edges.

IGRAPH_STAR_UNDIRECTED

an undirected star graph is created.

center:

Id of the vertex which will be the center of the graph.

Returns:

Error code:

IGRAPH_EINVVID

invalid number of vertices.

IGRAPH_EINVAL

invalid center vertex.

IGRAPH_EINVMODE

invalid mode argument.

Time complexity: O(|V|), the number of vertices in the graph.

See also:

`igraph_lattice()`, `igraph_ring()`, `igraph_tree()` for creating other regular structures.

Example 9-5. File examples/simple/igraph_star.c

9.1.7. `igraph_lattice` — Creates most kinds of lattices.

```
int igraph_lattice(igraph_t *graph, const igraph_vector_t *dimvector,
  igraph_integer_t nei, igraph_bool_t directed, igraph_bool_t mutual,
  igraph_bool_t circular);
```

Arguments:

`graph`:

An uninitialized graph object.

`dimvector`:

Vector giving the sizes of the lattice in each of its dimensions. Ie. the dimension of the lattice will be the same as the length of this vector.

`nei`:

Integer value giving the distance (number of steps) within which two vertices will be connected.

`directed`:

Boolean, whether to create a directed graph. The direction of the edges is determined by the generation algorithm and is unlikely to suit you, so this isn't a very useful option.

`mutual`:

Boolean, if the graph is directed this gives whether to create all connections as mutual.

`circular`:

Boolean, defines whether the generated lattice is periodic.

Returns:

Error code: `IGRAPH_EINVAL`: invalid (negative) dimension vector.

Time complexity: if *nei* is less than two then it is O(|V|+|E|) (as far as I remember), |V| and |E| are the number of vertices and edges in the generated graph. Otherwise it is O(|V|*d^o+|E|), d is the average degree of the graph, o is the *nei* argument.

9.1.8. `igraph_ring` — Creates a *ring* graph, a one dimensional lattice.

```
int igraph_ring(igraph_t *graph, igraph_integer_t n, igraph_bool_t directed,
  igraph_bool_t mutual, igraph_bool_t circular);
```

An undirected (circular) ring on n vertices is commonly known in graph theory as the cycle graph C_n.

Arguments:

graph:

Pointer to an uninitialized graph object.

n:

The number of vertices in the ring.

directed:

Logical, whether to create a directed ring.

mutual:

Logical, whether to create mutual edges in a directed ring. It is ignored for undirected graphs.

circular:

Logical, if false, the ring will be open (this is not a real *ring* actually).

Returns:

Error code: `IGRAPH_EINVAL`: invalid number of vertices.

Time complexity: O(|V|), the number of vertices in the graph.

See also:

`igraph_lattice()` for generating more general lattices.

Example 9-6. File examples/simple/igraph_ring.c

9.1.9. `igraph_tree` — Creates a tree in which almost all vertices have the same number of children.

```
int igraph_tree(igraph_t *graph, igraph_integer_t n, igraph_integer_t children,
  igraph_tree_mode_t type);
```

Arguments:

graph:

Pointer to an uninitialized graph object.

n:

Integer, the number of vertices in the graph.

children:

Integer, the number of children of a vertex in the tree.

type:

Constant, gives whether to create a directed tree, and if this is the case, also its orientation. Possible values:

`IGRAPH_TREE_OUT`

directed tree, the edges point from the parents to their children,

`IGRAPH_TREE_IN`

directed tree, the edges point from the children to their parents.

`IGRAPH_TREE_UNDIRECTED`

undirected tree.

Returns:

Error code: `IGRAPH_EINVAL`: invalid number of vertices. `IGRAPH_INVMODE`: invalid mode argument.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph.

See also:

`igraph_lattice()`, `igraph_star()` for creating other regular structures.

Example 9-7. File examples/simple/igraph_tree.c

9.1.10. `igraph_full` — Creates a full graph (directed or undirected, with or without loops).

```
int igraph_full(igraph_t *graph, igraph_integer_t n, igraph_bool_t directed,
  igraph_bool_t loops);
```

In a full graph every possible edge is present, every vertex is connected to every other vertex. A full graph in `igraph` should be distinguished from the concept of complete graphs as used in graph theory. If n is a positive integer, then the complete graph K_n on n vertices is the undirected simple graph with the following property. For any distinct pair (u,v) of vertices in K_n, uv (or equivalently vu) is an edge of K_n. In `igraph`, a full graph on n vertices can be K_n, a directed version of K_n, or K_n with at least one loop edge. In any case, if F is a full graph on n vertices as generated by `igraph`, then K_n is a subgraph of the undirected version of F.

Arguments:

`graph`:

Pointer to an uninitialized graph object.

n:

Integer, the number of vertices in the graph.

directed:

Logical, whether to create a directed graph.

loops:

Logical, whether to include self-edges (loops).

Returns:

Error code: `IGRAPH_EINVAL`: invalid number of vertices.

Time complexity: O(|V|+|E|), |V| is the number of vertices, |E| the number of edges in the graph. Of course this is the same as O(|E|)=O(|V||V|) here.

See also:

`igraph_lattice()`, `igraph_star()`, `igraph_tree()` for creating other regular structures.

Example 9-8. File examples/simple/igraph_full.c

9.1.11. `igraph_full_citation` — Creates a full citation graph

```
int igraph_full_citation(igraph_t *graph, igraph_integer_t n,
    igraph_bool_t directed);
```

This is a directed graph, where every `i->j` edge is present if and only if `j<i`. If the `directed` argument is zero then an undirected graph is created, and it is just a full graph.

Arguments:

graph:

Pointer to an uninitialized graph object, the result is stored here.

n:

The number of vertices.

directed:

Whether to created a directed graph. If zero an undirected graph is created.

Returns:

Error code.

Time complexity: O(|V|^2), as we have many edges.

9.1.12. `igraph_famous` — Create a famous graph by simply providing its name

```
int igraph_famous(igraph_t *graph, const char *name);
```

The name of the graph can be simply supplied as a string. Note that this function creates graphs which don't take any parameters, there are separate functions for graphs with parameters, eg. `igraph_full()` for creating a full graph.

The following graphs are supported:

`Bull`

The bull graph, 5 vertices, 5 edges, resembles the head of a bull if drawn properly.

`Chvatal`

This is the smallest triangle-free graph that is both 4-chromatic and 4-regular. According to the Grunbaum conjecture there exists an m-regular, m-chromatic graph with n vertices for every m>1 and n>2. The Chvatal graph is an example for m=4 and n=12. It has 24 edges.

`Coxeter`

A non-Hamiltonian cubic symmetric graph with 28 vertices and 42 edges.

`Cubical`

The Platonic graph of the cube. A convex regular polyhedron with 8 vertices and 12 edges.

`Diamond`

A graph with 4 vertices and 5 edges, resembles a schematic diamond if drawn properly.

`Dodecahedral, Dodecahedron`

Another Platonic solid with 20 vertices and 30 edges.

`Folkman`

The semisymmetric graph with minimum number of vertices, 20 and 40 edges. A semisymmetric graph is regular, edge transitive and not vertex transitive.

`Franklin`

This is a graph whose embedding to the Klein bottle can be colored with six colors, it is a counterexample to the necessity of the Heawood conjecture on a Klein bottle. It has 12 vertices and 18 edges.

`Frucht`

The Frucht Graph is the smallest cubical graph whose automorphism group consists only of the identity element. It has 12 vertices and 18 edges.

`Grotzsch`

The Grötzsch graph is a triangle-free graph with 11 vertices, 20 edges, and chromatic number 4. It is named after German mathematician Herbert Grötzsch, and its existence demonstrates that the assumption of planarity is necessary in Grötzsch's theorem that every triangle-free planar graph is 3-colorable.

`Heawood`

The Heawood graph is an undirected graph with 14 vertices and 21 edges. The graph is cubic, and all cycles in the graph have six or more edges. Every smaller cubic graph has shorter cycles, so this graph is the 6-cage, the smallest cubic graph of girth 6.

`Herschel`

The Herschel graph is the smallest nonhamiltonian polyhedral graph. It is the unique such graph on 11 nodes, and has 18 edges.

`House`

The house graph is a 5-vertex, 6-edge graph, the schematic draw of a house if drawn properly, basically a triangle on top of a square.

`HouseX`

The same as the house graph with an X in the square. 5 vertices and 8 edges.

`Icosahedral, Icosahedron`

A Platonic solid with 12 vertices and 30 edges.

`Krackhardt_Kite`

A social network with 10 vertices and 18 edges. Krackhardt, D. Assessing the Political Landscape: Structure, Cognition, and Power in Organizations. Admin. Sci. Quart. 35, 342-369, 1990.

`Levi`

The graph is a 4-arc transitive cubic graph, it has 30 vertices and 45 edges.

`McGee`

The McGee graph is the unique 3-regular 7-cage graph, it has 24 vertices and 36 edges.

`Meredith`

The Meredith graph is a quartic graph on 70 nodes and 140 edges that is a counterexample to the conjecture that every 4-regular 4-connected graph is Hamiltonian.

`Noperfectmatching`

A connected graph with 16 vertices and 27 edges containing no perfect matching. A matching in a graph is a set of pairwise non-incident edges; that is, no two edges share a common vertex. A perfect matching is a matching which covers all vertices of the graph.

`Nonline`

A graph whose connected components are the 9 graphs whose presence as a vertex-induced subgraph in a graph makes a nonline graph. It has 50 vertices and 72 edges.

`Octahedral, Octahedron`

Platonic solid with 6 vertices and 12 edges.

`Petersen`

A 3-regular graph with 10 vertices and 15 edges. It is the smallest hypohamiltonian graph, ie. it is non-hamiltonian but removing any single vertex from it makes it Hamiltonian.

`Robertson`

The unique (4,5)-cage graph, ie. a 4-regular graph of girth 5. It has 19 vertices and 38 edges.

`Smallestcyclicgroup`

A smallest nontrivial graph whose automorphism group is cyclic. It has 9 vertices and 15 edges.

`Tetrahedral, Tetrahedron`

Platonic solid with 4 vertices and 6 edges.

`Thomassen`

The smallest hypotraceable graph, on 34 vertices and 52 edges. A hypotracable graph does not contain a Hamiltonian path but after removing any single vertex from it the remainder always contains a Hamiltonian path. A graph containing a Hamiltonian path is called traceable.

`Tutte`

Tait's Hamiltonian graph conjecture states that every 3-connected 3-regular planar graph is Hamiltonian. This graph is a counterexample. It has 46 vertices and 69 edges.

`Uniquely3colorable`

Returns a 12-vertex, triangle-free graph with chromatic number 3 that is uniquely 3-colorable.

`Walther`

An identity graph with 25 vertices and 31 edges. An identity graph has a single graph automorphism, the trivial one.

`Zachary`

Social network of friendships between 34 members of a karate club at a US university in the 1970s. See W. W. Zachary, An information flow model for conflict and fission in small groups, Journal of Anthropological Research 33, 452-473 (1977).

Arguments:

`graph`:

Pointer to an uninitialized graph object.

`name`:

Character constant, the name of the graph to be created, it is case insensitive.

Returns:

Error code, IGRAPH_EINVAL if there is no graph with the given name.

See also:

Other functions for creating graph structures: `igraph_ring()`, `igraph_tree()`, `igraph_lattice()`, `igraph_full()`.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph.

9.1.13. `igraph_lcf` — Create a graph from LCF notation

```
int igraph_lcf(igraph_t *graph, igraph_integer_t n, ...);
```

LCF is short for Lederberg-Coxeter-Frucht, it is a concise notation for 3-regular Hamiltonian graphs. It consists of three parameters: the number of vertices in the graph, a list of shifts giving additional edges to a cycle backbone, and another integer giving how many times the shifts should be performed. See http://mathworld.wolfram.com/LCFNotation.html for details.

Arguments:

`graph`:

Pointer to an uninitialized graph object.

`n`:

Integer, the number of vertices in the graph.

`...`:

The shifts and the number of repeats for the shifts, plus an additional 0 to mark the end of the arguments.

Returns:

Error code.

See also:

See `igraph_lcf_vector()` for a similar function using a vector_t instead of the variable length argument list.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

Example 9-9. File examples/simple/igraph_lcf.c

9.1.14. `igraph_lcf_vector` — Create a graph from LCF notation

```
int igraph_lcf_vector(igraph_t *graph, igraph_integer_t n,
        const igraph_vector_t *shifts,
        igraph_integer_t repeats);
```

This function is essentially the same as `igraph_lcf()`, only the way for giving the arguments is different. See `igraph_lcf()` for details.

Arguments:

`graph`:

Pointer to an uninitialized graph object.

`n`:

Integer constant giving the number of vertices.

`shifts`:

A vector giving the shifts.

`repeats`:

An integer constant giving the number of repeats for the shifts.

Returns:

Error code.

See also:

`igraph_lcf()`

Time complexity: O(|V|+|E|), linear in the number of vertices plus the number of edges.

9.1.15. `igraph_atlas` — Create a small graph from the “Graph Atlas”.

```
int igraph_atlas(igraph_t *graph, int number);
```

The number of the graph is given as a parameter. The graphs are listed:

1. in increasing order of number of nodes;
2. for a fixed number of nodes, in increasing order of the number of edges;
3. for fixed numbers of nodes and edges, in increasing order of the degree sequence, for example 111223 < 112222;
4. for fixed degree sequence, in increasing number of automorphisms.

The data was converted from the networkx software package, see http://networkx.lanl.gov.

See *An Atlas of Graphs* by Ronald C. Read and Robin J. Wilson, Oxford University Press, 1998.

Arguments:

`graph`:

Pointer to an uninitialized graph object.

`number`:

The number of the graph to generate.

Added in version 0.2.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

Example 9-10. File examples/simple/igraph_atlas.c

9.1.16. `igraph_de_bruijn` — Generate a de Bruijn graph.

```
int igraph_de_bruijn(igraph_t *graph, igraph_integer_t m, igraph_integer_t n);
```

A de Bruijn graph represents relationships between strings. An alphabet of `m` letters are used and strings of length `n` are considered. A vertex corresponds to every possible string and there is a directed edge from vertex `v` to vertex `w` if the string of `v` can be transformed into the string of `w` by removing its first letter and appending a letter to it.

Please note that the graph will have `m` to the power `n` vertices and even more edges, so probably you don't want to supply too big numbers for `m` and `n`.

De Bruijn graphs have some interesting properties, please see another source, eg. Wikipedia for details.

Arguments:

graph:

Pointer to an uninitialized graph object, the result will be stored here.

m:

Integer, the number of letters in the alphabet.

n:

Integer, the length of the strings.

Returns:

Error code.

See also:

`igraph_kautz()`.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

9.1.17. `igraph_kautz` — Generate a Kautz graph.

```
int igraph_kautz(igraph_t *graph, igraph_integer_t m, igraph_integer_t n);
```

A Kautz graph is a labeled graph, vertices are labeled by strings of length `n`+1 above an alphabet with `m`+1 letters, with the restriction that every two consecutive letters in the string must be different. There is a directed edge from a vertex `v` to another vertex `w` if it is possible to transform the string of `v` into the string of `w` by removing the first letter and appending a letter to it.

Kautz graphs have some interesting properties, see eg. Wikipedia for details.

Vincent Matossian wrote the first version of this function in R, thanks.

Arguments:

`graph`:

Pointer to an uninitialized graph object, the result will be stored here.

`m`:

Integer, `m`+1 is the number of letters in the alphabet.

`n`:

Integer, `n`+1 is the length of the strings.

Returns:

Error code.

See also:

`igraph_de_bruijn()`.

Time complexity: O(|V|* [(m+1)/m]^n +|E|), in practice it is more like O(|V|+|E|). |V| is the number of vertices, |E| is the number of edges and `m` and `n` are the corresponding arguments.

9.1.18. `igraph_extended_chordal_ring` — Create an extended chordal ring

```
int igraph_extended_chordal_ring(igraph_t *graph, igraph_integer_t nodes,
     const igraph_matrix_t *W);
```

An extended chordal ring is a regular graph, each node has the same degree. It can be obtained from a simple ring by adding some extra edges specified by a matrix. Let p denote the number of columns in the *W* matrix. The extra edges of vertex i are added according to column (i mod p) in *W*. The number of extra edges is the number of rows in *W*: for each row j an edge i->i+w[ij] is added if i+w[ij] is less than the number of total nodes.

See also Kotsis, G: Interconnection Topologies for Parallel Processing Systems, PARS Mitteilungen 11, 1-6, 1993.

Arguments:

graph:

Pointer to an uninitialized graph object, the result will be stored here. The result is always an undirected graph.

nodes:

Integer constant, the number of vertices in the graph. It must be at least 3.

W:

The matrix specifying the extra edges. The number of columns should divide the number of total vertices.

Returns:

Error code.

See also:

`igraph_ring()`.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

9.1.19. `igraph_connect_neighborhood` — Connects every vertex to its neighborhood

```
int igraph_connect_neighborhood(igraph_t *graph, igraph_integer_t order,
    igraph_neimode_t mode);
```

This function adds new edges to the input graph. For each vertex vertices reachable by at most *order* steps and not yet connected to the vertex a new edge is created.

Note that the input graph is modified in place, no new graph is created, call `igraph_copy()` if you want to keep the original graph as well.

For undirected graphs reachability is always symmetric: if vertex A can be reached from vertex B in at most *order* steps, then the opposite is also true. Only one undirected (A,B) edge will be added in this case.

Arguments:

graph:

The input graph, this is the output graph as well.

order:

Integer constant, it gives the distance within which the vertices will be connected to the source vertex.

mode:

Constant, it specifies how the neighborhood search is performed for directed graphs. If `IGRAPH_OUT` then vertices reachable from the source vertex will be connected, `IGRAPH_IN` is the opposite. If `IGRAPH_ALL` then the directed graph is considered as an undirected one.

Returns:

Error code.

See also:

`igraph_lattice()` uses this function to connect the neighborhood of the vertices.

Time complexity: O(|V|*d^o), |V| is the number of vertices in the graph, d is the average degree and o is the `order` argument.

9.2. Games: Randomized Graph Generators

Games are randomized graph generators. Randomization means that they generate a different graph every time you call them.

9.2.1. `igraph_grg_game` — Generating geometric random graphs.

```
int igraph_grg_game(igraph_t *graph, igraph_integer_t nodes,
      igraph_real_t radius, igraph_bool_t torus,
      igraph_vector_t *x, igraph_vector_t *y);
```

A geometric random graph is created by dropping points (=vertices) randomly to the unit square and then connecting all those pairs which are less than `radius` apart in Euclidean norm.

Original code contributed by Keith Briggs, thanks Keith.

Arguments:

`graph`:

Pointer to an uninitialized graph object,

`nodes`:

The number of vertices in the graph.

`radius`:

The radius within which the vertices will be connected.

`torus`:

Logical constant, if true periodic boundary conditions will be used, ie. the vertices are assumed to be on a torus instead of a square.

Returns:

Error code.

Time complexity: TODO, less than O(|V|^2+|E|).

Example 9-11. File examples/simple/igraph_grg_game.c

9.2.2. `igraph_barabasi_game` — Generates a graph based on the Barabási-Albert model.

```
int igraph_barabasi_game(igraph_t *graph, igraph_integer_t n,
    igraph_real_t power,
    igraph_integer_t m,
    const igraph_vector_t *outseq,
    igraph_bool_t outpref,
    igraph_real_t A,
    igraph_bool_t directed,
    igraph_barabasi_algorithm_t algo,
    const igraph_t *start_from);
```

Arguments:

graph:

An uninitialized graph object.

n:

The number of vertices in the graph.

power:

Power of the preferential attachment. The probability that a vertex is cited is proportional to d^power+A, where d is its degree (see also the *outpref* argument), power and A are given by arguments. In the classic preferential attachment model power=1.

m:

The number of outgoing edges generated for each vertex. (Only if *outseq* is `NULL`.)

outseq:

Gives the (out-)degrees of the vertices. If this is constant, this can be a NULL pointer or an empty (but initialized!) vector, in this case *m* contains the constant out-degree. The very first vertex has by definition no outgoing edges, so the first number in this vector is ignored.

outpref:

Boolean, if true not only the in- but also the out-degree of a vertex increases its citation probability. Ie. the citation probability is determined by the total degree of the vertices.

A:

The probability that a vertex is cited is proportional to d^power+A, where d is its degree (see also the *outpref* argument), power and A are given by arguments. In the previous versions of the function this parameter was implicitly set to one.

directed:

Boolean, whether to generate a directed graph.

algo:

The algorithm to use to generate the network. Possible values:

`IGRAPH_BARABASI_BAG`

This is the algorithm that was previously (before version 0.6) solely implemented in igraph. It works by putting the ids of the vertices into a bag (multiset, really), exactly as many times as their (in-)degree, plus once more. Then the required number of cited vertices are drawn from the bag, with replacement. This method might generate multiple edges. It only works if power=1 and A=1.

`IGRAPH_BARABASI_PSUMTREE`

This algorithm uses a partial prefix-sum tree to generate the graph. It does not generate multiple edges and works for any power and A values.

`IGRAPH_BARABASI_PSUMTREE_MULTIPLE`

This algorithm also uses a partial prefix-sum tree to generate the graph. The difference is, that now multiple edges are allowed. This method was implemented under the name `igraph_nonlinear_barabasi_game` before version 0.6.

start_from:

Either a null pointer, or a graph. In the latter case the graph as a starting configuration. The graph must be non-empty, i.e. it must have at least one vertex. If a graph is supplied here and the *outseq* argument is also given, then *outseq* should only contain information on the vertices that are not in the *start_from* graph.

Returns:

Error code: `IGRAPH_EINVAL`: invalid *n*, *m* or *outseq* parameter.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

Example 9-12. File examples/simple/igraph_barabasi_game.c

Example 9-13. File examples/simple/igraph_barabasi_game2.c

9.2.3. `igraph_erdos_renyi_game` — Generates a random (Erdos-Renyi) graph.

```
int igraph_erdos_renyi_game(igraph_t *graph, igraph_erdos_renyi_t type,
       igraph_integer_t n, igraph_real_t p_or_m,
       igraph_bool_t directed, igraph_bool_t loops);
```

Arguments:

graph:

Pointer to an uninitialized graph object.

type:

The type of the random graph, possible values:

`IGRAPH_ERDOS_RENYI_GNM`

G(n,m) graph, m edges are selected uniformly randomly in a graph with n vertices.

`IGRAPH_ERDOS_RENYI_GNP`

G(n,p) graph, every possible edge is included in the graph with probability p.

n:

The number of vertices in the graph.

p_or_m:

This is the p parameter for G(n,p) graphs and the m parameter for G(n,m) graphs.

directed:

Logical, whether to generate a directed graph.

loops:

Logical, whether to generate loops (self) edges.

Returns:

Error code: `IGRAPH_EINVAL`: invalid *type*, *n*, *p* or *m* parameter. `IGRAPH_ENOMEM`: there is not enough memory for the operation.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph.

See also:

`igraph_barabasi_game()`, `igraph_growing_random_game()`

Example 9-14. File examples/simple/igraph_erdos_renyi_game.c

9.2.4. `igraph_watts_strogatz_game` — The Watts-Strogatz small-world model

```
int igraph_watts_strogatz_game(igraph_t *graph, igraph_integer_t dim,
          igraph_integer_t size, igraph_integer_t nei,
          igraph_real_t p, igraph_bool_t loops,
          igraph_bool_t multiple);
```

This function generates a graph according to the Watts-Strogatz model of small-world networks. The graph is obtained by creating a circular undirected lattice and then rewire the edges randomly with a constant probability.

See also: Duncan J Watts and Steven H Strogatz: Collective dynamics of “small world” networks, Nature 393, 440-442, 1998.

Arguments:

`graph`:

The graph to initialize.

`dim`:

The dimension of the lattice.

`size`:

The size of the lattice along each dimension.

`nei`:

The size of the neighborhood for each vertex. This is the same as the `nei` argument of `igraph_connect_neighborhood()`.

`p`:

The rewiring probability. A real number between zero and one (inclusive).

`loops`:

Logical, whether to generate loop edges.

`multiple`:

Logical, whether to allow multiple edges in the generated graph.

Returns:

Error code.

See also:

`igraph_lattice()`, `igraph_connect_neighborhood()` and `igraph_rewire_edges()` can be used if more flexibility is needed, eg. a different type of lattice.

Time complexity: O(|V|*d^o+|E|), |V| ans |E| are the number of vertices and edges, d is the average degree, o is the `nei` argument.

9.2.5. `igraph_rewire_edges` — Rewire the edges of a graph with constant probability

```
int igraph_rewire_edges(igraph_t *graph, igraph_real_t prob,
   igraph_bool_t loops, igraph_bool_t multiple);
```

This function rewires the edges of a graph with a constant probability. More precisely each end point of each edge is rewired to an uniformly randomly chosen vertex with constant probability *prob*.

Note that this function modifies the input *graph*, call `igraph_copy()` if you want to keep it.

Arguments:

graph:

The input graph, this will be rewired, it can be directed or undirected.

prob:

The rewiring probability a constant between zero and one (inclusive).

loops:

Boolean, whether loop edges are allowed in the new graph, or not.

multiple:

Boolean, whether multiple edges are allowed in the new graph.

Returns:

Error code.

See also:

`igraph_watts_strogatz_game()` uses this function for the rewiring.

Time complexity: O(|V|+|E|).

9.2.6. `igraph_degree_sequence_game` — Generates a random graph with a given degree sequence

```
int igraph_degree_sequence_game(igraph_t *graph, const igraph_vector_t *out_deg,
    const igraph_vector_t *in_deg,
    igraph_degseq_t method);
```

Arguments:

graph:

Pointer to an uninitialized graph object.

out_deg:

The degree sequence for an undirected graph (if *in_seq* is of length zero), or the out-degree sequence of a directed graph (if *in_deq* is not of length zero.

in_deg:

It is either a zero-length vector or `NULL` (if an undirected graph is generated), or the in-degree sequence.

method:

The method to generate the graph. Possible values:

`IGRAPH_DEGSEQ_SIMPLE`

For undirected graphs, this method puts all vertex ids in a bag such that the multiplicity of a vertex in the bag is the same as its degree. Then it draws pairs from the bag until the bag becomes empty. This method can generate both loop (self) edges and multiple edges. For directed graphs, the algorithm is basically the same, but two separate bags are used for the in- and out-degrees.

`IGRAPH_DEGSEQ_SIMPLE_NO_MULTIPLE`

This method is similar to `IGRAPH_DEGSEQ_SIMPLE` but tries to avoid multiple and loop edges and restarts the generation from scratch if it gets stuck. It is not guaranteed to sample uniformly from the space of all possible graphs with the given sequence, but it is relatively fast and it will eventually succeed if the provided degree sequence is graphical, but there is no upper bound on the number of iterations.

`IGRAPH_DEGSEQ_VL`

This method is a much more sophisticated generator than the previous ones. It can sample undirected, connected simple graphs uniformly and uses Monte-Carlo methods to randomize the graphs. This generator should be favoured if undirected and connected graphs are to be generated and execution time is not a concern. igraph uses the original implementation of Fabien Viger; see http://www-rp.lip6.fr/~latapy/FV/generation.html and the paper cited on it for the details of the algorithm.

Returns:

Error code: `IGRAPH_ENOMEM`: there is not enough memory to perform the operation. `IGRAPH_EINVAL`: invalid method parameter, or invalid in- and/or out-degree vectors. The degree vectors should be non-negative, *`out_deg`* should sum up to an even integer for undirected graphs; the length and sum of *`out_deg`* and *`in_deg`* should match for directed graphs.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges for `IGRAPH_DEGSEQ_SIMPLE`. The time complexity of the other modes is not known.

See also:

`igraph_barabasi_game()`, `igraph_erdos_renyi_game()`, `igraph_is_degree_sequence()`, `igraph_is_graphical_degree_sequence()`

Example 9-15. File examples/simple/igraph_degree_sequence_game.c

9.2.7. `igraph_k_regular_game` — Generates a random graph where each vertex has the same degree.

```
int igraph_k_regular_game(igraph_t *graph,
    igraph_integer_t no_of_nodes, igraph_integer_t k,
    igraph_bool_t directed, igraph_bool_t multiple);
```

This game generates a directed or undirected random graph where the degrees of vertices are equal to a predefined constant k. For undirected graphs, at least one of k and the number of vertices must be even.

The game simply uses `igraph_degree_sequence_game` with appropriately constructed degree sequences.

Arguments:

graph:

Pointer to an uninitialized graph object.

no_of_nodes:

The number of nodes in the generated graph.

k:

The degree of each vertex in an undirected graph, or the out-degree and in-degree of each vertex in a directed graph.

directed:

Whether the generated graph will be directed.

multiple:

Whether to allow multiple edges in the generated graph.

Returns:

Error code: `IGRAPH_EINVAL`: invalid parameter; e.g., negative number of nodes, or odd number of nodes and odd k for undirected graphs. `IGRAPH_ENOMEM`: there is not enough memory for the operation.

Time complexity: O(|V|+|E|) if `multiple` is true, otherwise not known.

9.2.8. `igraph_static_fitness_game` — Generates a non-growing random graph with edge probabilities

```
int igraph_static_fitness_game(igraph_t *graph, igraph_integer_t no_of_edges,
                igraph_vector_t* fitness_out, igraph_vector_t* fitness_in,
                igraph_bool_t loops, igraph_bool_t multiple);
```

proportional to node fitness scores. This game generates a directed or undirected random graph where the probability of an edge between vertices i and j depends on the fitness scores of the two vertices involved. For undirected graphs, each vertex has a single fitness score. For directed graphs, each vertex has an out-and an in-fitness, and the probability of an edge from i to j depends on the out-fitness of vertex i and the in-fitness of vertex j.

The generation process goes as follows. We start from N disconnected nodes (where N is given by the length of the fitness vector). Then we randomly select two vertices i and j, with probabilities proportional to their fitnesses. (When the generated graph is directed, i is selected according to the out-fitnesses and j is selected according to the in-fitnesses). If the vertices are not connected yet (or if multiple edges are allowed), we connect them; otherwise we select a new pair. This is repeated until the desired number of links are created.

It can be shown that the *expected* degree of each vertex will be proportional to its fitness, although the actual, observed degree will not be. If you need to generate a graph with an exact degree sequence, consider `igraph_degree_sequence_game` instead.

This model is commonly used to generate static scale-free networks. To achieve this, you have to draw the fitness scores from the desired power-law distribution. Alternatively, you may use `igraph_static_power_law_game` which generates the fitnesses for you with a given exponent.

Reference: Goh K-I, Kahng B, Kim D: Universal behaviour of load distribution in scale-free networks. Phys Rev Lett 87(27):278701, 2001.

Arguments:

`graph`:

Pointer to an uninitialized graph object.

`fitness_out`:

A numeric vector containing the fitness of each vertex. For directed graphs, this specifies the out-fitness of each vertex.

`fitness_in`:

If `NULL`, the generated graph will be undirected. If not `NULL`, this argument specifies the in-fitness of each vertex.

`no_of_edges`:

The number of edges in the generated graph.

`loops`:

Whether to allow loop edges in the generated graph.

`multiple`:

Whether to allow multiple edges in the generated graph.

Returns:

Error code: `IGRAPH_EINVAL`: invalid parameter `IGRAPH_ENOMEM`: there is not enough memory for the operation.

Time complexity: O(|V| + |E| log |E|).

9.2.9. `igraph_static_power_law_game` — Generates a non-growing random graph with expected power-law degree distributions.

```
int igraph_static_power_law_game(igraph_t *graph,
    igraph_integer_t no_of_nodes, igraph_integer_t no_of_edges,
    igraph_real_t exponent_out, igraph_real_t exponent_in,
    igraph_bool_t loops, igraph_bool_t multiple,
    igraph_bool_t finite_size_correction);
```

This game generates a directed or undirected random graph where the degrees of vertices follow power-law distributions with prescribed exponents. For directed graphs, the exponents of the in- and out-degree distributions may be specified separately.

The game simply uses `igraph_static_fitness_game` with appropriately constructed fitness vectors. In particular, the fitness of vertex i is i^{-alpha}, where alpha = 1/(gamma-1) and gamma is the exponent given in the arguments.

To remove correlations between in- and out-degrees in case of directed graphs, the in-fitness vector will be shuffled after it has been set up and before `igraph_static_fitness_game` is called.

Note that significant finite size effects may be observed for exponents smaller than 3 in the original formulation of the game. This function provides an argument that lets you remove the finite size effects by assuming that the fitness of vertex i is $(i+i0-1)^{-alpha}$, where i0 is a constant chosen appropriately to ensure that the maximum degree is less than the square root of the number of edges times the average degree; see the paper of Chung and Lu, and Cho et al for more details.

References:

Goh K-I, Kahng B, Kim D: Universal behaviour of load distribution in scale-free networks. Phys Rev Lett 87(27):278701, 2001.

Chung F and Lu L: Connected components in a random graph with given degree sequences. Annals of Combinatorics 6, 125-145, 2002.

Cho YS, Kim JS, Park J, Kahng B, Kim D: Percolation transitions in scale-free networks under the Achlioptas process. Phys Rev Lett 103:135702, 2009.

Arguments:

graph:

Pointer to an uninitialized graph object.

no_of_nodes:

The number of nodes in the generated graph.

no_of_edges:

The number of edges in the generated graph.

exponent_out:

The power law exponent of the degree distribution. For directed graphs, this specifies the exponent of the out-degree distribution. It must be greater than or equal to 2. If you pass `IGRAPH_INFINITY` here, you will get back an Erdos-Renyi random network.

exponent_in:

If negative, the generated graph will be undirected. If greater than or equal to 2, this argument specifies the exponent of the in-degree distribution. If non-negative but less than 2, an error will be generated.

loops:

Whether to allow loop edges in the generated graph.

multiple:

Whether to allow multiple edges in the generated graph.

finite_size_correction:

Whether to use the proposed finite size correction of Cho et al.

Returns:

Error code: `IGRAPH_EINVAL`: invalid parameter `IGRAPH_ENOMEM`: there is not enough memory for the operation.

Time complexity: O(|V| + |E| log |E|).

9.2.10. `igraph_forest_fire_game` — Generates a network according to the “forest fire game”

```
int igraph_forest_fire_game(igraph_t *graph, igraph_integer_t nodes,
       igraph_real_t fw_prob, igraph_real_t bw_factor,
       igraph_integer_t pambs, igraph_bool_t directed);
```

The forest fire model intends to reproduce the following network characteristics, observed in real networks:

- Heavy-tailed in-degree distribution.
- Heavy-tailed out-degree distribution.
- Communities.
- Densification power-law. The network is densifying in time, according to a power-law rule.
- Shrinking diameter. The diameter of the network decreases in time.

The network is generated in the following way. One vertex is added at a time. This vertex connects to (cites) `ambs` vertices already present in the network, chosen uniformly random. Now, for each cited vertex `v` we do the following procedure:

1. We generate two random number, `x` and `y` , that are geometrically distributed with means `p/(1-p)` and `rp(1-rp)` . (`p` is `fw_prob` , `r` is `bw_factor` .) The new vertex cites `x` outgoing neighbors and `y` incoming neighbors of `v` , from those which are not yet cited by the new vertex. If there are less than `x` or `y` such vertices available then we cite all of them.
2. The same procedure is applied to all the newly cited vertices.

See also: Jure Leskovec, Jon Kleinberg and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. *KDD ’05: Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining* , 177--187, 2005.

Note however, that the version of the model in the published paper is incorrect in the sense that it cannot generate the kind of graphs the authors claim. A corrected version is available from http://www.cs.cmu.edu/~jure/pubs/powergrowth-tkdd.pdf, our implementation is based on this.

Arguments:

`graph`:

Pointer to an uninitialized graph object.

nodes:

The number of vertices in the graph.

fw_prob:

The forward burning probability.

bw_factor:

The backward burning ratio. The backward burning probability is calculated as `bw.factor*fw.prob`.

pambs:

The number of ambassador vertices.

directed:

Whether to create a directed graph.

Returns:

Error code.

Time complexity: TODO.

9.2.11. `igraph_rewire` — Randomly rewires a graph while preserving the degree distribution.

```
int igraph_rewire(igraph_t *graph, igraph_integer_t n, igraph_rewiring_t mode);
```

This function generates a new graph based on the original one by randomly rewiring edges while preserving the original graph's degree distribution. Please note that the rewiring is done "in place", so no new graph will be allocated. If you would like to keep the original graph intact, use `igraph_copy()` beforehand.

Arguments:

graph:

The graph object to be rewired.

n:

Number of rewiring trials to perform.

mode:

The rewiring algorithm to be used. It can be one of the following:

`IGRAPH_REWIRING_SIMPLE`

Simple rewiring algorithm which chooses two arbitrary edges in each step (namely (a,b) and (c,d)) and substitutes them with (a,d) and (c,b) if they don't exist. The method will neither destroy nor create self-loops. Undirected edges may be chosen for rewiring in either direction.

`IGRAPH_REWIRING_SIMPLE_LOOPS`

Same as `IGRAPH_REWIRING_SIMPLE` but allows the creation or destruction of self-loops.

Returns:

Error code:

`IGRAPH_EINVMODE`

Invalid rewiring mode.

`IGRAPH_EINVAL`

Graph unsuitable for rewiring (e.g. it has less than 4 nodes in case of `IGRAPH_REWIRING_SIMPLE`)

`IGRAPH_ENOMEM`

Not enough memory for temporary data.

Time complexity: TODO.

Example 9-16. File examples/simple/igraph_rewire.c

9.2.12. `igraph_growing_random_game` — Generates a growing random graph.

```
int igraph_growing_random_game(igraph_t *graph, igraph_integer_t n,
          igraph_integer_t m, igraph_bool_t directed,
          igraph_bool_t citation);
```

This function simulates a growing random graph. In each discrete time step a new vertex is added and a number of new edges are also added. These graphs are known to be different from standard (not growing) random graphs.

Arguments:

graph:

Uninitialized graph object.

n:

The number of vertices in the graph.

m:

The number of edges to add in a time step (ie. after adding a vertex).

directed:

Boolean, whether to generate a directed graph.

citation:

Boolean, if `TRUE`, the edges always originate from the most recently added vertex.

Returns:

Error code: `IGRAPH_EINVAL`: invalid *n* or *m* parameter.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

Example 9-17. File examples/simple/igraph_growing_random_game.c

9.2.13. `igraph_callaway_traits_game` — Simulate a growing network with vertex types.

```
int igraph_callaway_traits_game (igraph_t *graph, igraph_integer_t nodes,
    igraph_integer_t types, igraph_integer_t edges_per_step,
    igraph_vector_t *type_dist,
    igraph_matrix_t *pref_matrix,
    igraph_bool_t directed);
```

The different types of vertices prefer to connect other types of vertices with a given probability.

The simulation goes like this: in each discrete time step a new vertex is added to the graph. The type of this vertex is generated based on *type_dist*. Then two vertices are selected uniformly randomly from the graph. The probability that they will be connected depends on the types of these vertices and is taken from *pref_matrix*. Then another two vertices are selected and this is repeated *edges_per_step* times in each time step.

Arguments:

graph:

Pointer to an uninitialized graph.

nodes:

The number of nodes in the graph.

types:

Number of node types.

edges_per_step:

The number of edges to be add per time step.

type_dist:

Vector giving the distribution of the vertex types.

pref_matrix:

Matrix giving the connection probabilities for the vertex types.

directed:

Logical, whether to generate a directed graph.

Returns:

Error code.

Added in version 0.2.

Time complexity: O(|V|e*log(|V|)), |V| is the number of vertices, e is *edges_per_step*.

9.2.14. `igraph_establishment_game` — Generates a graph with a simple growing model with vertex types.

```
int igraph_establishment_game(igraph_t *graph, igraph_integer_t nodes,
         igraph_integer_t types, igraph_integer_t k,
         igraph_vector_t *type_dist,
         igraph_matrix_t *pref_matrix,
         igraph_bool_t directed);
```

The simulation goes like this: a single vertex is added at each time step. This new vertex tries to connect to *k* vertices in the graph. The probability that such a connection is realized depends on the types of the vertices involved.

Arguments:

graph:

Pointer to an uninitialized graph.

nodes:

The number of vertices in the graph.

types:

The number of vertex types.

k:

The number of connections tried in each time step.

type_dist:

Vector giving the distribution of vertex types.

pref_matrix:

Matrix giving the connection probabilities for different vertex types.

directed:

Logical, whether to generate a directed graph.

Returns:

Error code.

Added in version 0.2.

Time complexity: O(|V|*k*log(|V|)), |V| is the number of vertices and k is the `k` parameter.

9.2.15. `igraph_preference_game` — Generates a graph with vertex types and connection preferences

```
int igraph_preference_game(igraph_t *graph, igraph_integer_t nodes,
      igraph_integer_t types,
      const igraph_vector_t *type_dist,
      igraph_bool_t fixed_sizes,
      const igraph_matrix_t *pref_matrix,
      igraph_vector_t *node_type_vec,
      igraph_bool_t directed,
      igraph_bool_t loops);
```

This is practically the nongrowing variant of `igraph_establishment_game`. A given number of vertices are generated. Every vertex is assigned to a vertex type according to the given type probabilities. Finally, every vertex pair is evaluated and an edge is created between them with a probability depending on the types of the vertices involved.

In other words, this function generates a graph according to a block-model. Vertices are divided into groups (or blocks), and the probability the two vertices are connected depends on their groups only.

Arguments:

graph:

Pointer to an uninitialized graph.

nodes:

The number of vertices in the graph.

types:

The number of vertex types.

type_dist:

Vector giving the distribution of vertex types. If `NULL`, all vertex types will have equal probability. See also the `fixed_sizes` argument.

fixed_sizes:

Boolean. If true, then the number of vertices with a given vertex type is fixed and the `type_dist` argument gives these numbers for each vertex type. If true, and `type_dist` is `NULL`, then the function tries to make vertex groups of the same size. If this is not possible, then some groups will have an extra vertex.

pref_matrix:

Matrix giving the connection probabilities for different vertex types. This should be symmetric if the requested graph is undirected.

node_type_vec:

A vector where the individual generated vertex types will be stored. If `NULL` , the vertex types won't be saved.

directed:

Logical, whether to generate a directed graph. If undirected graphs are requested, only the lower left triangle of the preference matrix is considered.

loops:

Logical, whether loop edges are allowed.

Returns:

Error code.

Added in version 0.3.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph.

See also:

igraph_establishment_game()

Example 9-18. File examples/simple/igraph_preference_game.c

9.2.16. `igraph_asymmetric_preference_game` — Generates a graph with asymmetric vertex types and connection preferences

```
int igraph_asymmetric_preference_game(igraph_t *graph, igraph_integer_t nodes,
          igraph_integer_t types,
          igraph_matrix_t *type_dist_matrix,
          igraph_matrix_t *pref_matrix,
          igraph_vector_t *node_type_in_vec,
          igraph_vector_t *node_type_out_vec,
          igraph_bool_t loops);
```

This is the asymmetric variant of `igraph_preference_game()` . A given number of vertices are generated. Every vertex is assigned to an "incoming" and an "outgoing" vertex type according to the given joint type probabilities. Finally, every vertex pair is evaluated and a directed edge is created between them with a probability depending on the "outgoing" type of the source vertex and the "incoming" type of the target vertex.

Arguments:

`graph`:

Pointer to an uninitialized graph.

`nodes`:

The number of vertices in the graph.

`types`:

The number of vertex types.

type_dist_matrix:

Matrix giving the joint distribution of vertex types. If null, incoming and outgoing vertex types are independent and uniformly distributed.

pref_matrix:

Matrix giving the connection probabilities for different vertex types.

node_type_in_vec:

A vector where the individual generated "incoming" vertex types will be stored. If NULL, the vertex types won't be saved.

node_type_out_vec:

A vector where the individual generated "outgoing" vertex types will be stored. If NULL, the vertex types won't be saved.

loops:

Logical, whether loop edges are allowed.

Returns:

Error code.

Added in version 0.3.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph.

See also:

`igraph_preference_game()`

9.2.17. `igraph_recent_degree_game` — Stochastic graph generator based on the number of incident edges a node has gained recently

```
int igraph_recent_degree_game(igraph_t *graph, igraph_integer_t n,
          igraph_real_t power,
          igraph_integer_t window,
          igraph_integer_t m,
          const igraph_vector_t *outseq,
```

```
igraph_bool_t outpref,
igraph_real_t zero_appeal,
igraph_bool_t directed);
```

Arguments:

graph:

Pointer to an uninitialized graph object.

n:

The number of vertices in the graph, this is the same as the number of time steps.

power:

The exponent, the probability that a node gains a new edge is proportional to the number of edges it has gained recently (in the last *window* time steps) to *power*.

window:

Integer constant, the size of the time window to use to count the number of recent edges.

m:

Integer constant, the number of edges to add per time step if the *outseq* parameter is a null pointer or a zero-length vector.

outseq:

The number of edges to add in each time step. This argument is ignored if it is a null pointer or a zero length vector, is this case the constant *m* parameter is used.

outpref:

Logical constant, if true the edges originated by a vertex also count as recent incident edges. It is false in most cases.

zero_appeal:

Constant giving the attractiveness of the vertices which haven't gained any edge recently.

directed:

Logical constant, whether to generate a directed graph.

Returns:

Error code.

Time complexity: O(|V|*log(|V|)+|E|), |V| is the number of vertices, |E| is the number of edges in the graph.

9.2.18. `igraph_barabasi_aging_game` — Preferential attachment with aging of vertices

```
int igraph_barabasi_aging_game(igraph_t *graph,
          igraph_integer_t nodes,
          igraph_integer_t m,
          const igraph_vector_t *outseq,
          igraph_bool_t outpref,
          igraph_real_t pa_exp,
          igraph_real_t aging_exp,
          igraph_integer_t aging_bin,
          igraph_real_t zero_deg_appeal,
          igraph_real_t zero_age_appeal,
          igraph_real_t deg_coef,
          igraph_real_t age_coef,
          igraph_bool_t directed);
```

In this game, the probability that a node gains a new edge is given by its (in-)degree (k) and age (l). This probability has a degree dependent component multiplied by an age dependent component. The degree dependent part is: *`deg_coef`* times k to the power of *`pa_exp`* plus *`zero_deg_appeal`*; and the age dependent part is *`age_coef`* times l to the power of *`aging_exp`* plus *`zero_age_appeal`*.

The age is based on the number of vertices in the network and the *`aging_bin`* argument: vertices grew one unit older after each *`aging_bin`* vertices added to the network.

Arguments:

`graph`:

Pointer to an uninitialized graph object.

`nodes`:

The number of vertices in the graph.

`m`:

The number of edges to add in each time step. If the *`outseq`* argument is not a null vector and not a zero-length vector.

outseq:

The number of edges to add in each time step. If it is a null pointer or a zero-length vector then it is ignored and the *m* argument is used instead.

outpref:

Logical constant, whether the edges initiated by a vertex contribute to the probability to gain a new edge.

pa_exp:

The exponent of the preferential attachment, a small positive number usually, the value 1 yields the classic linear preferential attachment.

aging_exp:

The exponent of the aging, this is a negative number usually.

aging_bin:

Integer constant, the number of vertices to add before vertices in the network grew one unit older.

zero_deg_appeal:

The degree dependent part of the attractiveness of the zero degree vertices.

zero_age_appeal:

The age dependent part of the attractiveness of the vertices of age zero. This parameter is usually zero.

deg_coef:

The coefficient for the degree.

age_coef:

The coefficient for the age.

directed:

Logical constant, whether to generate a directed graph.

Returns:

Error code.

Time complexity: O((|V|+|V|/aging_bin)*log(|V|)+|E|). |V| is the number of vertices, |E| the number of edges.

9.2.19. `igraph_recent_degree_aging_game` — Preferential attachment based on the number of edges gained recently, with aging of vertices

```
int igraph_recent_degree_aging_game(igraph_t *graph,
        igraph_integer_t nodes,
        igraph_integer_t m,
        const igraph_vector_t *outseq,
        igraph_bool_t outpref,
        igraph_real_t pa_exp,
        igraph_real_t aging_exp,
        igraph_integer_t aging_bin,
        igraph_integer_t time_window,
        igraph_real_t zero_appeal,
        igraph_bool_t directed);
```

This game is very similar to `igraph_barabasi_aging_game()`, except that instead of the total number of incident edges the number of edges gained in the last *time_window* time steps are counted.

The degree dependent part of the attractiveness is given by k to the power of *pa_exp* plus *zero_appeal*; the age dependent part is l to the power to *aging_exp*. k is the number of edges gained in the last *time_window* time steps, l is the age of the vertex.

Arguments:

graph:

Pointer to an uninitialized graph object.

nodes:

The number of vertices in the graph.

m:

The number of edges to add in each time step. If the *outseq* argument is not a null vector or a zero-length vector then it is ignored.

outseq:

Vector giving the number of edges to add in each time step. If it is a null pointer or a zero-length vector then it is ignored and the *m* argument is used.

outpref:

Logical constant, if true the edges initiated by a vertex are also counted. Normally it is false.

pa_exp:

The exponent for the preferential attachment.

aging_exp:

The exponent for the aging, normally it is negative: old vertices gain edges with less probability.

aging_bin:

Integer constant, gives the scale of the aging. The age of the vertices is incremented by one after every *aging_bin* vertex added.

time_window:

The time window to use to count the number of incident edges for the vertices.

zero_appeal:

The degree dependent part of the attractiveness for zero degree vertices.

directed:

Logical constant, whether to create a directed graph.

Returns:

Error code.

Time complexity: O((|V|+|V|/aging_bin)*log(|V|)+|E|). |V| is the number of vertices, |E| the number of edges.

9.2.20. `igraph_cited_type_game` — Simulate a citation based on vertex types.

```
int igraph_cited_type_game(igraph_t *graph, igraph_integer_t nodes,
      const igraph_vector_t *types,
      const igraph_vector_t *pref,
      igraph_integer_t edges_per_step,
      igraph_bool_t directed);
```

Function to create a network based on some vertex categories. This function creates a citation network, in each step a single vertex and *edges_per_step* citating edges are added, nodes with different categories (may) have different probabilities to get cited, as given by the *pref* vector.

Note that this function might generate networks with multiple edges if *edges_per_step* is greater than one. You might want to call `igraph_simplify()` on the result to remove multiple edges.

Arguments:

graph:

Pointer to an uninitialized graph object.

nodes:

The number of vertices in the network.

types:

Numeric vector giving the categories of the vertices, so it should contain *nodes* non-negative integer numbers. Types are numbered from zero.

pref:

The attractivity of the different vertex categories in a vector. Its length should be the maximum element in *types* plus one (types are numbered from zero).

edges_per_step:

Integer constant, the number of edges to add in each time step.

directed:

Logical constant, whether to create a directed network.

Returns:

Error code.

See also:

`igraph_citing_cited_type_game()` for a bit more general game.

Time complexity: O((|V|+|E|)log|V|), |V| and |E| are number of vertices and edges, respectively.

9.2.21. `igraph_citing_cited_type_game` — Simulate a citation network based on vertex types.

```
int igraph_citing_cited_type_game(igraph_t *graph, igraph_integer_t nodes,
      const igraph_vector_t *types,
```

```
const igraph_matrix_t *pref,
igraph_integer_t edges_per_step,
igraph_bool_t directed);
```

This game is similar to `igraph_cited_type_game()` but here the category of the citing vertex is also considered.

An evolving citation network is modeled here, a single vertex and its *`edges_per_step`* citation are added in each time step. The odds the a given vertex is cited by the new vertex depends on the category of both the citing and the cited vertex and is given in the *`pref`* matrix. The categories of the citing vertex correspond to the rows, the categories of the cited vertex to the columns of this matrix. Ie. the element in row `i` and column `j` gives the probability that a `j` vertex is cited, if the category of the citing vertex is `i`.

Note that this function might generate networks with multiple edges if *`edges_per_step`* is greater than one. You might want to call `igraph_simplify()` on the result to remove multiple edges.

Arguments:

`graph`:

Pointer to an uninitialized graph object.

`nodes`:

The number of vertices in the network.

`types`:

A numeric matrix of length *`nodes`*, containing the categories of the vertices. The categories are numbered from zero.

`pref`:

The preference matrix, a square matrix is required, both the number of rows and columns should be the maximum element in *`types`* plus one (types are numbered from zero).

`directed`:

Logical constant, whether to create a directed network.

Returns:

Error code.

Time complexity: O((|V|+|E|)log|V|), |V| and |E| are number of vertices and edges, respectively.

Chapter 10. Games on Graphs

10.1. Microscopic Update Rules

10.1.1. `igraph_deterministic_optimal_imitation` — Adopt a strategy via deterministic optimal imitation.

```
int igraph_deterministic_optimal_imitation(const igraph_t *graph,
                                           igraph_integer_t vid,
                                           igraph_optimal_t optimality,
                                           const igraph_vector_t *quantities,
                                           igraph_vector_t *strategies,
                                           igraph_neimode_t mode);
```

A simple deterministic imitation strategy where a vertex revises its strategy to that which yields a local optimal. Here "local" is with respect to the immediate neighbours of the vertex. The vertex retains its current strategy where this strategy yields a locally optimal quantity. The quantity in this case could be a measure such as fitness.

Arguments:

`graph`:

The graph object representing the game network. This cannot be the empty or trivial graph, but must have at least two vertices and one edge. If *`graph`* has one vertex, then no strategy update would take place. Furthermore, if *`graph`* has at least two vertices but zero edges, then strategy update would also not take place.

`vid`:

The vertex whose strategy is to be updated. It is assumed that *`vid`* represents a vertex in *`graph`*. No checking is performed and it is your responsibility to ensure that *`vid`* is indeed a vertex of *`graph`*. If an isolated vertex is provided, i.e. the input vertex has degree 0, then no strategy update would take place and *`vid`* would retain its current strategy. Strategy update would also not take place if the local neighbourhood of *`vid`* are its in-neighbours (respectively out-neighbours), but *`vid`* has zero in-neighbours (respectively out-neighbours). Loops are ignored in computing the degree (in, out, all) of *`vid`*.

optimality:

Logical; controls the type of optimality to be used. Supported values are:

IGRAPH_MAXIMUM

Use maximum deterministic imitation, where the strategy of the vertex with maximum quantity (e.g. fitness) would be adopted. We update the strategy of *vid* to that which yields a local maximum.

IGRAPH_MINIMUM

Use minimum deterministic imitation. That is, the strategy of the vertex with minimum quantity would be imitated. In other words, update to the strategy that yields a local minimum.

quantities:

A vector of quantities providing the quantity of each vertex in *graph*. Think of each entry of the vector as being generated by a function such as the fitness function for the game. So if the vector represents fitness quantities, then each vector entry is the fitness of some vertex. The length of this vector must be the same as the number of vertices in the vertex set of *graph*.

strategies:

A vector of the current strategies for the vertex population. The updated strategy for *vid* would be stored here. Each strategy is identified with a nonnegative integer, whose interpretation depends on the payoff matrix of the game. Generally we use the strategy ID as a row or column index of the payoff matrix. The length of this vector must be the same as the number of vertices in the vertex set of *graph*.

mode:

Defines the sort of neighbourhood to consider for *vid*. If *graph* is undirected, then we use all the immediate neighbours of *vid*. Thus if you know that *graph* is undirected, then it is safe to pass the value *IGRAPH_ALL* here. Supported values are:

IGRAPH_OUT

Use the out-neighbours of *vid*. This option is only relevant when *graph* is a directed graph.

IGRAPH_IN

Use the in-neighbours of *vid*. Again this option is only relevant when *graph* is a directed graph.

IGRAPH_ALL

Use both the in- and out-neighbours of *vid*. This option is only relevant if *graph* is a digraph. Also use this value if *graph* is undirected.

Returns:

The error code *IGRAPH_EINVAL* is returned in each of the following cases: (1) Any of the parameters *graph*, *quantities*, or *strategies* is a null pointer. (2) The vector *quantities* or *strategies* has a length different from the number of vertices in *graph*. (3) The parameter *graph* is the empty or null graph, i.e. the graph with zero vertices and edges.

Time complexity: O(2d), where d is the degree of the vertex *vid*.

Example 10-1. File examples/simple/igraph_deterministic_optimal_imitation.c

10.1.2. `igraph_moran_process` — The Moran process in a network setting.

```
int igraph_moran_process(const igraph_t *graph,
                         const igraph_vector_t *weights,
                         igraph_vector_t *quantities,
                         igraph_vector_t *strategies,
                         igraph_neimode_t mode);
```

This is an extension of the classic Moran process to a network setting. The Moran process is a model of haploid (asexual) reproduction within a population having a fixed size. In the network setting, the Moran process operates on a weighted graph. At each time step a vertex a is chosen for reproduction and another vertex b is chosen for death. Vertex a gives birth to an identical clone c, which replaces b. Vertex c is a clone of a in that c inherits both the current quantity (e.g. fitness) and current strategy of a.

The graph G representing the game network is assumed to be simple, i.e. free of loops and without multiple edges. If, on the other hand, G has a loop incident on some vertex v, then it is possible that when v is chosen for reproduction it would forgo this opportunity. In particular, when v is chosen for reproduction and v is also chosen for death, the clone of v would be v itself with its current vertex ID. In effect v forgoes its chance for reproduction.

Arguments:

graph:

The graph object representing the game network. This cannot be the empty or trivial graph, but must have at least two vertices and one edge. The Moran process will not take place in each of the following cases: (1) If *graph* has one vertex. (2) If *graph* has at least two vertices but zero edges.

weights:

A vector of all edge weights for *graph*. Thus weights[i] means the weight of the edge with edge ID i. For the purpose of the Moran process, each weight is assumed to be positive; it is your responsibility to ensure this condition holds. The length of this vector must be the same as the number of edges in *graph*.

quantities:

A vector of quantities providing the quantity of each vertex in *graph*. The quantity of the new clone will be stored here. Think of each entry of the vector as being generated by a function such as the fitness function for the game. So if the vector represents fitness quantities, then each vector entry is the fitness of some vertex. The length of this vector must be the same as the number of vertices in the vertex set of *graph*. For the purpose of the Moran process, each vector entry is assumed to be nonnegative; no checks will be performed for this. It is your responsibility to ensure that at least one entry is positive. Furthermore, this vector cannot be a vector of zeros; this condition will be checked.

strategies:

A vector of the current strategies for the vertex population. The strategy of the new clone will be stored here. Each strategy is identified with a nonnegative integer, whose interpretation depends on the payoff matrix of the game. Generally we use the strategy ID as a row or column index of the payoff matrix. The length of this vector must be the same as the number of vertices in the vertex set of *graph*.

mode:

Defines the sort of neighbourhood to consider for the vertex a chosen for reproduction. This is only relevant if *graph* is directed. If *graph* is undirected, then it is safe to pass the value *IGRAPH_ALL* here. Supported values are:

`IGRAPH_OUT`

Use the out-neighbours of a. This option is only relevant when *graph* is directed.

`IGRAPH_IN`

Use the in-neighbours of a. Again this option is only relevant when *graph* is directed.

`IGRAPH_ALL`

Use both the in- and out-neighbours of a. This option is only relevant if *graph* is directed. Also use this value if *graph* is undirected.

Returns:

The error code *IGRAPH_EINVAL* is returned in each of the following cases: (1) Any of the parameters *graph*, *weights*, *quantities* or *strategies* is a null pointer. (2) The vector *quantities* or *strategies* has a length different from the number of vertices in *graph*. (3) The vector *weights* has a length different from the number of edges in *graph*. (4) The parameter *graph* is the empty or null graph, i.e. the graph with zero vertices and edges. (5) The vector *weights*, or the combination of interest, sums to zero. (6) The vector *quantities*, or the combination of interest, sums to zero.

Time complexity: depends on the random number generator, but is usually O(n) where n is the number of vertices in *graph*.

References:

(Lieberman et al. 2005)

E. Lieberman, C. Hauert, and M. A. Nowak. Evolutionary dynamics on graphs. *Nature,* 433(7023):312--316, 2005.

(Moran 1958)

P. A. P. Moran. Random processes in genetics. *Mathematical Proceedings of the Cambridge Philosophical Society,* 54(1):60--71, 1958.

Example 10-2. File examples/simple/igraph_moran_process.c

10.1.3. `igraph_roulette_wheel_imitation` — Adopt a strategy via roulette wheel selection.

```
int igraph_roulette_wheel_imitation(const igraph_t *graph,
                                    igraph_integer_t vid,
                                    igraph_bool_t islocal,
                                    const igraph_vector_t *quantities,
                                    igraph_vector_t *strategies,
                                    igraph_neimode_t mode);
```

A simple stochastic imitation strategy where a vertex revises its strategy to that of a vertex u chosen proportionate to u's quantity (e.g. fitness). This is a special case of stochastic imitation, where a candidate is not chosen uniformly at random but proportionate to its quantity.

Arguments:

graph:

The graph object representing the game network. This cannot be the empty or trivial graph, but must have at least two vertices and one edge. If *graph* has one vertex, then no strategy update would take place. Furthermore, if *graph* has at least two vertices but zero edges, then strategy update would also not take place.

vid:

The vertex whose strategy is to be updated. It is assumed that *vid* represents a vertex in *graph*. No checking is performed and it is your responsibility to ensure that *vid* is indeed a vertex of *graph*. If an isolated vertex is provided, i.e. the input vertex has degree 0, then no strategy update would take place and *vid* would retain its current strategy. Strategy update would also not take place if the local neighbourhood of *vid* are its in-neighbours (respectively out-neighbours), but *vid* has zero in-neighbours (respectively out-neighbours). Loops are ignored in computing the degree (in, out, all) of *vid*.

islocal:

Boolean; this flag controls which perspective to use in computing the relative quantity. If true then we use the local perspective; otherwise we use the global perspective. The local perspective for *vid* is the set of all immediate neighbours of *vid*. In contrast, the global perspective for *vid* is the vertex set of *graph*.

quantities:

A vector of quantities providing the quantity of each vertex in *graph*. Think of each entry of the vector as being generated by a function such as the fitness function for the game. So if the vector represents fitness quantities, then each vector entry is the fitness of some vertex. The length of this vector must be the same as the number of vertices in the vertex set of *graph*. For the purpose of roulette wheel selection, each vector entry is assumed to be nonnegative; no checks will be performed for this. It is your responsibility to ensure that at least one entry is nonzero. Furthermore, this vector cannot be a vector of zeros; this condition will be checked.

strategies:

A vector of the current strategies for the vertex population. The updated strategy for *vid* would be stored here. Each strategy is identified with a nonnegative integer, whose interpretation depends on the payoff matrix of the game. Generally we use the strategy ID as a row or column index of the payoff matrix. The length of this vector must be the same as the number of vertices in the vertex set of *graph*.

mode:

Defines the sort of neighbourhood to consider for *vid*. This is only relevant if we are considering the local perspective, i.e. if *islocal* is true. If we are considering the global perspective, then it is safe to pass the value *IGRAPH_ALL* here. If *graph* is undirected, then we use all the immediate neighbours of *vid*. Thus if you know that *graph* is undirected, then it is safe to pass the value *IGRAPH_ALL* here. Supported values are:

`IGRAPH_OUT`

Use the out-neighbours of *vid*. This option is only relevant when *graph* is a digraph and we are considering the local perspective.

IGRAPH_IN

Use the in-neighbours of *vid*. Again this option is only relevant when *graph* is a directed graph and we are considering the local perspective.

IGRAPH_ALL

Use both the in- and out-neighbours of *vid*. This option is only relevant if *graph* is a digraph. Also use this value if *graph* is undirected or we are considering the global perspective.

Returns:

The error code *IGRAPH_EINVAL* is returned in each of the following cases: (1) Any of the parameters *graph*, *quantities*, or *strategies* is a null pointer. (2) The vector *quantities* or *strategies* has a length different from the number of vertices in *graph*. (3) The parameter *graph* is the empty or null graph, i.e. the graph with zero vertices and edges. (4) The vector *quantities* sums to zero.

Time complexity: O(n) where n is the number of vertices in the perspective to consider. If we consider the global perspective, then n is the number of vertices in the vertex set of *graph*. On the other hand, for the local perspective n is the degree of *vid*, excluding loops.

Reference:

(Yu & Gen 2010)

X. Yu and M. Gen. *Introduction to Evolutionary Algorithms.* Springer, 2010, pages 18--20.

Example 10-3. File examples/simple/igraph_roulette_wheel_imitation.c

10.1.4. `igraph_stochastic_imitation` — Adopt a strategy via stochastic imitation with uniform selection.

```
int igraph_stochastic_imitation(const igraph_t *graph,
                                igraph_integer_t vid,
                                igraph_imitate_algorithm_t algo,
                                const igraph_vector_t *quantities,
```

```
igraph_vector_t *strategies,
igraph_neimode_t mode);
```

A simple stochastic imitation strategy where a vertex revises its strategy to that of a vertex chosen uniformly at random from its local neighbourhood. This is called stochastic imitation via uniform selection, where the strategy to imitate is chosen via some random process. For the purposes of this function, we use uniform selection from a pool of candidates.

Arguments:

`graph`:

The graph object representing the game network. This cannot be the empty or trivial graph, but must have at least two vertices and one edge. If `graph` has one vertex, then no strategy update would take place. Furthermore, if `graph` has at least two vertices but zero edges, then strategy update would also not take place.

`vid`:

The vertex whose strategy is to be updated. It is assumed that `vid` represents a vertex in `graph`. No checking is performed and it is your responsibility to ensure that `vid` is indeed a vertex of `graph`. If an isolated vertex is provided, i.e. the input vertex has degree 0, then no strategy update would take place and `vid` would retain its current strategy. Strategy update would also not take place if the local neighbourhood of `vid` are its in-neighbours (respectively out-neighbours), but `vid` has zero in-neighbours (respectively out-neighbours). Loops are ignored in computing the degree (in, out, all) of `vid`.

`algo`:

This flag controls which algorithm to use in stochastic imitation. Supported values are:

`IGRAPH_IMITATE_AUGMENTED`

Augmented imitation. Vertex `vid` imitates the strategy of the chosen vertex u provided that doing so would increase the quantity (e.g. fitness) of `vid`. Augmented imitation can be thought of as "imitate if better".

`IGRAPH_IMITATE_BLIND`

Blind imitation. Vertex `vid` blindly imitates the strategy of the chosen vertex u, regardless of whether doing so would increase or decrease the quantity of `vid`.

`IGRAPH_IMITATE_CONTRACTED`

Contracted imitation. Here vertex `vid` imitates the strategy of the chosen vertex u if doing so would decrease the quantity of `vid`. Think of contracted imitation as "imitate if worse".

`quantities`:

A vector of quantities providing the quantity of each vertex in `graph`. Think of each entry of the vector as being generated by a function such as the fitness function for the game. So if the vector represents fitness quantities, then each vector entry is the fitness of some vertex. The length of this vector must be the same as the number of vertices in the vertex set of *graph*.

strategies:

A vector of the current strategies for the vertex population. The updated strategy for *vid* would be stored here. Each strategy is identified with a nonnegative integer, whose interpretation depends on the payoff matrix of the game. Generally we use the strategy ID as a row or column index of the payoff matrix. The length of this vector must be the same as the number of vertices in the vertex set of *graph*.

mode:

Defines the sort of neighbourhood to consider for *vid*. If *graph* is undirected, then we use all the immediate neighbours of *vid*. Thus if you know that *graph* is undirected, then it is safe to pass the value *IGRAPH_ALL* here. Supported values are:

IGRAPH_OUT

Use the out-neighbours of *vid*. This option is only relevant when *graph* is a directed graph.

IGRAPH_IN

Use the in-neighbours of *vid*. Again this option is only relevant when *graph* is a directed graph.

IGRAPH_ALL

Use both the in- and out-neighbours of *vid*. This option is only relevant if *graph* is a digraph. Also use this value if *graph* is undirected.

Returns:

The error code *IGRAPH_EINVAL* is returned in each of the following cases: (1) Any of the parameters *graph*, *quantities*, or *strategies* is a null pointer. (2) The vector *quantities* or *strategies* has a length different from the number of vertices in *graph*. (3) The parameter *graph* is the empty or null graph, i.e. the graph with zero vertices and edges. (4) The parameter *algo* refers to an unsupported stochastic imitation algorithm.

Time complexity: depends on the uniform random number generator, but should usually be O(1).

Example 10-4. File examples/simple/igraph_stochastic_imitation.c

Chapter 11. Vertex and Edge Selectors and Sequences, Iterators

11.1. About selectors, iterators

Everything about vertices and vertex selectors also applies to edges and edge selectors unless explicitly noted otherwise.

The vertex (and edge) selector notion was introduced in igraph 0.2. It is a way to reference a sequence of vertices or edges independently of the graph.

While this might sound quite mysterious, it is actually very simple. For example, all vertices of a graph can be selected by `igraph_vs_all()` and the graph independence means that `igraph_vs_all()` is not parametrized by a graph object. That is, `igraph_vs_all()` is the general *concept* of selecting all vertices of a graph. A vertex selector is then a way to specify the class of vertices to be visited. The selector might specify that all vertices of a graph or all the neighbours of a vertex are to be visited. A vertex selector is a way of saying that you want to visit a bunch of vertices, as opposed to a vertex iterator which is a concrete plan for visiting each of the chosen vertices of a specific graph.

To determine the actual vertex IDs implied by a vertex selector, you need to apply the concept of selecting vertices to a specific graph object. This can be accomplished by instantiating a vertex iterator using a specific vertex selection concept and a specific graph object. The notion of vertex iterators can be thought of in the following way. Given a specific graph object and the class of vertices to be visited, a vertex iterator is a road map, plan or route for how to visit the chosen vertices.

Some vertex selectors have *immediate* versions. These have the prefix `igraph_vss` instead of `igraph_vs`, e.g. `igraph_vss_all()` instead of `igraph_vs_all()`. The immediate versions are to be used in the parameter list of the igraph functions, such as `igraph_degree()`. These functions are not associated with any igraph_vs_t object, so they have no separate constructors and destructors (destroy functions).

11.2. Vertex selector constructors

Vertex selectors are created by vertex selector constructors, can be instantiated with `igraph_vit_create()`, and are destroyed with `igraph_vs_destroy()`.

11.2.1. `igraph_vs_all` — Vertex set, all vertices of a graph.

```
int igraph_vs_all(igraph_vs_t *vs);
```

Arguments:

`vs`:

Pointer to an uninitialized igraph_vs_t object.

Returns:

Error code.

See also:

`igraph_vss_all()`, `igraph_vs_destroy()`

This selector includes all vertices of a given graph in increasing vertex id order.

Time complexity: O(1).

11.2.2. `igraph_vs_adj` — Adjacent vertices of a vertex.

```
int igraph_vs_adj(igraph_vs_t *vs,
    igraph_integer_t vid, igraph_neimode_t mode);
```

All neighboring vertices of a given vertex are selected by this selector. The `mode` argument controls the type of the neighboring vertices to be selected. The vertices are visited in increasing vertex ID order, as of igraph version 0.4.

Arguments:

vs:

Pointer to an uninitialized vertex selector object.

vid:

Vertex ID, the center of the neighborhood.

mode:

Decides the type of the neighborhood for directed graphs. This parameter is ignored for undirected graphs. Possible values:

`IGRAPH_OUT`

All vertices to which there is a directed edge from `vid`. That is, all the out-neighbors of `vid`.

`IGRAPH_IN`

All vertices from which there is a directed edge to `vid`. In other words, all the in-neighbors of `vid`.

`IGRAPH_ALL`

All vertices to which or from which there is a directed edge from/to `vid`. That is, all the neighbors of `vid` considered as if the graph is undirected.

Returns:

Error code.

See also:

`igraph_vs_destroy()`

Time complexity: O(1).

11.2.3. `igraph_vs_nonadj` — Non-adjacent vertices of a vertex.

```
int igraph_vs_nonadj(igraph_vs_t *vs, igraph_integer_t vid,
        igraph_neimode_t mode);
```

All non-neighboring vertices of a given vertex. The `mode` argument controls the type of neighboring vertices *not* to select. Instead of selecting immediate neighbors of `vid` as is done by `igraph_vs_adj()`, the current function selects vertices that are *not* immediate neighbors of `vid`.

Arguments:

`vs`:

Pointer to an uninitialized vertex selector object.

`vid`:

Vertex ID, the “center” of the non-neighborhood.

`mode`:

The type of neighborhood not to select in directed graphs. Possible values:

`IGRAPH_OUT`

All vertices will be selected except those to which there is a directed edge from `vid`. That is, we select all vertices excluding the out-neighbors of `vid`.

`IGRAPH_IN`

All vertices will be selected except those from which there is a directed edge to `vid`. In other words, we select all vertices but the in-neighbors of `vid`.

`IGRAPH_ALL`

All vertices will be selected except those from or to which there is a directed edge to or from `vid`. That is, we select all vertices of `vid` except for its immediate neighbors.

Returns:

Error code.

See also:

`igraph_vs_destroy()`

Time complexity: O(1).

Example 11-1. File examples/simple/igraph_vs_nonadj.c

11.2.4. `igraph_vs_none` — Empty vertex set.

```
int igraph_vs_none(igraph_vs_t *vs);
```

Creates an empty vertex selector.

Arguments:

vs:

Pointer to an uninitialized vertex selector object.

Returns:

Error code.

See also:

`igraph_vss_none()`, `igraph_vs_destroy()`

Time complexity: O(1).

11.2.5. `igraph_vs_1` — Vertex set with a single vertex.

```
int igraph_vs_1(igraph_vs_t *vs, igraph_integer_t vid);
```

This vertex selector selects a single vertex.

Arguments:

vs:

Pointer to an uninitialized vertex selector object.

vid:

The vertex id to be selected.

Returns:

Error Code.

See also:

`igraph_vss_1()`, `igraph_vs_destroy()`

Time complexity: O(1).

11.2.6. `igraph_vs_vector` — Vertex set based on a vector.

```
int igraph_vs_vector(igraph_vs_t *vs,
       const igraph_vector_t *v);
```

This function makes it possible to handle a vector_t temporarily as a vertex selector. The vertex selector should be thought of like a *view* to the vector. If you make changes to the vector that also affects the vertex selector. Destroying the vertex selector does not destroy the vector. (Of course.) Do not destroy the vector before destroying the vertex selector, or you might get strange behavior.

Arguments:

vs:

Pointer to an uninitialized vertex selector.

v:

Pointer to a igraph_vector_t object.

Returns:

Error code.

See also:

`igraph_vss_vector()`, `igraph_vs_destroy()`

Time complexity: O(1).

Example 11-2. File examples/simple/igraph_vs_vector.c

11.2.7. `igraph_vs_vector_small` — Create a vertex set by giving its elements.

```
int igraph_vs_vector_small(igraph_vs_t *vs, ...);
```

This function can be used to create a vertex selector with a couple of vertices. Do not forget to include a `-1` after the last vertex id. The behavior of the function is undefined if you don't use a `-1` properly.

Note that the vertex ids supplied will be parsed as `int`'s so you cannot supply arbitrarily large (too large for int) vertex ids here.

Arguments:

`vs`:

Pointer to an uninitialized vertex selector object.

`...`:

Additional parameters, these will be the vertex ids to be included in the vertex selector. Supply a `-1` after the last vertex id.

Returns:

Error code.

See also:

`igraph_vs_destroy()`

Time complexity: O(n), the number of vertex ids supplied.

11.2.8. `igraph_vs_vector_copy` — Vertex set based on a vector, with copying.

```
int igraph_vs_vector_copy(igraph_vs_t *vs,
     const igraph_vector_t *v);
```

This function makes it possible to handle a vector_t permanently as a vertex selector. The vertex selector creates a copy of the original vector, so the vector can safely be destroyed after creating the vertex selector. Changing the original vector will not affect the vertex selector. The vertex selector is responsible for deleting the copy made by itself.

Arguments:

`vs`:

Pointer to an uninitialized vertex selector.

`v`:

Pointer to a igraph_vector_t object.

Returns:

Error code.

See also:

`igraph_vs_destroy()`

Time complexity: O(1).

11.2.9. `igraph_vs_seq` — Vertex set, an interval of vertices.

```
int igraph_vs_seq(igraph_vs_t *vs,
    igraph_integer_t from, igraph_integer_t to);
```

Creates a vertex selector containing all vertices with vertex id equal to or bigger than `from` and equal to or smaller than `to`.

Arguments:

`vs`:

Pointer to an uninitialized vertex selector object.

`from`:

The first vertex id to be included in the vertex selector.

`to`:

The last vertex id to be included in the vertex selector.

Returns:

Error code.

See also:

`igraph_vss_seq()`, `igraph_vs_destroy()`

Time complexity: O(1).

Example 11-3. File examples/simple/igraph_vs_seq.c

11.3. Generic vertex selector operations

11.3.1. `igraph_vs_copy` — Creates a copy of a vertex selector.

```
int igraph_vs_copy(igraph_vs_t* dest, const igraph_vs_t* src);
```

Arguments:

src:

The selector being copied.

dest:

An uninitialized selector that will contain the copy.

11.3.2. `igraph_vs_destroy` — Destroy a vertex set.

```
void igraph_vs_destroy(igraph_vs_t *vs);
```

This function should be called for all vertex selectors when they are not needed. The memory allocated for the vertex selector will be deallocated. Do not call this function on vertex selectors created with the immediate versions of the vertex selector constructors (starting with `igraph_vss`).

Arguments:

vs:

Pointer to a vertex selector object.

Time complexity: operating system dependent, usually O(1).

11.3.3. `igraph_vs_is_all` — Check whether all vertices are included.

```
igraph_bool_t igraph_vs_is_all(const igraph_vs_t *vs);
```

This function checks whether the vertex selector object was created by `igraph_vs_all()` or `igraph_vss_all()`. Note that the vertex selector might contain all vertices in a given graph but if it wasn't created by the two constructors mentioned here the return value will be FALSE.

Arguments:

vs:

Pointer to a vertex selector object.

Returns:

TRUE (1) if the vertex selector contains all vertices and FALSE (0) otherwise.

Time complexity: O(1).

11.3.4. `igraph_vs_size` — Returns the size of the vertex selector.

```
int igraph_vs_size(const igraph_t *graph, const igraph_vs_t *vs,
  igraph_integer_t *result);
```

The size of the vertex selector is the number of vertices it will yield when it is iterated over.

Arguments:

graph:

The graph over which we will iterate.

result:

The result will be returned here.

11.3.5. `igraph_vs_type` — Returns the type of the vertex selector.

```
int igraph_vs_type(const igraph_vs_t *vs);
```

11.4. Immediate vertex selectors

11.4.1. `igraph_vss_all` — All vertices of a graph (immediate version).

```
igraph_vs_t igraph_vss_all(void);
```

Immediate vertex selector for all vertices in a graph. It can be used conveniently when some vertex property (eg. betweenness, degree, etc.) should be calculated for all vertices.

Returns:

A vertex selector for all vertices in a graph.

See also:

`igraph_vs_all()`

Time complexity: O(1).

11.4.2. `igraph_vss_none` — Empty vertex set (immediate version).

```
igraph_vs_t igraph_vss_none(void);
```

The immediate version of the empty vertex selector.

Returns:

An empty vertex selector.

See also:

`igraph_vs_none()`

Time complexity: O(1).

11.4.3. `igraph_vss_1` — Vertex set with a single vertex (immediate version).

```
igraph_vs_t igraph_vss_1(igraph_integer_t vid);
```

The immediate version of the single-vertex selector.

Arguments:

`vid`:

The vertex to be selected.

Returns:

A vertex selector containing a single vertex.

See also:

`igraph_vs_1()`

Time complexity: O(1).

11.4.4. `igraph_vss_vector` — Vertex set based on a vector (immediate version).

```
igraph_vs_t igraph_vss_vector(const igraph_vector_t *v);
```

This is the immediate version of `igraph_vs_vector`.

Arguments:

v:

Pointer to a igraph_vector_t object.

Returns:

A vertex selector object containing the vertices in the vector.

See also:

`igraph_vs_vector()`

Time complexity: O(1).

11.4.5. `igraph_vss_seq` — An interval of vertices (immediate version).

```
igraph_vs_t igraph_vss_seq(igraph_integer_t from, igraph_integer_t to);
```

The immediate version of `igraph_vs_seq()`.

Arguments:

from:

The first vertex id to be included in the vertex selector.

to:

The last vertex id to be included in the vertex selector.

Returns:

Error code.

See also:

`igraph_vs_seq()`

Time complexity: O(1).

11.5. Vertex iterators

11.5.1. `igraph_vit_create` — Creates a vertex iterator from a vertex selector.

```
int igraph_vit_create(const igraph_t *graph,
        igraph_vs_t vs, igraph_vit_t *vit);
```

This function instantiates a vertex selector object with a given graph. This is the step when the actual vertex ids are created from the *logical* notion of the vertex selector based on the graph. Eg. a vertex selector created with `igraph_vs_all()` contains knowledge that *all* vertices are included in a (yet indefinite) graph. When instantiating it a vertex iterator object is created, this contains the actual vertex ids in the graph supplied as a parameter.

The same vertex selector object can be used to instantiate any number vertex iterators.

Arguments:

graph:

An igraph_t object, a graph.

vs:

A vertex selector object.

vit:

Pointer to an uninitialized vertex iterator object.

Returns:

Error code.

See also:

`igraph_vit_destroy()`.

Time complexity: it depends on the vertex selector type. O(1) for vertex selectors created with `igraph_vs_all()`, `igraph_vs_none()`, `igraph_vs_1`, `igraph_vs_vector`, `igraph_vs_seq()`, `igraph_vs_vector()`, `igraph_vs_vector_small()`. O(d) for `igraph_vs_adj()`, d is the number of vertex ids to be included in the iterator. O(|V|) for `igraph_vs_nonadj()`, |V| is the number of vertices in the graph.

11.5.2. `igraph_vit_destroy` — Destroys a vertex iterator.

```
void igraph_vit_destroy(const igraph_vit_t *vit);
```

Deallocates memory allocated for a vertex iterator.

Arguments:

vit:

Pointer to an initialized vertex iterator object.

See also:

```
igraph_vit_create()
```

Time complexity: operating system dependent, usually O(1).

11.5.3. Stepping over the vertices

After creating an iterator with `igraph_vit_create()`, it points to the first vertex in the vertex determined by the vertex selector (if there is any). The `IGRAPH_VIT_NEXT()` macro steps to the next vertex, `IGRAPH_VIT_END()` checks whether there are more vertices to visit, `IGRAPH_VIT_SIZE()` gives the total size of the vertices visited so far and to be visited. `IGRAPH_VIT_RESET()` resets the iterator, it will point to the first vertex again. Finally `IGRAPH_VIT_GET()` gives the current vertex pointed to by the iterator (call this only if `IGRAPH_VIT_END()` is false).

Here is an example on how to step over the neighbors of vertex 0:

```
igraph_vs_t vs;
igraph_vit_t vit;
...
igraph_vs_adj(&vs, 0, IGRAPH_ALL);
igraph_vit_create(&graph, vs, &vit);
while (!IGRAPH_VIT_END(vit)) {
  printf(" %li", (long int) IGRAPH_VIT_GET(vit));
  IGRAPH_VIT_NEXT(vit);
}
printf("\n");
...
igraph_vit_destroy(&vit);
igraph_vs_destroy(&vs);
```

11.5.4. `IGRAPH_VIT_NEXT` — Next vertex.

```
#define IGRAPH_VIT_NEXT(vit)
```

Steps the iterator to the next vertex. Only call this function if `IGRAPH_VIT_END()` returns false.

Arguments:

vit:

The vertex iterator to step.

Time complexity: O(1).

11.5.5. `IGRAPH_VIT_END` — Are we at the end?

```
#define IGRAPH_VIT_END(vit)
```

Checks whether there are more vertices to step to.

Arguments:

vit:

The vertex iterator to check.

Returns:

Logical value, if true there are no more vertices to step to.

Time complexity: O(1).

11.5.6. `IGRAPH_VIT_SIZE` — Size of a vertex iterator.

```
#define IGRAPH_VIT_SIZE(vit)
```

Gives the number of vertices in a vertex iterator.

Arguments:

vit:

The vertex iterator.

Returns:

The number of vertices.

Time complexity: O(1).

11.5.7. IGRAPH_VIT_RESET — Reset a vertex iterator.

```
#define IGRAPH_VIT_RESET(vit)
```

Resets a vertex iterator. After calling this macro the iterator will point to the first vertex.

Arguments:

vit:

The vertex iterator.

Time complexity: O(1).

11.5.8. IGRAPH_VIT_GET — Query the current position.

```
#define IGRAPH_VIT_GET(vit)
```

Gives the vertex id of the current vertex pointed to by the iterator.

Arguments:

vit:

The vertex iterator.

Returns:

The vertex id of the current vertex.

Time complexity: O(1).

11.6. Edge selector constructors

11.6.1. `igraph_es_all` — Edge set, all edges.

```
int igraph_es_all(igraph_es_t *es,
    igraph_edgeorder_type_t order);
```

Arguments:

`es`:

Pointer to an uninitialized edge selector object.

`order`:

Constant giving the order in which the edges will be included in the selector. Possible values: `IGRAPH_EDGEORDER_ID`, edge id order. `IGRAPH_EDGEORDER_FROM`, vertex id order, the id of the *source* vertex counts for directed graphs. The order of the incident edges of a given vertex is arbitrary. `IGRAPH_EDGEORDER_TO`, vertex id order, the id of the *target* vertex counts for directed graphs. The order of the incident edges of a given vertex is arbitrary. For undirected graph the latter two is the same.

Returns:

Error code.

See also:

`igraph_ess_all()`, `igraph_es_destroy()`

Time complexity: O(1).

11.6.2. `igraph_es_incident` — Edges incident on a given vertex.

```
int igraph_es_incident(igraph_es_t *es,
    igraph_integer_t vid, igraph_neimode_t mode);
```

Arguments:

`es`:

Pointer to an uninitialized edge selector object.

`vid`:

Vertex id, of which the incident edges will be selected.

`mode`:

Constant giving the type of the incident edges to select. This is ignored for undirected graphs. Possible values: `IGRAPH_OUT`, outgoing edges; `IGRAPH_IN`, incoming edges; `IGRAPH_ALL`, all edges.

Returns:

Error code.

See also:

`igraph_es_destroy()`

Time complexity: O(1).

Example 11-4. File examples/simple/igraph_es_adj.c

11.6.3. `igraph_es_none` — Empty edge selector.

```
int igraph_es_none(igraph_es_t *es);
```

Arguments:

es:

Pointer to an uninitialized edge selector object to initialize.

Returns:

Error code.

See also:

`igraph_ess_none()`, `igraph_es_destroy()`

Time complexity: O(1).

11.6.4. `igraph_es_1` — Edge selector containing a single edge.

```
int igraph_es_1(igraph_es_t *es, igraph_integer_t eid);
```

Arguments:

es:

Pointer to an uninitialized edge selector object.

eid:

Edge id of the edge to select.

Returns:

Error code.

See also:

`igraph_ess_1()`, `igraph_es_destroy()`

Time complexity: O(1).

11.6.5. `igraph_es_vector` — Handle a vector as an edge selector.

```
int igraph_es_vector(igraph_es_t *es,
       const igraph_vector_t *v);
```

Creates an edge selector which serves as a view to a vector containing edge ids. Do not destroy the vector before destroying the view. Many views can be created to the same vector.

Arguments:

es:

Pointer to an uninitialized edge selector.

v:

Vector containing edge ids.

Returns:

Error code.

See also:

```
igraph_ess_vector(), igraph_es_destroy()
```

Time complexity: O(1).

11.6.6. `igraph_es_fromto` — Edge selector, all edges between two vertex sets.

```
int igraph_es_fromto(igraph_es_t *es,
       igraph_vs_t from, igraph_vs_t to);
```

This function is not implemented yet.

Arguments:

`es`:

Pointer to an uninitialized edge selector.

`from`:

Vertex selector, their outgoing edges will be selected.

`to`:

Vertex selector, their incoming edges will be selected from the previous selection.

Returns:

Error code.

See also:

```
igraph_es_destroy()
```

Time complexity: O(1).

Example 11-5. File examples/simple/igraph_es_fromto.c

11.6.7. `igraph_es_seq` — Edge selector, a sequence of edge ids.

```
int igraph_es_seq(igraph_es_t *es,
    igraph_integer_t from, igraph_integer_t to);
```

All edge ids between `from` and `to` will be included in the edge selection.

Arguments:

`es`:

Pointer to an uninitialized edge selector object.

`from`:

The first edge id to be included.

`to`:

The last edge id to be included.

Returns:

Error code.

See also:

`igraph_ess_seq()`, `igraph_es_destroy()`

Time complexity: O(1).

11.6.8. `igraph_es_pairs` — Edge selector, multiple edges defined by their endpoints in a vector.

```
int igraph_es_pairs(igraph_es_t *es, const igraph_vector_t *v,
      igraph_bool_t directed);
```

The edges between the given pairs of vertices will be included in the edge selection. The vertex pairs must be defined in the vector `v`, the first element of the vector is the first vertex of the first edge to be selected, the second element is the second vertex of the first edge, the third element is the first vertex of the second edge and so on.

Arguments:

`es`:

Pointer to an uninitialized edge selector object.

`v`:

The vector containing the endpoints of the edges.

`directed`:

Whether the graph is directed or not.

Returns:

Error code.

See also:

`igraph_es_pairs_small()`, `igraph_es_destroy()`

Time complexity: O(n), the number of edges being selected.

Example 11-6. File examples/simple/igraph_es_pairs.c

11.6.9. `igraph_es_pairs_small` — Edge selector, multiple edges defined by their endpoints as arguments.

```
int igraph_es_pairs_small(igraph_es_t *es, igraph_bool_t directed, ...);
```

The edges between the given pairs of vertices will be included in the edge selection. The vertex pairs must be given as the arguments of the function call, the third argument is the first vertex of the first edge, the fourth argument is the second vertex of the first edge, the fifth is the first vertex of the second edge and so on. The last element of the argument list must be -1 to denote the end of the argument list.

Arguments:

`es`:

Pointer to an uninitialized edge selector object.

`directed`:

Whether the graph is directed or not.

Returns:

Error code.

See also:

`igraph_es_pairs()`, `igraph_es_destroy()`

Time complexity: O(n), the number of edges being selected.

11.6.10. `igraph_es_vector_copy` — Edge set, based on a vector, with copying.

```
int igraph_es_vector_copy(igraph_es_t *es, const igraph_vector_t *v);
```

This function makes it possible to handle a vector_t permanently as an edge selector. The edge selector creates a copy of the original vector, so the vector can safely be destroyed after creating the edge selector. Changing the original vector will not affect the edge selector. The edge selector is responsible for deleting the copy made by itself.

Arguments:

`es`:

Pointer to an uninitialized edge selector.

`v`:

Pointer to a igraph_vector_t object.

Returns:

Error code.

See also:

`igraph_es_destroy()`

Time complexity: O(1).

11.7. Immediate edge selectors

11.7.1. `igraph_ess_all` — Edge set, all edges (immediate version)

```
igraph_es_t igraph_ess_all(igraph_edgeorder_type_t order);
```

The immediate version of the all-vertices selector.

Arguments:

order:

Constant giving the order of the edges in the edge selector. See `igraph_es_all()` for the possible values.

Returns:

The edge selector.

See also:

`igraph_es_all()`

Time complexity: O(1).

11.7.2. `igraph_ess_none` — Immediate empty edge selector.

```
igraph_es_t igraph_ess_none(void);
```

Immediate version of the empty edge selector.

Returns:

Initialized empty edge selector.

See also:

`igraph_es_none()`

Time complexity: O(1).

11.7.3. `igraph_ess_1` — Immediate version of the single edge edge selector.

```
igraph_es_t igraph_ess_1(igraph_integer_t eid);
```

Arguments:

eid:

The id of the edge.

Returns:

The edge selector.

See also:

`igraph_es_1()`

Time complexity: O(1).

11.7.4. `igraph_ess_vector` — Immediate vector view edge selector.

```
igraph_es_t igraph_ess_vector(const igraph_vector_t *v);
```

This is the immediate version of the vector of edge ids edge selector.

Arguments:

v:

The vector of edge ids.

Returns:

Edge selector, initialized.

See also:

`igraph_es_vector()`

Time complexity: O(1).

11.7.5. `igraph_ess_seq` — Immediate version of the sequence edge selector.

```
igraph_es_t igraph_ess_seq(igraph_integer_t from, igraph_integer_t to);
```

Arguments:

`from`:

The first edge id to include.

`to`:

The last edge id to include.

Returns:

The initialized edge selector.

See also:

`igraph_es_seq()`

Time complexity: O(1).

11.8. Generic edge selector operations

11.8.1. `igraph_es_copy` — Creates a copy of an edge selector.

```
int igraph_es_copy(igraph_es_t* dest, const igraph_es_t* src);
```

Arguments:

`src`:

The selector being copied.

`dest`:

An uninitialized selector that will contain the copy.

See also:

`igraph_es_destroy()`

11.8.2. `igraph_es_destroy` — Destroys an edge selector object.

```
void igraph_es_destroy(igraph_es_t *es);
```

Call this function on an edge selector when it is not needed any more. Do *not* call this function on edge selectors created by immediate constructors, those don't need to be destroyed.

Arguments:

`es`:

Pointer to an edge selector object.

Time complexity: operating system dependent, usually O(1).

11.8.3. `igraph_es_is_all` — Check whether an edge selector includes all edges.

```
igraph_bool_t igraph_es_is_all(const igraph_es_t *es);
```

Arguments:

es:

Pointer to an edge selector object.

Returns:

TRUE (1) if `es` was created with `igraph_es_all()` or `igraph_ess_all()`, and FALSE (0) otherwise.

Time complexity: O(1).

11.8.4. `igraph_es_size` — Returns the size of the edge selector.

```
int igraph_es_size(const igraph_t *graph, const igraph_es_t *es,
  igraph_integer_t *result);
```

The size of the edge selector is the number of edges it will yield when it is iterated over.

Arguments:

graph:

The graph over which we will iterate.

result:

The result will be returned here.

11.8.5. igraph_es_type — Returns the type of the edge selector.

```
int igraph_es_type(const igraph_es_t *es);
```

11.9. Edge iterators

11.9.1. igraph_eit_create — Creates an edge iterator from an edge selector.

```
int igraph_eit_create(const igraph_t *graph,
        igraph_es_t es, igraph_eit_t *eit);
```

This function creates an edge iterator based on an edge selector and a graph.

The same edge selector can be used to create many edge iterators, also for different graphs.

Arguments:

graph:

An igraph_t object for which the edge selector will be instantiated.

es:

The edge selector to instantiate.

eit:

Pointer to an uninitialized edge iterator.

Returns:

Error code.

See also:

`igraph_eit_destroy()`

Time complexity: depends on the type of the edge selector. For edge selectors created by `igraph_es_all()`, `igraph_es_none()`, `igraph_es_1()`, igraph_es_vector(), igraph_es_seq() it is O(1). For `igraph_es_incident()` it is O(d) where d is the number of incident edges of the vertex.

11.9.2. `igraph_eit_destroy` — Destroys an edge iterator.

```
void igraph_eit_destroy(const igraph_eit_t *eit);
```

Arguments:

eit:

Pointer to an edge iterator to destroy.

See also:

`igraph_eit_create()`

Time complexity: operating system dependent, usually O(1).

11.9.3. Stepping over the edges

Just like for vertex iterators, macros are provided for stepping over a sequence of edges: `IGRAPH_EIT_NEXT()` goes to the next edge, `IGRAPH_EIT_END()` checks whether there are more edges to visit, `IGRAPH_EIT_SIZE()` gives the number of edges in the edge sequence, `IGRAPH_EIT_RESET()` resets the iterator to the first edge and `IGRAPH_EIT_GET()` returns the id of the current edge.

11.9.4. `IGRAPH_EIT_NEXT` — Next edge.

```
#define IGRAPH_EIT_NEXT(eit)
```

Steps the iterator to the next edge. Call this function only if `IGRAPH_EIT_END()` returns false.

Arguments:

`eit`:

The edge iterator to step.

Time complexity: O(1).

11.9.5. `IGRAPH_EIT_END` — Are we at the end?

```
#define IGRAPH_EIT_END(eit)
```

Checks whether there are more edges to step to.

Arguments:

`wit`:

The edge iterator to check.

Returns:

Logical value, if true there are no more edges to step to.

Time complexity: O(1).

11.9.6. IGRAPH_EIT_SIZE — Number of edges in the iterator.

```
#define IGRAPH_EIT_SIZE(eit)
```

Gives the number of edges in an edge iterator.

Arguments:

eit:

The edge iterator.

Returns:

The number of edges.

Time complexity: O(1).

11.9.7. IGRAPH_EIT_RESET — Reset an edge iterator.

```
#define IGRAPH_EIT_RESET(eit)
```

Resets an edge iterator. After calling this macro the iterator will point to the first edge.

Arguments:

eit:

The edge iterator.

Time complexity: O(1).

11.9.8. `IGRAPH_EIT_GET` — Query an edge iterator.

```
#define IGRAPH_EIT_GET(eit)
```

Gives the edge id of the current edge pointed to by an iterator.

Arguments:

`eit`:

The edge iterator.

Returns:

The id of the current edge.

Time complexity: O(1).

Chapter 12. Graph, Vertex and Edge Attributes

Attributes are numbers or strings (or basically any kind of data) associated with the vertices or edges of a graph, or with the graph itself. Eg. you may label vertices with symbolic names or attach numeric weights to the edges of a graph.

igraph attributes are designed to be flexible and extensible. In igraph attributes are implemented via an interface abstraction: any type implementing the functions in the interface, can be used for storing vertex, edge and graph attributes. This means that different attribute implementations can be used together with igraph. This is reasonable: if igraph is used from Python attributes can be of any Python type, from GNU R all R types are allowed. There is an experimental attribute implementation to be used when programming in C, but by default it is currently turned off.

First we briefly look over how attribute handlers can be implemented. This is not something a user does every day. It is rather typically the job of the high level interface writers. (But it is possible to write an interface without implementing attributes.) Then we show the experimental C attribute handler.

12.1. The Attribute Handler Interface

It is possible to attach an attribute handling interface to **igraph**. This is simply a table of functions, of type `igraph_attribute_table_t`. These functions are invoked to notify the attribute handling code about the structural changes in a graph. See the documentation of this type for details.

By default there is no attribute interface attached to **igraph**, to attach one, call `igraph_i_set_attribute_table` with your new table.

12.1.1. `igraph_attribute_table_t` — Table of functions to perform operations on attributes

```
typedef struct igraph_attribute_table_t {
  int (*init)(igraph_t *graph, igraph_vector_ptr_t *attr);
  void (*destroy)(igraph_t *graph);
  int (*copy)(igraph_t *to, const igraph_t *from, igraph_bool_t ga,
       igraph_bool_t va, igraph_bool_t ea);
  int (*add_vertices)(igraph_t *graph, long int nv, igraph_vector_ptr_t *attr);
  int (*permute_vertices)(const igraph_t *graph,
     igraph_t *newgraph,
     const igraph_vector_t *idx);
  int (*combine_vertices)(const igraph_t *graph,
     igraph_t *newgraph,
     const igraph_vector_ptr_t *merges,
```

```
     const igraph_attribute_combination_t *comb);
  int (*add_edges)(igraph_t *graph, const igraph_vector_t *edges,
     igraph_vector_ptr_t *attr);
  int (*permute_edges)(const igraph_t *graph,
         igraph_t *newgraph, const igraph_vector_t *idx);
  int (*combine_edges)(const igraph_t *graph,
         igraph_t *newgraph,
         const igraph_vector_ptr_t *merges,
         const igraph_attribute_combination_t *comb);
  int (*get_info)(const igraph_t *graph,
    igraph_strvector_t *gnames, igraph_vector_t *gtypes,
    igraph_strvector_t *vnames, igraph_vector_t *vtypes,
    igraph_strvector_t *enames, igraph_vector_t *etypes);
  igraph_bool_t (*has_attr)(const igraph_t *graph, igraph_attribute_elemtype_t type,
       const char *name);
  int (*gettype)(const igraph_t *graph, igraph_attribute_type_t *type,
   igraph_attribute_elemtype_t elemtype, const char *name);
  int (*get_numeric_graph_attr)(const igraph_t *graph, const char *name,
    igraph_vector_t *value);
  int (*get_string_graph_attr)(const igraph_t *graph, const char *name,
          igraph_strvector_t *value);
  int (*get_numeric_vertex_attr)(const igraph_t *graph, const char *name,
     igraph_vs_t vs,
     igraph_vector_t *value);
  int (*get_string_vertex_attr)(const igraph_t *graph, const char *name,
    igraph_vs_t vs,
    igraph_strvector_t *value);
  int (*get_numeric_edge_attr)(const igraph_t *graph, const char *name,
          igraph_es_t es,
          igraph_vector_t *value);
  int (*get_string_edge_attr)(const igraph_t *graph, const char *name,
         igraph_es_t es,
         igraph_strvector_t *value);
} igraph_attribute_table_t;
```

This type collects the functions defining an attribute handler. It has the following members:

Values:

`init`:

This function is called whenever a new graph object is created, right after it is created but before any vertices or edges are added. It is supposed to set the `attr` member of the `igraph_t` object. It is expected to return an error code.

`destroy`:

This function is called whenever the graph object is destroyed, right before freeing the allocated memory.

`copy`:

This function is called when copying a graph with `igraph_copy`, after the structure of the graph has been already copied. It is expected to return an error code.

`add_vertices`:

Called when vertices are added to a graph, before adding the vertices themselves. The number of vertices to add is supplied as an argument. Expected to return an error code.

`permute_vertices`:

Typically called when a new graph is created based on an existing one, e.g. if vertices are removed from a graph. The supplied index vector defines which old vertex a new vertex corresponds to. Its length must be the same as the number of vertices in the new graph.

`combine_vertices`:

This function is called when the creation of a new graph involves a merge (contraction, etc.) of vertices from another graph. The function is after the new graph was created. An argument specifies how several vertices from the old graph map to a single vertex in the new graph.

`add_edges`:

Called when new edges have been added. The number of new edges are supplied as well. It is expected to return an error code.

`permute_edges`:

Typically called when a new graph is created and some of the new edges should carry the attributes of some of the old edges. The idx vector shows the mapping between the old edges and the new ones. Its length is the same as the number of edges in the new graph, and for each edge it gives the id of the old edge (the edge in the old graph).

`combine_edges`:

This function is called when the creation of a new graph involves a merge (contraction, etc.) of edges from another graph. The function is after the new graph was created. An argument specifies how several edges from the old graph map to a single edge in the new graph.

`get_info`:

Query the attributes of a graph, the names and types should be returned.

`has_attr`:

Check whether a graph has the named graph/vertex/edge attribute.

`gettype`:

Query the type of a graph/vertex/edge attribute.

`get_numeric_graph_attr`:

Query a numeric graph attribute. The value should be placed as the first element of the *`value`* vector.

`get_string_graph_attr`:

Query a string graph attribute. The value should be placed as the first element of the *`value`* string vector.

`get_numeric_vertex_attr`:

Query a numeric vertex attribute, for the vertices included in *`vs`*.

`get_string_vertex_attr`:

Query a string vertex attribute, for the vertices included in *`vs`*.

`get_numeric_edge_attr`:

Query a numeric edge attribute, for the edges included in *`es`*.

`get_string_edge_attr`:

Query a string edge attribute, for the edge included in *`es`*.

Note that the `get_*_*_attr` are allowed to convert the attributes to numeric or string. E.g. if a vertex attribute is a GNU R complex data type, then `get_string_vertex_attribute` may serialize it into a string, but this probably makes sense only if `add_vertices` is able to deserialize it.

12.1.2. `igraph_i_set_attribute_table` — Attach an attribute table.

```
igraph_attribute_table_t *
igraph_i_set_attribute_table(const igraph_attribute_table_t * table);
```

This function attaches attribute handling code to the igraph library. Note that the attribute handler table is *not* thread-local even if igraph is compiled in thread-local mode. In the vast majority of cases, this is not a significant restriction.

Arguments:

`table`:

Pointer to an `igraph_attribute_table_t` object containing the functions for attribute manipulation. Supply `NULL` here if you don't want attributes.

Returns:

Pointer to the old attribute handling table.

Time complexity: O(1).

12.1.3. `igraph_attribute_type_t` — The possible types of the attributes.

```
typedef enum { IGRAPH_ATTRIBUTE_DEFAULT=0,
        IGRAPH_ATTRIBUTE_NUMERIC=1,
        IGRAPH_ATTRIBUTE_STRING=2,
        IGRAPH_ATTRIBUTE_R_OBJECT=3,
        IGRAPH_ATTRIBUTE_PY_OBJECT=4 } igraph_attribute_type_t;
```

Note that this is only the type communicated by the attribute interface towards igraph functions. Eg. in the GNU R attribute handler, it is safe to say that all complex R object attributes are strings, as long as this interface is able to serialize them into strings. See also `igraph_attribute_table_t`.

Values:

`IGRAPH_ATTRIBUTE_DEFAULT`:

Currently not used for anything.

`IGRAPH_ATTRIBUTE_NUMERIC`:

Numeric attribute.

`IGRAPH_ATTRIBUTE_STRING`:

Attribute that can be converted to a string.

`IGRAPH_ATTRIBUTE_R_OBJECT`:

An R object. This is usually ignored by the igraph functions.

`IGRAPH_ATTRIBUTE_PY_OBJECT`:

A Python object. Usually ignored by the igraph functions.

12.2. Accessing attributes from C

There is an experimental attribute handler that can be used from C code. In this section we show how this works. This attribute handler is by default not attached (the default is no attribute handler), so we first need to attach it:

```
igraph_i_set_attribute_table(&igraph_cattribute_table);
```

Now the attribute functions are available. Please note that the attribute handler must be attached before you call any other igraph functions, otherwise you might end up with graphs without attributes and an active attribute handler, which might cause unexpected program behaviour. The rule is that you attach the attribute handler in the beginning of your `main()` and never touch it again. (Detaching the attribute handler might lead to memory leaks.)

It is not currently possible to have attribute handlers on a per-graph basis. All graphs in an application must be managed with the same attribute handler. (Including the default case when there is no attribute handler at all.

The C attribute handler supports attaching real numbers and character strings as attributes. No vectors are allowed, ie. every vertex might have an attribute called `name`, but it is not possible to have a `coords` graph (or other) attribute which is a vector of numbers.

Example 12-1. File examples/simple/cattributes.c

Example 12-2. File examples/simple/cattributes2.c

Example 12-3. File examples/simple/cattributes3.c

Example 12-4. File examples/simple/cattributes4.c

12.2.1. Query attributes

12.2.1.1. `igraph_cattribute_list` — List all attributes

```
int igraph_cattribute_list(const igraph_t *graph,
      igraph_strvector_t *gnames, igraph_vector_t *gtypes,
      igraph_strvector_t *vnames, igraph_vector_t *vtypes,
      igraph_strvector_t *enames, igraph_vector_t *etypes);
```

See `igraph_attribute_type_t` for the various attribute types.

Arguments:

`graph`:

The input graph.

`gnames`:

String vector, the names of the graph attributes.

`gtypes`:

Numeric vector, the types of the graph attributes.

`vnames`:

String vector, the names of the vertex attributes.

`vtypes`:

Numeric vector, the types of the vertex attributes.

`enames`:

String vector, the names of the edge attributes.

`etypes`:

Numeric vector, the types of the edge attributes.

Returns:

Error code.

Naturally, the string vector with the attribute names and the numeric vector with the attribute types are in the right order, i.e. the first name corresponds to the first type, etc. Time complexity: O(Ag+Av+Ae), the number of all attributes.

12.2.1.2. `igraph_cattribute_has_attr` — Checks whether a (graph, vertex or edge) attribute exists

```
igraph_bool_t igraph_cattribute_has_attr(const igraph_t *graph,
      igraph_attribute_elemtype_t type,
      const char *name);
```

Arguments:

graph:

The graph.

type:

The type of the attribute, `IGRAPH_ATTRIBUTE_GRAPH`, `IGRAPH_ATTRIBUTE_VERTEX` or `IGRAPH_ATTRIBUTE_EDGE`.

name:

Character constant, the name of the attribute.

Returns:

Logical value, TRUE if the attribute exists, FALSE otherwise.

Time complexity: O(A), the number of (graph, vertex or edge) attributes, assuming attribute names are not too long.

12.2.1.3. `igraph_cattribute_GAN` — Query a numeric graph attribute.

```
igraph_real_t igraph_cattribute_GAN(const igraph_t *graph, const char *name);
```

Returns the value of the given numeric graph attribute. The attribute must exist, otherwise an error is triggered.

Arguments:

graph:

The input graph.

name:

The name of the attribute to query.

Returns:

The value of the attribute.

See also:

`GAN` for a simpler interface.

Time complexity: O(Ag), the number of graph attributes.

12.2.1.4. GAN — Query a numeric graph attribute.

```
#define GAN(graph,n)
```

This is shorthand for `igraph_cattribute_GAN()`.

Arguments:

graph:

The graph.

n:

The name of the attribute.

Returns:

The value of the attribute.

12.2.1.5. `igraph_cattribute_GAS` — Query a string graph attribute.

```
const char* igraph_cattribute_GAS(const igraph_t *graph, const char *name);
```

Returns a const pointer to the string graph attribute specified in *name*. The attribute must exist, otherwise an error is triggered.

Arguments:

graph:

The input graph.

name:

The name of the attribute to query.

Returns:

The value of the attribute.

See also:

`GAS` for a simpler interface.

Time complexity: O(Ag), the number of graph attributes.

12.2.1.6. `GAS` — Query a string graph attribute.

```
#define GAS(graph,n)
```

This is shorthand for `igraph_cattribute_GAS()`.

Arguments:

graph:

The graph.

n:

The name of the attribute.

Returns:

The value of the attribute.

12.2.1.7. `igraph_cattribute_VAN` — Query a numeric vertex attribute.

```
igraph_real_t igraph_cattribute_VAN(const igraph_t *graph, const char *name,
          igraph_integer_t vid);
```

The attribute must exist, otherwise an error is triggered.

Arguments:

graph:

The input graph.

name:

The name of the attribute.

vid:

The id of the queried vertex.

Returns:

The value of the attribute.

See also:

`VAN` macro for a simpler interface.

Time complexity: O(Av), the number of vertex attributes.

12.2.1.8. `VAN` — Query a numeric vertex attribute.

```
#define VAN(graph,n,v)
```

This is shorthand for `igraph_cattribute_VAN()`.

Arguments:

graph:

The graph.

n:

The name of the attribute.

v:

The id of the vertex.

Returns:

The value of the attribute.

12.2.1.9. `igraph_cattribute_VANV` — Query a numeric vertex attribute for many vertices

```
int igraph_cattribute_VANV(const igraph_t *graph, const char *name,
      igraph_vs_t vids, igraph_vector_t *result);
```

Arguments:

graph:

The input graph.

name:

The name of the attribute.

vids:

The vertices to query.

result:

Pointer to an initialized vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

Time complexity: O(v), where v is the number of vertices in 'vids'.

12.2.1.10. VANV — Query a numeric vertex attribute for all vertices.

```
#define VANV(graph,n,vec)
```

This is a shorthand for `igraph_cattribute_VANV()`.

Arguments:

graph:

The graph.

n:

The name of the attribute.

vec:

Pointer to an initialized vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

12.2.1.11. `igraph_cattribute_VAS` — Query a string vertex attribute.

```
const char* igraph_cattribute_VAS(const igraph_t *graph, const char *name,
        igraph_integer_t vid);
```

The attribute must exist, otherwise an error is triggered.

Arguments:

graph:

The input graph.

name:

The name of the attribute.

vid:

The id of the queried vertex.

Returns:

The value of the attribute.

See also:

The macro `VAS` for a simpler interface.

Time complexity: O(Av), the number of vertex attributes.

12.2.1.12. `VAS` — Query a string vertex attribute.

```
#define VAS(graph,n,v)
```

This is shorthand for `igraph_cattribute_VAS()`.

Arguments:

graph:

The graph.

n:

The name of the attribute.

v:

The id of the vertex.

Returns:

The value of the attribute.

12.2.1.13. `igraph_cattribute_VASV` — Query a string vertex attribute for many vertices

```
int igraph_cattribute_VASV(const igraph_t *graph, const char *name,
      igraph_vs_t vids, igraph_strvector_t *result);
```

Arguments:

graph:

The input graph.

name:

The name of the attribute.

vids:

The vertices to query.

result:

Pointer to an initialized string vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

Time complexity: O(v), where v is the number of vertices in 'vids'. (We assume that the string attributes have a bounded length.)

12.2.1.14. VASV — Query a string vertex attribute for all vertices.

```
#define VASV(graph,n,vec)
```

This is a shorthand for `igraph_cattribute_VASV()`.

Arguments:

graph:

The graph.

n:

The name of the attribute.

vec:

Pointer to an initialized string vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

12.2.1.15. igraph_cattribute_EAN — Query a numeric edge attribute.

```
igraph_real_t igraph_cattribute_EAN(const igraph_t *graph, const char *name,
          igraph_integer_t eid);
```

The attribute must exist, otherwise an error is triggered.

Arguments:

`graph`:

The input graph.

`name`:

The name of the attribute.

`eid`:

The id of the queried edge.

Returns:

The value of the attribute.

See also:

`EAN` for an easier interface.

Time complexity: O(Ae), the number of edge attributes.

12.2.1.16. `EAN` — Query a numeric edge attribute.

```
#define EAN(graph,n,e)
```

This is shorthand for `igraph_cattribute_EAN()`.

Arguments:

`graph`:

The graph.

`n`:

The name of the attribute.

e:

The id of the edge.

Returns:

The value of the attribute.

12.2.1.17. `igraph_cattribute_EANV` — Query a numeric edge attribute for many edges

```
int igraph_cattribute_EANV(const igraph_t *graph, const char *name,
      igraph_es_t eids, igraph_vector_t *result);
```

Arguments:

graph:

The input graph.

name:

The name of the attribute.

eids:

The edges to query.

result:

Pointer to an initialized vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

Time complexity: O(e), where e is the number of edges in 'eids'.

12.2.1.18. `EANV` — Query a numeric edge attribute for all edges.

```
#define EANV(graph,n,vec)
```

This is a shorthand for `igraph_cattribute_EANV()`.

Arguments:

graph:

The graph.

n:

The name of the attribute.

vec:

Pointer to an initialized vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

12.2.1.19. `igraph_cattribute_EAS` — Query a string edge attribute.

```
const char* igraph_cattribute_EAS(const igraph_t *graph, const char *name,
        igraph_integer_t eid);
```

The attribute must exist, otherwise an error is triggered.

Arguments:

graph:

The input graph.

name:

The name of the attribute.

eid:

The id of the queried edge.

Returns:

The value of the attribute.

\se `EAS` if you want to type less. Time complexity: O(Ae), the number of edge attributes.

12.2.1.20. `EAS` — Query a string edge attribute.

```
#define EAS(graph,n,e)
```

This is shorthand for `igraph_cattribute_EAS()`.

Arguments:

graph:

The graph.

n:

The name of the attribute.

e:

The id of the edge.

Returns:

The value of the attribute.

12.2.1.21. `igraph_cattribute_EASV` — Query a string edge attribute for many edges

```
int igraph_cattribute_EASV(const igraph_t *graph, const char *name,
```

```
    igraph_es_t eids, igraph_strvector_t *result);
```

Arguments:

graph:

The input graph.

name:

The name of the attribute.

vids:

The edges to query.

result:

Pointer to an initialized string vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

Time complexity: O(e), where e is the number of edges in 'eids'. (We assume that the string attributes have a bounded length.)

12.2.1.22. `EASV` — Query a string edge attribute for all edges.

```
#define EASV(graph,n,vec)
```

This is a shorthand for `igraph_cattribute_EASV()`.

Arguments:

graph:

The graph.

n:

The name of the attribute.

vec:

Pointer to an initialized string vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

12.2.2. Set attributes

12.2.2.1. `igraph_cattribute_GAN_set` — Set a numeric graph attribute

```
int igraph_cattribute_GAN_set(igraph_t *graph, const char *name,
         igraph_real_t value);
```

Arguments:

graph:

The graph.

name:

Name of the graph attribute. If there is no such attribute yet, then it will be added.

value:

The (new) value of the graph attribute.

Returns:

Error code.

\se `SETGAN` if you want to type less. Time complexity: O(1).

12.2.2.2. `SETGAN` — Set a numeric graph attribute

```
#define SETGAN(graph,n,value)
```

This is a shorthand for `igraph_cattribute_GAN_set()`.

Arguments:

graph:

The graph.

n:

The name of the attribute.

value:

The new value of the attribute.

Returns:

Error code.

12.2.2.3. `igraph_cattribute_GAS_set` — Set a string graph attribute.

```
int igraph_cattribute_GAS_set(igraph_t *graph, const char *name,
          const char *value);
```

Arguments:

graph:

The graph.

name:

Name of the graph attribute. If there is no such attribute yet, then it will be added.

value:

The (new) value of the graph attribute. It will be copied.

Returns:

Error code.

\se `SETGAS` if you want to type less. Time complexity: O(1).

12.2.2.4. `SETGAS` — Set a string graph attribute

```
#define SETGAS(graph,n,value)
```

This is a shorthand for `igraph_cattribute_GAS_set()`.

Arguments:

graph:

The graph.

n:

The name of the attribute.

value:

The new value of the attribute.

Returns:

Error code.

12.2.2.5. `igraph_cattribute_VAN_set` — Set a numeric vertex attribute

```
int igraph_cattribute_VAN_set(igraph_t *graph, const char *name,
          igraph_integer_t vid, igraph_real_t value);
```

The attribute will be added if not present already. If present it will be overwritten. The same `value` is set for all vertices included in `vid`.

Arguments:

`graph`:

The graph.

`name`:

Name of the attribute.

`vid`:

Vertices for which to set the attribute.

`value`:

The (new) value of the attribute.

Returns:

Error code.

See also:

`SETVAN` for a simpler way.

Time complexity: O(n), the number of vertices if the attribute is new, O(|vid|) otherwise.

12.2.2.6. SETVAN — Set a numeric vertex attribute

```
#define SETVAN(graph,n,vid,value)
```

This is a shorthand for `igraph_cattribute_VAN_set()`.

Arguments:

graph:

The graph.

n:

The name of the attribute.

vid:

Ids of the vertices to set.

value:

The new value of the attribute.

Returns:

Error code.

12.2.2.7. `igraph_cattribute_VAS_set` — Set a string vertex attribute

```
int igraph_cattribute_VAS_set(igraph_t *graph, const char *name,
         igraph_integer_t vid, const char *value);
```

The attribute will be added if not present already. If present it will be overwritten. The same *value* is set for all vertices included in *vid*.

Arguments:

graph:

The graph.

name:

Name of the attribute.

vid:

Vertices for which to set the attribute.

value:

The (new) value of the attribute.

Returns:

Error code.

See also:

`SETVAS` for a simpler way.

Time complexity: O(n*l), n is the number of vertices, l is the length of the string to set. If the attribute if not new then only O(|vid|*l).

12.2.2.8. `SETVAS` — Set a string vertex attribute

```
#define SETVAS(graph,n,vid,value)
```

This is a shorthand for `igraph_cattribute_VAS_set()`.

Arguments:

`graph`:

The graph.

`n`:

The name of the attribute.

`vid`:

Ids of the vertices to set.

`value`:

The new value of the attribute.

Returns:

Error code.

12.2.2.9. `igraph_cattribute_EAN_set` — Set a numeric edge attribute

```
int igraph_cattribute_EAN_set(igraph_t *graph, const char *name,
        igraph_integer_t eid, igraph_real_t value);
```

The attribute will be added if not present already. If present it will be overwritten. The same *`value`* is set for all edges included in *`vid`*.

Arguments:

`graph`:

The graph.

`name`:

Name of the attribute.

`eid`:

Edges for which to set the attribute.

`value`:

The (new) value of the attribute.

Returns:

Error code.

See also:

`SETEAN` for a simpler way.

Time complexity: O(e), the number of edges if the attribute is new, O(|eid|) otherwise.

12.2.2.10. `SETEAN` — Set a numeric edge attribute

```
#define SETEAN(graph,n,eid,value)
```

This is a shorthand for `igraph_cattribute_EAN_set()`.

Arguments:

graph:

The graph.

n:

The name of the attribute.

eid:

Ids of the edges to set.

value:

The new value of the attribute.

Returns:

Error code.

12.2.2.11. igraph_cattribute_EAS_set — Set a string edge attribute

```
int igraph_cattribute_EAS_set(igraph_t *graph, const char *name,
         igraph_integer_t eid, const char *value);
```

The attribute will be added if not present already. If present it will be overwritten. The same *value* is set for all edges included in *vid*.

Arguments:

graph:

The graph.

name:

Name of the attribute.

eid:

Edges for which to set the attribute.

value:

The (new) value of the attribute.

Returns:

Error code.

See also:

`SETEAS` for a simpler way.

Time complexity: O(e*l), n is the number of edges, l is the length of the string to set. If the attribute if not new then only O(|eid|*l).

12.2.2.12. SETEAS — Set a string edge attribute

```
#define SETEAS(graph,n,eid,value)
```

This is a shorthand for `igraph_cattribute_EAS_set()`.

Arguments:

graph:

The graph.

n:

The name of the attribute.

eid:

Ids of the edges to set.

value:

The new value of the attribute.

Returns:

Error code.

12.2.2.13. `igraph_cattribute_VAN_setv` — Set a numeric vertex attribute for all vertices.

```
int igraph_cattribute_VAN_setv(igraph_t *graph, const char *name,
          const igraph_vector_t *v);
```

The attribute will be added if not present yet.

Arguments:

graph:

The graph.

name:

Name of the attribute.

v:

The new attribute values. The length of this vector must match the number of vertices.

Returns:

Error code.

See also:

`SETVANV` for a simpler way.

Time complexity: O(n), the number of vertices.

12.2.2.14. `SETVANV` — Set a numeric vertex attribute for all vertices

```
#define SETVANV(graph,n,v)
```

This is a shorthand for `igraph_cattribute_VAN_setv()`.

Arguments:

graph:

The graph.

n:

The name of the attribute.

v:

Vector containing the new values of the attributes.

Returns:

Error code.

12.2.2.15. `igraph_cattribute_VAS_setv` — Set a string vertex attribute for all vertices.

```
int igraph_cattribute_VAS_setv(igraph_t *graph, const char *name,
            const igraph_strvector_t *sv);
```

The attribute will be added if not present yet.

Arguments:

graph:

The graph.

name:

Name of the attribute.

sv:

String vector, the new attribute values. The length of this vector must match the number of vertices.

Returns:

Error code.

See also:

`SETVASV` for a simpler way.

Time complexity: O(n+l), n is the number of vertices, l is the total length of the strings.

12.2.2.16. `SETVASV` — Set a string vertex attribute for all vertices

```
#define SETVASV(graph,n,v)
```

This is a shorthand for `igraph_cattribute_VAS_setv()`.

Arguments:

`graph`:

The graph.

`n`:

The name of the attribute.

`v`:

Vector containing the new values of the attributes.

Returns:

Error code.

12.2.2.17. `igraph_cattribute_EAN_setv` — Set a numeric edge attribute for all vertices.

```
int igraph_cattribute_EAN_setv(igraph_t *graph, const char *name,
          const igraph_vector_t *v);
```

The attribute will be added if not present yet.

Arguments:

`graph`:

The graph.

`name`:

Name of the attribute.

`v`:

The new attribute values. The length of this vector must match the number of edges.

Returns:

Error code.

See also:

`SETEANV` for a simpler way.

Time complexity: O(e), the number of edges.

12.2.2.18. `SETEANV` — Set a numeric edge attribute for all vertices

```
#define SETEANV(graph,n,v)
```

This is a shorthand for `igraph_cattribute_EAN_setv()`.

Arguments:

graph:

The graph.

n:

The name of the attribute.

v:

Vector containing the new values of the attributes.

12.2.2.19. `igraph_cattribute_EAS_setv` — Set a string edge attribute for all vertices.

```
int igraph_cattribute_EAS_setv(igraph_t *graph, const char *name,
          const igraph_strvector_t *sv);
```

The attribute will be added if not present yet.

Arguments:

graph:

The graph.

name:

Name of the attribute.

sv:

String vector, the new attribute values. The length of this vector must match the number of edges.

Returns:

Error code.

See also:

SETEASV for a simpler way.

Time complexity: O(e+l), e is the number of edges, l is the total length of the strings.

12.2.2.20. SETEASV — Set a string edge attribute for all vertices

```
#define SETEASV(graph,n,v)
```

This is a shorthand for `igraph_cattribute_EAS_setv()`.

Arguments:

graph:

The graph.

n:

The name of the attribute.

v:

Vector containing the new values of the attributes.

12.2.3. Remove attributes

12.2.3.1. igraph_cattribute_remove_g — Remove a graph attribute

```
void igraph_cattribute_remove_g(igraph_t *graph, const char *name);
```

Arguments:

graph:

The graph object.

name:

Name of the graph attribute to remove.

See also:

`DELGA` for a simpler way.

12.2.3.2. `DELGA` — Remove a graph attribute.

```
#define DELGA(graph,n)
```

A shorthand for `igraph_cattribute_remove_g()`.

Arguments:

graph:

The graph.

n:

The name of the attribute to remove.

12.2.3.3. `igraph_cattribute_remove_v` — Remove a vertex attribute

```
void igraph_cattribute_remove_v(igraph_t *graph, const char *name);
```

Arguments:

graph:

The graph object.

name:

Name of the vertex attribute to remove.

See also:

`DELVA` for a simpler way.

12.2.3.4. `DELVA` — Remove a vertex attribute.

```
#define DELVA(graph,n)
```

A shorthand for `igraph_cattribute_remove_v()`.

Arguments:

graph:

The graph.

n:

The name of the attribute to remove.

12.2.3.5. `igraph_cattribute_remove_e` — Remove an edge attribute

```
void igraph_cattribute_remove_e(igraph_t *graph, const char *name);
```

Arguments:

graph:

The graph object.

name:

Name of the edge attribute to remove.

See also:

`DELEA` for a simpler way.

12.2.3.6. `DELEA` — Remove an edge attribute.

```
#define DELEA(graph,n)
```

A shorthand for `igraph_cattribute_remove_e()`.

Arguments:

graph:

The graph.

n:

The name of the attribute to remove.

12.2.3.7. `igraph_cattribute_remove_all` — Remove all graph/vertex/edge attributes

```
void igraph_cattribute_remove_all(igraph_t *graph, igraph_bool_t g,
      igraph_bool_t v, igraph_bool_t e);
```

Arguments:

graph:

The graph object.

g:

Boolean, whether to remove graph attributes.

v:

Boolean, whether to remove vertex attributes.

e:

Boolean, whether to remove edge attributes.

See also:

`DELGAS`, `DELVAS`, `DELEAS`, `DELALL` for simpler ways.

12.2.3.8. `DELGAS` — Remove all graph attributes.

```
#define DELGAS(graph)
```

Calls `igraph_cattribute_remove_all()`.

Arguments:

graph:

The graph.

12.2.3.9. `DELVAS` — Remove all vertex attributes.

```
#define DELVAS(graph)
```

Calls `igraph_cattribute_remove_all()`.

Arguments:

graph:

The graph.

12.2.3.10. DELEAS — Remove all edge attributes.

```
#define DELEAS(graph)
```

Calls `igraph_cattribute_remove_all()`.

Arguments:

graph:

The graph.

12.2.3.11. DELALL — Remove all attributes.

```
#define DELALL(graph)
```

All graph, vertex and edges attributes will be removed. Calls `igraph_cattribute_remove_all()`.

Arguments:

graph:

The graph.

Chapter 13. Structural Properties of Graphs

These functions usually calculate some structural property of a graph, like its diameter, the degree of the nodes, etc.

13.1. Basic Properties

13.1.1. `igraph_are_connected` — Decides whether two vertices are connected

```
int igraph_are_connected(const igraph_t *graph,
    igraph_integer_t v1, igraph_integer_t v2,
    igraph_bool_t *res);
```

Arguments:

`graph`:

The graph object.

`v1`:

The first vertex.

`v2`:

The second vertex.

`res`:

Boolean, `TRUE` if there is an edge from *`v1`* to *`v2`*, `FALSE` otherwise.

Returns:

The error code `IGRAPH_EINVVID` is returned if an invalid vertex ID is given.

The function is of course symmetric for undirected graphs.

Time complexity: O(min(log(d1), log(d2))), d1 is the (out-)degree of *`v1`* and d2 is the (in-)degree of *`v2`*.

13.2. Shortest Path Related Functions

13.2.1. `igraph_shortest_paths` — The length of the shortest paths between vertices.

```
int igraph_shortest_paths(const igraph_t *graph, igraph_matrix_t *res,
     const igraph_vs_t from, const igraph_vs_t to,
     igraph_neimode_t mode);
```

Arguments:

graph:

The graph object.

res:

The result of the calculation, a matrix. A pointer to an initialized matrix, to be more precise. The matrix will be resized if needed. It will have the same number of rows as the length of the `from` argument, and its number of columns is the number of vertices in the `to` argument. One row of the matrix shows the distances from/to a given vertex to the ones in `to`. For the unreachable vertices IGRAPH_INFINITY is returned.

from:

Vector of the vertex ids for which the path length calculations are done.

to:

Vector of the vertex ids to which the path length calculations are done. It is not allowed to have duplicated vertex ids here.

mode:

The type of shortest paths to be used for the calculation in directed graphs. Possible values:

`IGRAPH_OUT`

the lengths of the outgoing paths are calculated.

`IGRAPH_IN`

the lengths of the incoming paths are calculated.

`IGRAPH_ALL`

the directed graph is considered as an undirected one for the computation.

Returns:

Error code:

`IGRAPH_ENOMEM`

not enough memory for temporary data.

`IGRAPH_EINVVID`

invalid vertex id passed.

`IGRAPH_EINVMODE`

invalid mode argument.

Time complexity: O(n(|V|+|E|)), n is the number of vertices to calculate, |V| and |E| are the number of vertices and edges in the graph.

See also:

`igraph_get_shortest_paths()` to get the paths themselves, `igraph_shortest_paths_dijkstra()` for the weighted version.

13.2.2. `igraph_shortest_paths_dijkstra` — Weighted shortest paths from some sources.

```
int igraph_shortest_paths_dijkstra(const igraph_t *graph,
       igraph_matrix_t *res,
       const igraph_vs_t from,
       const igraph_vs_t to,
       const igraph_vector_t *weights,
       igraph_neimode_t mode);
```

This function is Dijkstra's algorithm to find the weighted shortest paths to all vertices from a single source. (It is run independently for the given sources.) It uses a binary heap for efficient implementation.

Arguments:

`graph`:

The input graph, can be directed.

`res`:

The result, a matrix. A pointer to an initialized matrix should be passed here. The matrix will be resized as needed. Each row contains the distances from a single source, to the vertices given in the `to` argument. Unreachable vertices has distance `IGRAPH_INFINITY`.

`from`:

The source vertices.

`to`:

The target vertices. It is not allowed to include a vertex twice or more.

`weights`:

The edge weights. They must be all non-negative for Dijkstra's algorithm to work. An error code is returned if there is a negative edge weight in the weight vector. If this is a null pointer, then the unweighted version, `igraph_shortest_paths()` is called.

`mode`:

For directed graphs; whether to follow paths along edge directions (`IGRAPH_OUT`), or the opposite (`IGRAPH_IN`), or ignore edge directions completely (`IGRAPH_ALL`). It is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(s*|E|log|E|+|V|), where |V| is the number of vertices, |E| the number of edges and s the number of sources.

See also:

`igraph_shortest_paths()` for a (slightly) faster unweighted version or `igraph_shortest_paths_bellman_ford()` for a weighted variant that works in the presence of negative edge weights (but no negative loops).

Example 13-1. File examples/simple/dijkstra.c

13.2.3. `igraph_shortest_paths_bellman_ford` — Weighted shortest paths from some sources allowing negative weights.

```
int igraph_shortest_paths_bellman_ford(const igraph_t *graph,
           igraph_matrix_t *res,
           const igraph_vs_t from,
           const igraph_vs_t to,
           const igraph_vector_t *weights,
           igraph_neimode_t mode);
```

This function is the Bellman-Ford algorithm to find the weighted shortest paths to all vertices from a single source. (It is run independently for the given sources.). If there are no negative weights, you are better off with `igraph_shortest_paths_dijkstra()` .

Arguments:

graph:

The input graph, can be directed.

res:

The result, a matrix. A pointer to an initialized matrix should be passed here, the matrix will be resized if needed. Each row contains the distances from a single source, to all vertices in the graph, in the order of vertex ids. For unreachable vertices the matrix contains `IGRAPH_INFINITY`.

from:

The source vertices.

weights:

The edge weights. There mustn't be any closed loop in the graph that has a negative total weight (since this would allow us to decrease the weight of any path containing at least a single vertex of this loop infinitely). If this is a null pointer, then the unweighted version, `igraph_shortest_paths()` is called.

mode:

For directed graphs; whether to follow paths along edge directions (`IGRAPH_OUT`), or the opposite (`IGRAPH_IN`), or ignore edge directions completely (`IGRAPH_ALL`). It is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(s*|E|*|V|), where |V| is the number of vertices, |E| the number of edges and s the number of sources.

See also:

`igraph_shortest_paths()` for a faster unweighted version or `igraph_shortest_paths_dijkstra()` if you do not have negative edge weights.

Example 13-2. File examples/simple/bellman_ford.c

13.2.4. `igraph_shortest_paths_johnson` — Calculate shortest paths from some sources using Johnson's algorithm.

```
int igraph_shortest_paths_johnson(const igraph_t *graph,
      igraph_matrix_t *res,
      const igraph_vs_t from,
      const igraph_vs_t to,
      const igraph_vector_t *weights);
```

See Wikipedia at http://en.wikipedia.org/wiki/Johnson's_algorithm for Johnson's algorithm. This algorithm works even if the graph contains negative edge weights, and it is worth using it if we calculate the shortest paths from many sources.

If no edge weights are supplied, then the unweighted version, `igraph_shortest_paths()` is called.

If all the supplied edge weights are non-negative, then Dijkstra's algorithm is used by calling `igraph_shortest_paths_dijkstra()`.

Arguments:

graph:

The input graph, typically it is directed.

res:

Pointer to an initialized matrix, the result will be stored here, one line for each source vertex, one column for each target vertex.

from:

The source vertices.

to:

The target vertices. It is not allowed to include a vertex twice or more.

weights:

Optional edge weights. If it is a null-pointer, then the unweighted breadth-first search based `igraph_shortest_paths()` will be called.

Returns:

Error code.

Time complexity: O(s|V|log|V|+|V||E|), |V| and |E| are the number of vertices and edges, s is the number of source vertices.

See also:

`igraph_shortest_paths()` for a faster unweighted version or `igraph_shortest_paths_dijkstra()` if you do not have negative edge weights, `igraph_shortest_paths_bellman_ford()` if you only need to calculate shortest paths from a couple of sources.

13.2.5. `igraph_get_shortest_paths` — Calculates the shortest paths from/to one vertex.

```
int igraph_get_shortest_paths(const igraph_t *graph,
          igraph_vector_ptr_t *vertices,
          igraph_vector_ptr_t *edges,
          igraph_integer_t from, const igraph_vs_t to,
          igraph_neimode_t mode);
```

If there is more than one geodesic between two vertices, this function gives only one of them.

Arguments:

`graph`:

The graph object.

`vertices`:

The result, the ids of the vertices along the paths. This is a pointer vector, each element points to a vector object. These should be initialized before passing them to the function, which will properly clear and/or resize them and fill the ids of the vertices along the geodesics from/to the vertices. Supply a null pointer here if you don't need these vectors. Normally, either this argument, or the `edges` should be non-null, but no error or warning is given if they are both null pointers.

`edges`:

The result, the ids of the edges along the paths. This is a pointer vector, each element points to a vector object. These should be initialized before passing them to the function, which will properly clear and/or resize them and fill the ids of the vertices along the geodesics from/to the vertices. Supply a null pointer here if you don't need these vectors. Normally, either this argument, or the `vertices` should be non-null, but no error or warning is given if they are both null pointers.

`from`:

The id of the vertex from/to which the geodesics are calculated.

`to`:

Vertex sequence with the ids of the vertices to/from which the shortest paths will be calculated. A vertex might be given multiple times.

`mode`:

The type of shortest paths to be used for the calculation in directed graphs. Possible values:

`IGRAPH_OUT`

the outgoing paths are calculated.

`IGRAPH_IN`

the incoming paths are calculated.

`IGRAPH_ALL`

the directed graph is considered as an undirected one for the computation.

Returns:

Error code:

`IGRAPH_ENOMEM`

not enough memory for temporary data.

`IGRAPH_EINVVID`

from is invalid vertex id, or the length of *to* is not the same as the length of *res*.

`IGRAPH_EINVMODE`

invalid mode argument.

Time complexity: O(|V|+|E|), |V| is the number of vertices, |E| the number of edges in the graph.

See also:

`igraph_shortest_paths()` if you only need the path length but not the paths themselves.

Example 13-3. File examples/simple/igraph_get_shortest_paths.c

13.2.6. `igraph_get_shortest_path` — Shortest path from one vertex to another one.

```
int igraph_get_shortest_path(const igraph_t *graph,
        igraph_vector_t *vertices,
        igraph_vector_t *edges,
        igraph_integer_t from,
        igraph_integer_t to,
        igraph_neimode_t mode);
```

Calculates and returns a single unweighted shortest path from a given vertex to another one. If there are more than one shortest paths between the two vertices, then an arbitrary one is returned.

This function is a wrapper to `igraph_get_shortest_paths()`, for the special case when only one target vertex is considered.

Arguments:

`graph`:

The input graph, it can be directed or undirected. Directed paths are considered in directed graphs.

`vertices`:

Pointer to an initialized vector or a null pointer. If not a null pointer, then the vertex ids along the path are stored here, including the source and target vertices.

`edges`:

Pointer to an uninitialized vector or a null pointer. If not a null pointer, then the edge ids along the path are stored here.

`from`:

The id of the source vertex.

`to`:

The id of the target vertex.

`mode`:

A constant specifying how edge directions are considered in directed graphs. Valid modes are: `IGRAPH_OUT`, follows edge directions; `IGRAPH_IN`, follows the opposite directions; and `IGRAPH_ALL`, ignores edge directions. This argument is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges in the graph.

See also:

`igraph_get_shortest_paths()` for the version with more target vertices.

13.2.7. `igraph_get_shortest_paths_dijkstra` — Calculates the weighted shortest paths from/to one vertex.

```
int igraph_get_shortest_paths_dijkstra(const igraph_t *graph,
                                       igraph_vector_ptr_t *vertices,
           igraph_vector_ptr_t *edges,
           igraph_integer_t from,
           igraph_vs_t to,
           const igraph_vector_t *weights,
           igraph_neimode_t mode);
```

If there is more than one path with the smallest weight between two vertices, this function gives only one of them.

Arguments:

graph:

The graph object.

vertices:

The result, the ids of the vertices along the paths. This is a pointer vector, each element points to a vector object. These should be initialized before passing them to the function, which will properly clear and/or resize them and fill the ids of the vertices along the geodesics from/to the vertices. Supply a null pointer here if you don't need these vectors. Normally, either this argument, or the `edges` should be non-null, but no error or warning is given if they are both null pointers.

edges:

The result, the ids of the edges along the paths. This is a pointer vector, each element points to a vector object. These should be initialized before passing them to the function, which will properly clear and/or resize them and fill the ids of the vertices along the geodesics from/to the vertices. Supply a null pointer here if you don't need these vectors. Normally, either this argument, or the `vertices` should be non-null, but no error or warning is given if they are both null pointers.

from:

The id of the vertex from/to which the geodesics are calculated.

to:

Vertex sequence with the ids of the vertices to/from which the shortest paths will be calculated. A vertex might be given multiple times.

weights:

a vector holding the edge weights. All weights must be positive.

mode:

The type of shortest paths to be use for the calculation in directed graphs. Possible values:

`IGRAPH_OUT`

the outgoing paths are calculated.

`IGRAPH_IN`

the incoming paths are calculated.

`IGRAPH_ALL`

the directed graph is considered as an undirected one for the computation.

Returns:

Error code:

`IGRAPH_ENOMEM`

not enough memory for temporary data.

`IGRAPH_EINVVID`

from is invalid vertex id, or the length of *to* is not the same as the length of *res*.

`IGRAPH_EINVMODE`

invalid mode argument.

Time complexity: O(|E|log|E|+|V|), where |V| is the number of vertices and |E| is the number of edges

See also:

`igraph_shortest_paths_dijkstra()` if you only need the path length but not the paths themselves, `igraph_get_shortest_paths()` if all edge weights are equal.

Example 13-4. File examples/simple/igraph_get_shortest_paths_dijkstra.c

13.2.8. `igraph_get_shortest_path_dijkstra` — Weighted shortest path from one vertex to another one.

```
int igraph_get_shortest_path_dijkstra(const igraph_t *graph,
          igraph_vector_t *vertices,
          igraph_vector_t *edges,
          igraph_integer_t from,
          igraph_integer_t to,
          const igraph_vector_t *weights,
          igraph_neimode_t mode);
```

Calculates a single (positively) weighted shortest path from a single vertex to another one, using Dijkstra's algorithm.

This function is a special case (and a wrapper) to `igraph_get_shortest_paths_dijkstra()`.

Arguments:

graph:

The input graph, it can be directed or undirected.

vertices:

Pointer to an initialized vector or a null pointer. If not a null pointer, then the vertex ids along the path are stored here, including the source and target vertices.

edges:

Pointer to an uninitialized vector or a null pointer. If not a null pointer, then the edge ids along the path are stored here.

from:

The id of the source vertex.

to:

The id of the target vertex.

weights:

Vector of edge weights, in the order of edge ids. They must be non-negative, otherwise the algorithm does not work.

mode:

A constant specifying how edge directions are considered in directed graphs. `IGRAPH_OUT` follows edge directions, `IGRAPH_IN` follows the opposite directions, and `IGRAPH_ALL` ignores edge directions. This argument is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|E|log|E|+|V|), |V| is the number of vertices, |E| is the number of edges in the graph.

See also:

`igraph_get_shortest_paths_dijkstra()` for the version with more target vertices.

13.2.9. `igraph_get_all_shortest_paths` — Finds all shortest paths (geodesics) from a vertex to all other vertices.

```
int igraph_get_all_shortest_paths(const igraph_t *graph,
      igraph_vector_ptr_t *res,
      igraph_vector_t *nrgeo,
      igraph_integer_t from, const igraph_vs_t to,
      igraph_neimode_t mode);
```

Arguments:

`graph`:

The graph object.

`res`:

Pointer to an initialized pointer vector, the result will be stored here in igraph_vector_t objects. Each vector object contains the vertices along a shortest path from *`from`* to another vertex. The vectors are ordered according to their target vertex: first the shortest paths to vertex 0, then to vertex 1, etc. No data is included for unreachable vertices.

`nrgeo`:

Pointer to an initialized igraph_vector_t object or NULL. If not NULL the number of shortest paths from *`from`* are stored here for every vertex in the graph. Note that the values will be accurate only for those vertices that are in the target vertex sequence (see *`to`*), since the search terminates as soon as all the target vertices have been found.

`from`:

The id of the vertex from/to which the geodesics are calculated.

to:

Vertex sequence with the ids of the vertices to/from which the shortest paths will be calculated. A vertex might be given multiple times.

mode:

The type of shortest paths to be use for the calculation in directed graphs. Possible values:

`IGRAPH_OUT`

the lengths of the outgoing paths are calculated.

`IGRAPH_IN`

the lengths of the incoming paths are calculated.

`IGRAPH_ALL`

the directed graph is considered as an undirected one for the computation.

Returns:

Error code:

`IGRAPH_ENOMEM`

not enough memory for temporary data.

`IGRAPH_EINVVID`

from is invalid vertex id.

`IGRAPH_EINVMODE`

invalid mode argument.

Added in version 0.2.

Time complexity: O(|V|+|E|) for most graphs, O(|V|^2) in the worst case.

13.2.10. `igraph_get_all_shortest_paths_dijkstra` — Finds all shortest paths (geodesics) from a vertex to all other vertices.

```
int igraph_get_all_shortest_paths_dijkstra(const igraph_t *graph,
```

```
igraph_vector_ptr_t *res,
igraph_vector_t *nrgeo,
igraph_integer_t from, igraph_vs_t to,
const igraph_vector_t *weights,
igraph_neimode_t mode);
```

Arguments:

graph:

The graph object.

res:

Pointer to an initialized pointer vector, the result will be stored here in igraph_vector_t objects. Each vector object contains the vertices along a shortest path from *from* to another vertex. The vectors are ordered according to their target vertex: first the shortest paths to vertex 0, then to vertex 1, etc. No data is included for unreachable vertices.

nrgeo:

Pointer to an initialized igraph_vector_t object or NULL. If not NULL the number of shortest paths from *from* are stored here for every vertex in the graph. Note that the values will be accurate only for those vertices that are in the target vertex sequence (see *to*), since the search terminates as soon as all the target vertices have been found.

from:

The id of the vertex from/to which the geodesics are calculated.

to:

Vertex sequence with the ids of the vertices to/from which the shortest paths will be calculated. A vertex might be given multiple times.

weights:

a vector holding the edge weights. All weights must be non-negative.

mode:

The type of shortest paths to be use for the calculation in directed graphs. Possible values:

`IGRAPH_OUT`

the outgoing paths are calculated.

`IGRAPH_IN`

the incoming paths are calculated.

`IGRAPH_ALL`

the directed graph is considered as an undirected one for the computation.

Returns:

Error code:

`IGRAPH_ENOMEM`

not enough memory for temporary data.

`IGRAPH_EINVVID`

`from` is invalid vertex id, or the length of *`to`* is not the same as the length of *`res`*.

`IGRAPH_EINVMODE`

invalid mode argument.

Time complexity: O(|E|log|E|+|V|), where |V| is the number of vertices and |E| is the number of edges

See also:

`igraph_shortest_paths_dijkstra()` if you only need the path length but not the paths themselves, `igraph_get_all_shortest_paths()` if all edge weights are equal.

Example 13-5. File examples/simple/igraph_get_all_shortest_paths_dijkstra.c

13.2.11. `igraph_average_path_length` — Calculates the average geodesic length in a graph.

```
int igraph_average_path_length(const igraph_t *graph, igraph_real_t *res,
          igraph_bool_t directed, igraph_bool_t unconn);
```

Arguments:

graph:

The graph object.

res:

Pointer to a real number, this will contain the result.

directed:

Boolean, whether to consider directed paths. Ignored for undirected graphs.

unconn:

What to do if the graph is not connected. If `TRUE` the average of the geodesics within the components will be returned, otherwise the number of vertices is used for the length of non-existing geodesics. (The rationale behind this is that this is always longer than the longest possible geodesic in a graph.)

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for data structures

Time complexity: O(|V||E|), the number of vertices times the number of edges.

Example 13-6. File examples/simple/igraph_average_path_length.c

13.2.12. `igraph_path_length_hist` — Create a histogram of all shortest path lengths.

```
int igraph_path_length_hist(const igraph_t *graph, igraph_vector_t *res,
       igraph_real_t *unconnected, igraph_bool_t directed);
```

This function calculates a histogram, by calculating the shortest path length between each pair of vertices. For directed graphs both directions might be considered and then every pair of vertices appears twice in the histogram.

Arguments:

graph:

The input graph.

res:

Pointer to an initialized vector, the result is stored here. The first (i.e. zeroth) element contains the number of shortest paths of length 1, etc. The supplied vector is resized as needed.

unconnected:

Pointer to a real number, the number of pairs for which the second vertex is not reachable from the first is stored here.

directed:

Whether to consider directed paths in a directed graph (if not zero). This argument is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|V||E|), the number of vertices times the number of edges.

See also:

`igraph_average_path_length()` and `igraph_shortest_paths()`

13.2.13. `igraph_diameter` — Calculates the diameter of a graph (longest geodesic).

```
int igraph_diameter(const igraph_t *graph, igraph_integer_t *pres,
      igraph_integer_t *pfrom, igraph_integer_t *pto,
      igraph_vector_t *path,
      igraph_bool_t directed, igraph_bool_t unconn);
```

Arguments:

graph:

The graph object.

pres:

Pointer to an integer, if not `NULL` then it will contain the diameter (the actual distance).

pfrom:

Pointer to an integer, if not `NULL` it will be set to the source vertex of the diameter path.

pto:

Pointer to an integer, if not `NULL` it will be set to the target vertex of the diameter path.

path:

Pointer to an initialized vector. If not `NULL` the actual longest geodesic path will be stored here. The vector will be resized as needed.

directed:

Boolean, whether to consider directed paths. Ignored for undirected graphs.

unconn:

What to do if the graph is not connected. If `TRUE` the longest geodesic within a component will be returned, otherwise the number of vertices is returned. (The rationale behind the latter is that this is always longer than the longest possible diameter in a graph.)

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data.

Time complexity: O(|V||E|), the number of vertices times the number of edges.

Example 13-7. File examples/simple/igraph_diameter.c

13.2.14. `igraph_diameter_dijkstra` — Weighted diameter using Dijkstra's algorithm, non-negative weights only.

```
int igraph_diameter_dijkstra(const igraph_t *graph,
```

```
const igraph_vector_t *weights,
igraph_real_t *pres,
igraph_integer_t *pfrom,
igraph_integer_t *pto,
igraph_vector_t *path,
igraph_bool_t directed,
igraph_bool_t unconn);
```

The diameter of a graph is its longest geodesic. I.e. the (weighted) shortest path is calculated for all pairs of vertices and the longest one is the diameter.

Arguments:

graph:

The input graph, can be directed or undirected.

pres:

Pointer to a real number, if not `NULL` then it will contain the diameter (the actual distance).

pfrom:

Pointer to an integer, if not `NULL` it will be set to the source vertex of the diameter path.

pto:

Pointer to an integer, if not `NULL` it will be set to the target vertex of the diameter path.

path:

Pointer to an initialized vector. If not `NULL` the actual longest geodesic path will be stored here. The vector will be resized as needed.

directed:

Boolean, whether to consider directed paths. Ignored for undirected graphs.

unconn:

What to do if the graph is not connected. If `TRUE` the longest geodesic within a component will be returned, otherwise `IGRAPH_INFINITY` is returned.

Returns:

Error code.

Time complexity: O(|V||E|*log|E|), |V| is the number of vertices, |E| is the number of edges.

13.2.15. `igraph_girth` — The girth of a graph is the length of the shortest circle in it.

```
int igraph_girth(const igraph_t *graph, igraph_integer_t *girth,
   igraph_vector_t *circle);
```

The current implementation works for undirected graphs only, directed graphs are treated as undirected graphs. Loop edges and multiple edges are ignored.

If the graph is a forest (ie. acyclic), then zero is returned.

This implementation is based on Alon Itai and Michael Rodeh: Finding a minimum circuit in a graph *Proceedings of the ninth annual ACM symposium on Theory of computing* , 1-10, 1977. The first implementation of this function was done by Keith Briggs, thanks Keith.

Arguments:

graph:

The input graph.

girth:

Pointer to an integer, if not `NULL` then the result will be stored here.

circle:

Pointer to an initialized vector, the vertex ids in the shortest circle will be stored here. If `NULL` then it is ignored.

Returns:

Error code.

Time complexity: O((|V|+|E|)^2), |V| is the number of vertices, |E| is the number of edges in the general case. If the graph has no circles at all then the function needs O(|V|+|E|) time to realize this and then it stops.

Example 13-8. File examples/simple/igraph_girth.c

13.2.16. `igraph_eccentricity` — Eccentricity of some vertices

```
int igraph_eccentricity(const igraph_t *graph,
   igraph_vector_t *res,
   igraph_vs_t vids,
   igraph_neimode_t mode);
```

The eccentricity of a vertex is calculated by measuring the shortest distance from (or to) the vertex, to (or from) all vertices in the graph, and taking the maximum.

This implementation ignores vertex pairs that are in different components. Isolated vertices have eccentricity zero.

Arguments:

graph:

The input graph, it can be directed or undirected.

res:

Pointer to an initialized vector, the result is stored here.

vids:

The vertices for which the eccentricity is calculated.

mode:

What kind of paths to consider for the calculation: `IGRAPH_OUT`, paths that follow edge directions; `IGRAPH_IN`, paths that follow the opposite directions; and `IGRAPH_ALL`, paths that ignore edge directions. This argument is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(v*(|V|+|E|)), where |V| is the number of vertices, |E| is the number of edges and v is the number of vertices for which eccentricity is calculated.

See also:

`igraph_radius()`.

Example 13-9. File examples/simple/igraph_eccentricity.c

13.2.17. `igraph_radius` — Radius of a graph

```
int igraph_radius(const igraph_t *graph, igraph_real_t *radius,
    igraph_neimode_t mode);
```

The radius of a graph is the defined as the minimum eccentricity of its vertices, see `igraph_eccentricity()`.

Arguments:

`graph`:

The input graph, it can be directed or undirected.

`radius`:

Pointer to a real variable, the result is stored here.

`mode`:

What kind of paths to consider for the calculation: `IGRAPH_OUT`, paths that follow edge directions; `IGRAPH_IN`, paths that follow the opposite directions; and `IGRAPH_ALL`, paths that ignore edge directions. This argument is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|V|(|V|+|E|)), where |V| is the number of vertices and |E| is the number of edges.

See also:

`igraph_eccentricity()`.

Example 13-10. File examples/simple/igraph_radius.c

13.3. Neighborhood of a vertex

13.3.1. `igraph_neighborhood_size` — Calculates the size of the neighborhood of a given vertex.

```
int igraph_neighborhood_size(const igraph_t *graph, igraph_vector_t *res,
        igraph_vs_t vids, igraph_integer_t order,
        igraph_neimode_t mode);
```

The neighborhood of a given order of a vertex includes all vertices which are closer to the vertex than the order. Ie. order 0 is always the vertex itself, order 1 is the vertex plus its immediate neighbors, order 2 is order 1 plus the immediate neighbors of the vertices in order 1, etc.

This function calculates the size of the neighborhood of the given order for the given vertices.

Arguments:

graph:

The input graph.

res:

Pointer to an initialized vector, the result will be stored here. It will be resized as needed.

vids:

The vertices for which the calculation is performed.

order:

Integer giving the order of the neighborhood.

mode:

Specifies how to use the direction of the edges if a directed graph is analyzed. For `IGRAPH_OUT` only the outgoing edges are followed, so all vertices reachable from the source vertex in at most `order` steps are counted. For `IGRAPH_IN` all vertices from which the source vertex is reachable in at most `order` steps are counted. `IGRAPH_ALL` ignores the direction of the edges. This argument is ignored for undirected graphs.

Returns:

Error code.

See also:

`igraph_neighborhood()` for calculating the actual neighborhood, `igraph_neighborhood_graphs()` for creating separate graphs from the neighborhoods.

Time complexity: O(n*d*o), where n is the number vertices for which the calculation is performed, d is the average degree, o is the order.

13.3.2. `igraph_neighborhood` — Calculate the neighborhood of vertices.

```
int igraph_neighborhood(const igraph_t *graph, igraph_vector_ptr_t *res,
   igraph_vs_t vids, igraph_integer_t order,
   igraph_neimode_t mode);
```

The neighborhood of a given order of a vertex includes all vertices which are closer to the vertex than the order. Ie. order 0 is always the vertex itself, order 1 is the vertex plus its immediate neighbors, order 2 is order 1 plus the immediate neighbors of the vertices in order 1, etc.

This function calculates the vertices within the neighborhood of the specified vertices.

Arguments:

graph:

The input graph.

res:

An initialized pointer vector. Note that the objects (pointers) in the vector will *not* be freed, but the pointer vector will be resized as needed. The result of the calculation will be stored here in `vector_t` objects.

vids:

The vertices for which the calculation is performed.

order:

Integer giving the order of the neighborhood.

mode:

Specifies how to use the direction of the edges if a directed graph is analyzed. For `IGRAPH_OUT` only the outgoing edges are followed, so all vertices reachable from the source vertex in at most `order` steps are included. For `IGRAPH_IN` all vertices from which the source vertex is reachable in at most `order` steps are included. `IGRAPH_ALL` ignores the direction of the edges. This argument is ignored for undirected graphs.

Returns:

Error code.

See also:

`igraph_neighborhood_size()` to calculate the size of the neighborhood, `igraph_neighborhood_graphs()` for creating graphs from the neighborhoods.

Time complexity: O(n*d*o), n is the number of vertices for which the calculation is performed, d is the average degree, o is the order.

13.3.3. `igraph_neighborhood_graphs` — Create graphs from the neighborhood(s) of some vertex/vertices.

```
int igraph_neighborhood_graphs(const igraph_t *graph, igraph_vector_ptr_t *res,
```

```
igraph_vs_t vids, igraph_integer_t order,
igraph_neimode_t mode);
```

The neighborhood of a given order of a vertex includes all vertices which are closer to the vertex than the order. Ie. order 0 is always the vertex itself, order 1 is the vertex plus its immediate neighbors, order 2 is order 1 plus the immediate neighbors of the vertices in order 1, etc.

This function finds every vertex in the neighborhood of a given parameter vertex and creates a graph from these vertices.

The first version of this function was written by Vincent Matossian, thanks Vincent.

Arguments:

`graph`:

The input graph.

`res`:

Pointer to a pointer vector, the result will be stored here, ie. `res` will contain pointers to `igraph_t` objects. It will be resized if needed but note that the objects in the pointer vector will not be freed.

`vids`:

The vertices for which the calculation is performed.

`order`:

Integer giving the order of the neighborhood.

`mode`:

Specifies how to use the direction of the edges if a directed graph is analyzed. For `IGRAPH_OUT` only the outgoing edges are followed, so all vertices reachable from the source vertex in at most `order` steps are counted. For `IGRAPH_IN` all vertices from which the source vertex is reachable in at most `order` steps are counted. `IGRAPH_ALL` ignores the direction of the edges. This argument is ignored for undirected graphs.

Returns:

Error code.

See also:

`igraph_neighborhood_size()` for calculating the neighborhood sizes only,
`igraph_neighborhood()` for calculating the neighborhoods (but not creating graphs).

Time complexity: O(n*(|V|+|E|)), where n is the number vertices for which the calculation is performed, |V| and |E| are the number of vertices and edges in the original input graph.

13.4. Graph Components

13.4.1. `igraph_subcomponent` — The vertices in the same component as a given vertex.

```
int igraph_subcomponent(const igraph_t *graph, igraph_vector_t *res, igraph_real_t vertex,
   igraph_neimode_t mode);
```

Arguments:

graph:

The graph object.

res:

The result, vector with the ids of the vertices in the same component.

vertex:

The id of the vertex of which the component is searched.

mode:

Type of the component for directed graphs, possible values:

`IGRAPH_OUT`

the set of vertices reachable *from* the *vertex*,

`IGRAPH_IN`

the set of vertices from which the *vertex* is reachable.

`IGRAPH_ALL`

the graph is considered as an undirected graph. Note that this is *not* the same as the union of the previous two.

Returns:

Error code:

`IGRAPH_ENOMEM`

not enough memory for temporary data.

`IGRAPH_EINVVID`

vertex is an invalid vertex id

`IGRAPH_EINVMODE`

invalid mode argument passed.

Time complexity: O(|V|+|E|), |V| and |E| are the number of vertices and edges in the graph.

See also:

`igraph_subgraph()` if you want a graph object consisting only a given set of vertices and the edges between them.

13.4.2. `igraph_induced_subgraph` — Creates a subgraph induced by the specified vertices.

```
int igraph_induced_subgraph(const igraph_t *graph, igraph_t *res,
      const igraph_vs_t vids, igraph_subgraph_implementation_t impl);
```

This function collects the specified vertices and all edges between them to a new graph. As the vertex ids in a graph always start with zero, this function very likely needs to reassign ids to the vertices.

Arguments:

graph:

The graph object.

`res`:

The subgraph, another graph object will be stored here, do *not* initialize this object before calling this function, and call `igraph_destroy()` on it if you don't need it any more.

`vids`:

A vertex selector describing which vertices to keep.

`impl`:

This parameter selects which implementation should we use when constructing the new graph. Basically there are two possibilities: `IGRAPH_SUBGRAPH_COPY_AND_DELETE` copies the existing graph and deletes the vertices that are not needed in the new graph, while `IGRAPH_SUBGRAPH_CREATE_FROM_SCRATCH` constructs the new graph from scratch without copying the old one. The latter is more efficient if you are extracting a relatively small subpart of a very large graph, while the former is better if you want to extract a subgraph whose size is comparable to the size of the whole graph. There is a third possibility: `IGRAPH_SUBGRAPH_AUTO` will select one of the two methods automatically based on the ratio of the number of vertices in the new and the old graph.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data. `IGRAPH_EINVVID`, invalid vertex id in *`vids`*.

Time complexity: O(|V|+|E|), |V| and |E| are the number of vertices and edges in the original graph.

See also:

`igraph_delete_vertices()` to delete the specified set of vertices from a graph, the opposite of this function.

13.4.3. `igraph_subgraph_edges` — Creates a subgraph with the specified edges and their endpoints.

```
int igraph_subgraph_edges(const igraph_t *graph, igraph_t *res,
      const igraph_es_t eids, igraph_bool_t delete_vertices);
```

This function collects the specified edges and their endpoints to a new graph. As the vertex ids in a graph always start with zero, this function very likely needs to reassign ids to the vertices.

Arguments:

graph:

The graph object.

res:

The subgraph, another graph object will be stored here, do *not* initialize this object before calling this function, and call `igraph_destroy()` on it if you don't need it any more.

eids:

An edge selector describing which edges to keep.

delete_vertices:

Whether to delete the vertices not incident on any of the specified edges as well. If `FALSE`, the number of vertices in the result graph will always be equal to the number of vertices in the input graph.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data. `IGRAPH_EINVEID`, invalid edge id in *eids*.

Time complexity: O(|V|+|E|), |V| and |E| are the number of vertices and edges in the original graph.

See also:

`igraph_delete_edges()` to delete the specified set of edges from a graph, the opposite of this function.

13.4.4. `igraph_subgraph` — Creates a subgraph induced by the specified vertices.

```
int igraph_subgraph(const igraph_t *graph, igraph_t *res,
      const igraph_vs_t vids);
```

This function is an alias to `igraph_induced_subgraph()`, it is left here to ensure API compatibility with igraph versions prior to 0.6.

This function collects the specified vertices and all edges between them to a new graph. As the vertex ids in a graph always start with zero, this function very likely needs to reassign ids to the vertices.

Arguments:

graph:

The graph object.

res:

The subgraph, another graph object will be stored here, do *not* initialize this object before calling this function, and call `igraph_destroy()` on it if you don't need it any more.

vids:

A vertex selector describing which vertices to keep.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data. `IGRAPH_EINVVID`, invalid vertex id in *vids*.

Time complexity: O(|V|+|E|), |V| and |E| are the number of vertices and edges in the original graph.

See also:

`igraph_delete_vertices()` to delete the specified set of vertices from a graph, the opposite of this function.

13.4.5. `igraph_clusters` — Calculates the (weakly or strongly) connected components in a graph.

```
int igraph_clusters(const igraph_t *graph, igraph_vector_t *membership,
      igraph_vector_t *csize, igraph_integer_t *no,
      igraph_connectedness_t mode);
```

Arguments:

graph:

The graph object to analyze.

membership:

First half of the result will be stored here. For every vertex the id of its component is given. The vector has to be preinitialized and will be resized. Alternatively this argument can be `NULL`, in which case it is ignored.

csize:

The second half of the result. For every component it gives its size, the order is defined by the component ids. The vector has to be preinitialized and will be resized. Alternatively this argument can be `NULL`, in which case it is ignored.

no:

Pointer to an integer, if not `NULL` then the number of clusters will be stored here.

mode:

For directed graph this specifies whether to calculate weakly or strongly connected components. Possible values: `IGRAPH_WEAK`, `IGRAPH_STRONG`. This argument is ignored for undirected graphs.

Returns:

Error code: `IGRAPH_EINVAL`: invalid mode argument.

Time complexity: O(|V|+|E|), |V| and |E| are the number of vertices and edges in the graph.

13.4.6. `igraph_is_connected` — Decides whether the graph is (weakly or strongly) connected.

```
int igraph_is_connected(const igraph_t *graph, igraph_bool_t *res,
   igraph_connectedness_t mode);
```

A graph with zero vertices (i.e. the null graph) is connected by definition.

Arguments:

graph:

The graph object to analyze.

res:

Pointer to a logical variable, the result will be stored here.

mode:

For a directed graph this specifies whether to calculate weak or strong connectedness. Possible values: `IGRAPH_WEAK`, `IGRAPH_STRONG`. This argument is ignored for undirected graphs.

Returns:

Error code: `IGRAPH_EINVAL`: invalid mode argument.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph.

13.4.7. `igraph_decompose` — Decompose a graph into connected components.

```
int igraph_decompose(const igraph_t *graph, igraph_vector_ptr_t *components,
       igraph_connectedness_t mode,
       long int maxcompno, long int minelements);
```

Create separate graph for each component of a graph. Note that the vertex ids in the new graphs will be different than in the original graph. (Except if there is only one component in the original graph.)

Arguments:

graph:

The original graph.

components:

This pointer vector will contain pointers to the subcomponent graphs. It should be initialized before calling this function and will be resized to hold the graphs. Don't forget to call `igraph_destroy()` and free() on the elements of this pointer vector to free unneeded memory. Alternatively, you can simply call `igraph_decompose_destroy()` that does this for you.

mode:

Either `IGRAPH_WEAK` or `IGRAPH_STRONG` for weakly and strongly connected components respectively. Right now only the former is implemented.

maxcompno:

The maximum number of components to return. The first *maxcompno* components will be returned (which hold at least *minelements* vertices, see the next parameter), the others will be ignored. Supply -1 here if you don't want to limit the number of components.

minelements:

The minimum number of vertices a component should contain in order to place it in the *components* vector. Eg. supply 2 here to ignore isolated vertices.

Returns:

Error code, `IGRAPH_ENOMEM` if there is not enough memory to perform the operation.

Added in version 0.2.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

Example 13-11. File examples/simple/igraph_decompose.c

13.4.8. `igraph_decompose_destroy` — Free the memory allocated by `igraph_decompose()`.

```
void igraph_decompose_destroy(igraph_vector_ptr_t *complist);
```

Arguments:

complist:

The list of graph components, as returned by `igraph_decompose()`.

Time complexity: O(c), c is the number of components.

13.4.9. `igraph_biconnected_components` — Calculate biconnected components

```
int igraph_biconnected_components(const igraph_t *graph,
      igraph_integer_t *no,
      igraph_vector_ptr_t *tree_edges,
      igraph_vector_ptr_t *component_edges,
      igraph_vector_ptr_t *components,
      igraph_vector_t *articulation_points);
```

A graph is biconnected if the removal of any single vertex (and its incident edges) does not disconnect it.

A biconnected component of a graph is a maximal biconnected subgraph of it. The biconnected components of a graph can be given by the partition of its edges: every edge is a member of exactly one biconnected component. Note that this is not true for vertices: the same vertex can be part of many biconnected components.

Arguments:

graph:

The input graph.

no:

The number of biconnected components will be stored here.

tree_edges:

If not a NULL pointer, then the found components are stored here, in a list of vectors. Every vector in the list is a biconnected component, represented by its edges. More precisely, a spanning tree of the biconnected component is returned. Note you'll have to destroy each vector first by calling `igraph_vector_destroy()` and then `free()` on it, plus you need to call `igraph_vector_ptr_destroy()` on the list to regain all allocated memory.

component_edges:

If not a NULL pointer, then the edges of the biconnected components are stored here, in the same form as for `tree_edges`.

components:

If not a NULL pointer, then the vertices of the biconnected components are stored here, in the same format as for the previous two arguments.

articulation_points:

If not a NULL pointer, then the articulation points of the graph are stored in this vector. A vertex is an articulation point if its removal increases the number of (weakly) connected components in the graph.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges, but only if you do not calculate `components` and `component_edges`. If you calculate `components`, then it is quadratic in the number of vertices. If you calculate `component_edges` as well, then it is cubic in the number of vertices.

See also:

`igraph_articulation_points()`, `igraph_clusters()`.

Example 13-12. File examples/simple/igraph_biconnected_components.c

13.4.10. `igraph_articulation_points` — Find the articulation points in a graph.

```
int igraph_articulation_points(const igraph_t *graph,
          igraph_vector_t *res);
```

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

Arguments:

graph:

The input graph.

res:

Pointer to an initialized vector, the articulation points will be stored here.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

See also:

`igraph_biconnected_components()`, `igraph_clusters()`

13.5. Degree Sequences

13.5.1. `igraph_is_degree_sequence` — Determines whether a degree sequence is valid.

```
int igraph_is_degree_sequence(const igraph_vector_t *out_degrees,
    const igraph_vector_t *in_degrees, igraph_bool_t *res);
```

A sequence of n integers is a valid degree sequence if there exists some graph where the degree of the i-th vertex is equal to the i-th element of the sequence. Note that the graph may contain multiple or loop edges; if you are interested in whether the degrees of some *simple* graph may realize the given sequence, use `igraph_is_graphical_degree_sequence`.

In particular, the function checks whether all the degrees are non-negative. For undirected graphs, it also checks whether the sum of degrees is even. For directed graphs, the function checks whether the lengths of the two degree vectors are equal and whether their sums are also equal. These are known sufficient and necessary conditions for a degree sequence to be valid.

Arguments:

out_degrees:

an integer vector specifying the degree sequence for undirected graphs or the out-degree sequence for directed graphs.

in_degrees:

an integer vector specifying the in-degrees of the vertices for directed graphs. For undirected graphs, this must be null.

res:

pointer to a boolean variable, the result will be stored here

Returns:

Error code.

Time complexity: O(n), where n is the length of the degree sequence.

13.5.2. `igraph_is_graphical_degree_sequence` — Determines whether a sequence of integers can be a degree sequence of some

```
int igraph_is_graphical_degree_sequence(const igraph_vector_t *out_degrees,
    const igraph_vector_t *in_degrees, igraph_bool_t *res);
```

simple graph.

References:

Hakimi SL: On the realizability of a set of integers as degrees of the vertices of a simple graph. J SIAM Appl Math 10:496-506, 1962.

PL Erdos, I Miklos and Z Toroczkai: A simple Havel-Hakimi type algorithm to realize graphical degree sequences of directed graphs. The Electronic Journal of Combinatorics 17(1):R66, 2010.

Arguments:

out_degrees:

an integer vector specifying the degree sequence for undirected graphs or the out-degree sequence for directed graphs.

in_degrees:

an integer vector specifying the in-degrees of the vertices for directed graphs. For undirected graphs, this must be null.

res:

pointer to a boolean variable, the result will be stored here

Returns:

Error code.

Time complexity: O(n^2 log n) where n is the length of the degree sequence.

13.6. Centrality Measures

13.6.1. `igraph_closeness` — Closeness centrality calculations for some vertices.

```
int igraph_closeness(const igraph_t *graph, igraph_vector_t *res,
                     const igraph_vs_t vids, igraph_neimode_t mode,
       const igraph_vector_t *weights);
```

The closeness centrality of a vertex measures how easily other vertices can be reached from it (or the other way: how easily it can be reached from the other vertices). It is defined as the number of the number of vertices minus one divided by the sum of the lengths of all geodesics from/to the given vertex.

If the graph is not connected, and there is no path between two vertices, the number of vertices is used instead the length of the geodesic. This is always longer than the longest possible geodesic.

Arguments:

graph:

The graph object.

res:

The result of the computation, a vector containing the closeness centrality scores for the given vertices.

vids:

Vector giving the vertices for which the closeness centrality scores will be computed.

mode:

The type of shortest paths to be used for the calculation in directed graphs. Possible values:

`IGRAPH_OUT`

the lengths of the outgoing paths are calculated.

`IGRAPH_IN`

the lengths of the incoming paths are calculated.

`IGRAPH_ALL`

the directed graph is considered as an undirected one for the computation.

weights:

An optional vector containing edge weights for weighted closeness. Supply a null pointer here for traditional, unweighted closeness.

Returns:

Error code:

`IGRAPH_ENOMEM`

not enough memory for temporary data.

`IGRAPH_EINVVID`

invalid vertex id passed.

`IGRAPH_EINVMODE`

invalid mode argument.

Time complexity: O(n|E|), n is the number of vertices for which the calculation is done and |E| is the number of edges in the graph.

See also:

Other centrality types: `igraph_degree()`, `igraph_betweenness()`. See `igraph_closeness_estimate()` to estimate closeness values.

13.6.2. `igraph_betweenness` — Betweenness centrality of some vertices.

```
int igraph_betweenness(const igraph_t *graph, igraph_vector_t *res,
          const igraph_vs_t vids, igraph_bool_t directed,
          const igraph_vector_t* weights, igraph_bool_t nobigint);
```

The betweenness centrality of a vertex is the number of geodesics going through it. If there are more than one geodesic between two vertices, the value of these geodesics are weighted by one over the number of geodesics.

Arguments:

graph:

The graph object.

res:

The result of the computation, a vector containing the betweenness scores for the specified vertices.

vids:

The vertices of which the betweenness centrality scores will be calculated.

directed:

Logical, if true directed paths will be considered for directed graphs. It is ignored for undirected graphs.

weights:

An optional vector containing edge weights for calculating weighted betweenness. Supply a null pointer here for unweighted betweenness.

nobigint:

Logical, if true, then we don't use big integers for the calculation, setting this to 1 (=true) should work for most graphs. It is currently ignored for weighted graphs.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data. `IGRAPH_EINVVID`, invalid vertex id passed in *`vids`*.

Time complexity: O(|V||E|), |V| and |E| are the number of vertices and edges in the graph. Note that the time complexity is independent of the number of vertices for which the score is calculated.

See also:

Other centrality types: `igraph_degree()`, `igraph_closeness()`. See `igraph_edge_betweenness()` for calculating the betweenness score of the edges in a graph. See `igraph_betweenness_estimate()` to estimate the betweenness score of the vertices in a graph.

Example 13-13. File examples/simple/igraph_betweenness.c

13.6.3. `igraph_edge_betweenness` — Betweenness centrality of the edges.

```
int igraph_edge_betweenness(const igraph_t *graph, igraph_vector_t *result,
                            igraph_bool_t directed,
        const igraph_vector_t *weights);
```

The betweenness centrality of an edge is the number of geodesics going through it. If there are more than one geodesics between two vertices, the value of these geodesics are weighted by one over the number of geodesics.

Arguments:

`graph`:

The graph object.

result:

The result of the computation, vector containing the betweenness scores for the edges.

directed:

Logical, if true directed paths will be considered for directed graphs. It is ignored for undirected graphs.

weights:

An optional weight vector for weighted edge betweenness. Supply a null pointer here for the unweighted version.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data.

Time complexity: O(|V||E|), |V| and |E| are the number of vertices and edges in the graph.

See also:

Other centrality types: `igraph_degree()`, `igraph_closeness()`. See `igraph_edge_betweenness()` for calculating the betweenness score of the edges in a graph. See `igraph_edge_betweenness_estimate()` to estimate the betweenness score of the edges in a graph.

Example 13-14. File examples/simple/igraph_edge_betweenness.c

13.6.4. `igraph_pagerank` — Calculates the Google PageRank for the specified vertices.

```
int igraph_pagerank(const igraph_t *graph, igraph_vector_t *vector,
      igraph_real_t *value, const igraph_vs_t vids,
      igraph_bool_t directed, igraph_real_t damping,
      const igraph_vector_t *weights,
      igraph_arpack_options_t *options);
```

This is the new PageRank implementation, based on the ARPACK library. The old, power-method based implementation can be used as well, it is kept under the name `igraph_pagerank_old()`.

Please note that the PageRank of a given vertex depends on the PageRank of all other vertices, so even if you want to calculate the PageRank for only some of the vertices, all of them must be calculated. Requesting the PageRank for only some of the vertices does not result in any performance increase at all.

For the explanation of the PageRank algorithm, see the following webpage: http://www-db.stanford.edu/~backrub/google.html, or the following reference:

Sergey Brin and Larry Page: The Anatomy of a Large-Scale Hypertextual Web Search Engine. Proceedings of the 7th World-Wide Web Conference, Brisbane, Australia, April 1998.

Arguments:

`graph`:

The graph object.

`vector`:

Pointer to an initialized vector, the result is stored here. It is resized as needed.

`value`:

Pointer to a real variable, the eigenvalue corresponding to the PageRank vector is stored here. It should be always exactly one.

`vids`:

The vertex ids for which the PageRank is returned.

`directed`:

Boolean, whether to consider the directedness of the edges. This is ignored for undirected graphs.

`damping`:

The damping factor ("d" in the original paper)

`weights`:

Optional edge weights, it is either a null pointer, then the edges are not weighted, or a vector of the same length as the number of edges.

`options`:

Options to ARPACK. See `igraph_arpack_options_t` for details. Note that the function overwrites the `n` (number of vertices), `nev` (1), `ncv` (3) and `which` (LM) parameters and it always starts the calculation from a non-random vector calculated based on the degree of the vertices.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data. `IGRAPH_EINVVID`, invalid vertex id in *`vids`*.

Time complexity: depends on the input graph, usually it is O(|E|), the number of edges.

See also:

`igraph_pagerank_old()` for the old implementation, `igraph_personalized_pagerank()` and `igraph_personalized_pagerank_vs()` for the personalized PageRank measure, `igraph_arpack_rssolve()` and `igraph_arpack_rnsolve()` for the underlying machinery.

Example 13-15. File examples/simple/igraph_pagerank.c

13.6.5. `igraph_pagerank_old` — Calculates the Google PageRank for the specified vertices.

```
int igraph_pagerank_old(const igraph_t *graph, igraph_vector_t *res,
   const igraph_vs_t vids, igraph_bool_t directed,
   igraph_integer_t niter, igraph_real_t eps,
   igraph_real_t damping, igraph_bool_t old);
```

This is an old implementation, it is provided for compatibility with igraph versions earlier than 0.5. Please use the new implementation `igraph_pagerank()` in new projects.

Please note that the PageRank of a given vertex depends on the PageRank of all other vertices, so even if you want to calculate the PageRank for only some of the vertices, all of them must be calculated. Requesting the PageRank for only some of the vertices does not result in any performance increase at all.

Since the calculation is an iterative process, the algorithm is stopped after a given count of iterations or if the PageRank value differences between iterations are less than a predefined value.

For the explanation of the PageRank algorithm, see the following webpage: http://www-db.stanford.edu/~backrub/google.html, or the following reference:

Sergey Brin and Larry Page: The Anatomy of a Large-Scale Hypertextual Web Search Engine. Proceedings of the 7th World-Wide Web Conference, Brisbane, Australia, April 1998.

Arguments:

`graph`:

The graph object.

`res`:

The result vector containing the PageRank values for the given nodes.

`vids`:

Vector with the vertex ids

`directed`:

Logical, if true directed paths will be considered for directed graphs. It is ignored for undirected graphs.

`niter`:

The maximum number of iterations to perform

`eps`:

The algorithm will consider the calculation as complete if the difference of PageRank values between iterations change less than this value for every node

`damping`:

The damping factor ("d" in the original paper)

`old`:

Boolean, whether to use the pre-igraph 0.5 way to calculate page rank. Not recommended for new applications, only included for compatibility. If this is non-zero then the damping factor is not divided by the number of vertices before adding it to the weighted page rank scores to calculate the new scores. I.e. the formula in the original PageRank paper is used. Furthermore, if this is non-zero then the PageRank vector is renormalized after each iteration.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data. `IGRAPH_EINVVID`, invalid vertex id in *`vids`*.

Time complexity: O(|V|+|E|) per iteration. A handful iterations should be enough. Note that if the old-style dumping is used then the iteration might not converge at all.

See also:

`igraph_pagerank()` for the new implementation.

13.6.6. `igraph_personalized_pagerank` — Calculates the personalized Google PageRank for the specified vertices.

```
int igraph_personalized_pagerank(const igraph_t *graph, igraph_vector_t *vector,
      igraph_real_t *value, const igraph_vs_t vids,
      igraph_bool_t directed, igraph_real_t damping,
      igraph_vector_t *reset,
      const igraph_vector_t *weights,
      igraph_arpack_options_t *options);
```

The personalized PageRank is similar to the original PageRank measure, but the random walk is reset in every step with probability 1-damping to a non-uniform distribution (instead of the uniform distribution in the original PageRank measure.

Please note that the personalized PageRank of a given vertex depends on the personalized PageRank of all other vertices, so even if you want to calculate the personalized PageRank for only some of the vertices, all of them must be calculated. Requesting the personalized PageRank for only some of the vertices does not result in any performance increase at all.

Arguments:

`graph`:

The graph object.

vector:

Pointer to an initialized vector, the result is stored here. It is resized as needed.

value:

Pointer to a real variable, the eigenvalue corresponding to the PageRank vector is stored here. It should be always exactly one.

vids:

The vertex ids for which the PageRank is returned.

directed:

Boolean, whether to consider the directedness of the edges. This is ignored for undirected graphs.

damping:

The damping factor ("d" in the original paper)

reset:

The probability distribution over the vertices used when resetting the random walk. It is either a null pointer (denoting a uniform choice that results in the original PageRank measure) or a vector of the same length as the number of vertices.

weights:

Optional edge weights, it is either a null pointer, then the edges are not weighted, or a vector of the same length as the number of edges.

options:

Options to ARPACK. See `igraph_arpack_options_t` for details. Note that the function overwrites the `n` (number of vertices), `nev` (1), `ncv` (3) and `which` (LM) parameters and it always starts the calculation from a non-random vector calculated based on the degree of the vertices.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data. `IGRAPH_EINVVID`, invalid vertex id in *vids* or an invalid reset vector in *reset*.

Time complexity: depends on the input graph, usually it is O(|E|), the number of edges.

See also:

`igraph_pagerank()` for the non-personalized implementation, `igraph_arpack_rssolve()` and `igraph_arpack_rnsolve()` for the underlying machinery.

13.6.7. `igraph_personalized_pagerank_vs` — Calculates the personalized Google PageRank for the specified vertices.

```
int igraph_personalized_pagerank_vs(const igraph_t *graph, igraph_vector_t *vector,
      igraph_real_t *value, const igraph_vs_t vids,
      igraph_bool_t directed, igraph_real_t damping,
      igraph_vs_t reset_vids,
      const igraph_vector_t *weights,
      igraph_arpack_options_t *options);
```

The personalized PageRank is similar to the original PageRank measure, but the random walk is reset in every step with probability 1-damping to a non-uniform distribution (instead of the uniform distribution in the original PageRank measure.

This simplified interface takes a vertex sequence and resets the random walk to one of the vertices in the specified vertex sequence, chosen uniformly. A typical application of personalized PageRank is when the random walk is reset to the same vertex every time - this can easily be achieved using `igraph_vss_1()` which generates a vertex sequence containing only a single vertex.

Please note that the personalized PageRank of a given vertex depends on the personalized PageRank of all other vertices, so even if you want to calculate the personalized PageRank for only some of the vertices, all of them must be calculated. Requesting the personalized PageRank for only some of the vertices does not result in any performance increase at all.

Arguments:

`graph`:

The graph object.

`vector`:

Pointer to an initialized vector, the result is stored here. It is resized as needed.

`value`:

Pointer to a real variable, the eigenvalue corresponding to the PageRank vector is stored here. It should be always exactly one.

`vids`:

The vertex ids for which the PageRank is returned.

`directed`:

Boolean, whether to consider the directedness of the edges. This is ignored for undirected graphs.

damping:

The damping factor ("d" in the original paper)

reset_vids:

IDs of the vertices used when resetting the random walk.

weights:

Optional edge weights, it is either a null pointer, then the edges are not weighted, or a vector of the same length as the number of edges.

options:

Options to ARPACK. See `igraph_arpack_options_t` for details. Note that the function overwrites the `n` (number of vertices), `nev` (1), `ncv` (3) and `which` (LM) parameters and it always starts the calculation from a non-random vector calculated based on the degree of the vertices.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data. `IGRAPH_EINVVID`, invalid vertex id in *vids* or an empty reset vertex sequence in *vids_reset*.

Time complexity: depends on the input graph, usually it is O(|E|), the number of edges.

See also:

`igraph_pagerank()` for the non-personalized implementation, `igraph_arpack_rssolve()` and `igraph_arpack_rnsolve()` for the underlying machinery.

13.6.8. `igraph_constraint` — Burt's constraint scores.

```
int igraph_constraint(const igraph_t *graph, igraph_vector_t *res,
        igraph_vs_t vids, const igraph_vector_t *weights);
```

This function calculates Burt's constraint scores for the given vertices, also known as structural holes.

Burt's constraint is higher if ego has less, or mutually stronger related (i.e. more redundant) contacts. Burt's measure of constraint, C[i], of vertex i's ego network V[i], is defined for directed and valued graphs,

C[i] = sum(sum((p[i,q] p[q,j])^2, q in V[i], q != i,j), j in V[], j != i)

for a graph of order (ie. number of vertices) N, where proportional tie strengths are defined as

p[i,j]=(a[i,j]+a[j,i]) / sum(a[i,k]+a[k,i], k in V[i], k != i),

a[i,j] are elements of A and the latter being the graph adjacency matrix. For isolated vertices, constraint is undefined.

Burt, R.S. (2004). Structural holes and good ideas. American Journal of Sociology 110, 349-399.

The first R version of this function was contributed by Jeroen Bruggeman.

Arguments:

`graph`:

A graph object.

`res`:

Pointer to an initialized vector, the result will be stored here. The vector will be resized to have the appropriate size for holding the result.

`vids`:

Vertex selector containing the vertices for which the constraint should be calculated.

`weights`:

Vector giving the weights of the edges. If it is `NULL` then each edge is supposed to have the same weight.

Returns:

Error code.

Time complexity: O(|V|+E|+n*d^2), n is the number of vertices for which the constraint is calculated and d is the average degree, |V| is the number of vertices, |E| the number of edges in the graph. If the weights argument is `NULL` then the time complexity is O(|V|+n*d^2).

13.6.9. `igraph_maxdegree` — Calculate the maximum degree in a graph (or set of vertices).

```
int igraph_maxdegree(const igraph_t *graph, igraph_integer_t *res,
       igraph_vs_t vids, igraph_neimode_t mode,
       igraph_bool_t loops);
```

The largest in-, out- or total degree of the specified vertices is calculated.

Arguments:

graph:

The input graph.

res:

Pointer to an integer (`igraph_integer_t`), the result will be stored here.

vids:

Vector giving the vertex IDs for which the maximum degree will be calculated.

mode:

Defines the type of the degree. `IGRAPH_OUT`, out-degree, `IGRAPH_IN`, in-degree, `IGRAPH_ALL`, total degree (sum of the in- and out-degree). This parameter is ignored for undirected graphs.

loops:

Boolean, gives whether the self-loops should be counted.

Returns:

Error code: `IGRAPH_EINVVID`: invalid vertex id. `IGRAPH_EINVMODE`: invalid mode argument.

Time complexity: O(v) if loops is TRUE, and O(v*d) otherwise. v is the number vertices for which the degree will be calculated, and d is their (average) degree.

13.6.10. `igraph_strength` — Strength of the vertices, weighted vertex degree in other words.

```
int igraph_strength(const igraph_t *graph, igraph_vector_t *res,
      const igraph_vs_t vids, igraph_neimode_t mode,
      igraph_bool_t loops, const igraph_vector_t *weights);
```

In a weighted network the strength of a vertex is the sum of the weights of all incident edges. In a non-weighted network this is exactly the vertex degree.

Arguments:

graph:

The input graph.

res:

Pointer to an initialized vector, the result is stored here. It will be resized as needed.

vids:

The vertices for which the calculation is performed.

mode:

Gives whether to count only outgoing (`IGRAPH_OUT`), incoming (`IGRAPH_IN`) edges or both (`IGRAPH_ALL`).

loops:

A logical scalar, whether to count loop edges as well.

weights:

A vector giving the edge weights. If this is a NULL pointer, then `igraph_degree()` is called to perform the calculation.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number vertices and edges.

See also:

`igraph_degree()` for the traditional, non-weighted version.

13.6.11. `igraph_eigenvector_centrality` — Eigenvector centrality of the vertices

```
int igraph_eigenvector_centrality(const igraph_t *graph,
      igraph_vector_t *vector,
      igraph_real_t *value,
      igraph_bool_t directed, igraph_bool_t scale,
      const igraph_vector_t *weights,
      igraph_arpack_options_t *options);
```

Eigenvector centrality is a measure of the importance of a node in a network. It assigns relative scores to all nodes in the network based on the principle that connections to high-scoring nodes contribute more to the score of the node in question than equal connections to low-scoring nodes. In practice, this is determined by calculating the eigenvector corresponding to the largest positive eigenvalue of the adjacency matrix. The centrality scores returned by igraph are always normalized such that the largest eigenvector centrality score is one (with one exception, see below).

Since the eigenvector centrality scores of nodes in different components do not affect each other, it may be beneficial for large graphs to decompose it first into weakly connected components and calculate the centrality scores individually for each component.

Also note that the adjacency matrix of a directed acyclic graph or the adjacency matrix of an empty graph does not possess positive eigenvalues, therefore the eigenvector centrality is not defined for these graphs. igraph will return an eigenvalue of zero in such cases. The eigenvector centralities will all be equal for an empty graph and will all be zeros for a directed acyclic graph. Such pathological cases can be detected by asking igraph to calculate the eigenvalue as well (using the *`value`* parameter, see below) and checking whether the eigenvalue is very close to zero.

Arguments:

`graph`:

The input graph. It might be directed.

`vector`:

Pointer to an initialized vector, it will be resized as needed. The result of the computation is stored here. It can be a null pointer, then it is ignored.

value:

If not a null pointer, then the eigenvalue corresponding to the found eigenvector is stored here.

directed:

Boolean scalar, whether to consider edge directions in a directed graph. It is ignored for undirected graphs.

scale:

If not zero then the result will be scaled such that the absolute value of the maximum centrality is one.

weights:

A null pointer (=no edge weights), or a vector giving the weights of the edges. The algorithm might result complex numbers is some weights are negative. In this case only the real part is reported.

options:

Options to ARPACK. See `igraph_arpack_options_t` for details. Note that the function overwrites the `n` (number of vertices) parameter and it always starts the calculation from a non-random vector calculated based on the degree of the vertices.

Returns:

Error code.

Time complexity: depends on the input graph, usually it is O(|V|+|E|).

See also:

`igraph_pagerank` and `igraph_personalized_pagerank` for modifications of eigenvector centrality.

Example 13-16. File examples/simple/eigenvector_centrality.c

13.6.12. `igraph_hub_score` — Kleinberg's hub scores

```
int igraph_hub_score(const igraph_t *graph, igraph_vector_t *vector,
       igraph_real_t *value, igraph_bool_t scale,
    const igraph_vector_t *weights,
       igraph_arpack_options_t *options);
```

The hub scores of the vertices are defined as the principal eigenvector of `A*A^T`, where `A` is the adjacency matrix of the graph, `A^T` is its transposed.

See the following reference on the meaning of this score: J. Kleinberg. Authoritative sources in a hyperlinked environment. *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms,* 1998. Extended version in *Journal of the ACM* 46(1999). Also appears as IBM Research Report RJ 10076, May 1997.

Arguments:

`graph`:

The input graph. Can be directed and undirected.

`vector`:

Pointer to an initialized vector, the result is stored here. If a null pointer then it is ignored.

`value`:

If not a null pointer then the eigenvalue corresponding to the calculated eigenvector is stored here.

`scale`:

If not zero then the result will be scaled such that the absolute value of the maximum centrality is one.

`weights`:

A null pointer (=no edge weights), or a vector giving the weights of the edges.

`options`:

Options to ARPACK. See `igraph_arpack_options_t` for details. Note that the function overwrites the `n` (number of vertices) parameter and it always starts the calculation from a non-random vector calculated based on the degree of the vertices.

Returns:

Error code.

Time complexity: depends on the input graph, usually it is O(|V|), the number of vertices.

See also:

> `igraph_authority_score()` for the companion measure, `igraph_pagerank()`, `igraph_personalized_pagerank()`, `igraph_eigenvector_centrality()` for similar measures.

13.6.13. `igraph_authority_score` — Kleinerg's authority scores

```
int igraph_authority_score(const igraph_t *graph, igraph_vector_t *vector,
      igraph_real_t *value, igraph_bool_t scale,
      const igraph_vector_t *weights,
      igraph_arpack_options_t *options);
```

The authority scores of the vertices are defined as the principal eigenvector of `A^T*A`, where `A` is the adjacency matrix of the graph, `A^T` is its transposed.

See the following reference on the meaning of this score: J. Kleinberg. Authoritative sources in a hyperlinked environment. *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms,* 1998. Extended version in *Journal of the ACM* 46(1999). Also appears as IBM Research Report RJ 10076, May 1997.

Arguments:

`graph`:

> The input graph. Can be directed and undirected.

`vector`:

> Pointer to an initialized vector, the result is stored here. If a null pointer then it is ignored.

`value`:

> If not a null pointer then the eigenvalue corresponding to the calculated eigenvector is stored here.

`scale`:

> If not zero then the result will be scaled such that the absolute value of the maximum centrality is one.

`weights`:

> A null pointer (=no edge weights), or a vector giving the weights of the edges.

options:

Options to ARPACK. See `igraph_arpack_options_t` for details. Note that the function overwrites the `n` (number of vertices) parameter and it always starts the calculation from a non-random vector calculated based on the degree of the vertices.

Returns:

Error code.

Time complexity: depends on the input graph, usually it is O(|V|), the number of vertices.

See also:

`igraph_hub_score()` for the companion measure, `igraph_pagerank()`, `igraph_personalized_pagerank()`, `igraph_eigenvector_centrality()` for similar measures.

13.7. Estimating Centrality Measures

13.7.1. `igraph_closeness_estimate` — Closeness centrality estimations for some vertices.

```
int igraph_closeness_estimate(const igraph_t *graph, igraph_vector_t *res,
                const igraph_vs_t vids, igraph_neimode_t mode,
                              igraph_real_t cutoff,
         const igraph_vector_t *weights);
```

The closeness centrality of a vertex measures how easily other vertices can be reached from it (or the other way: how easily it can be reached from the other vertices). It is defined as the number of the number of vertices minus one divided by the sum of the lengths of all geodesics from/to the given vertex. When estimating closeness centrality, igraph considers paths having a length less than or equal to a prescribed cutoff value.

If the graph is not connected, and there is no such path between two vertices, the number of vertices is used instead the length of the geodesic. This is always longer than the longest possible geodesic.

Since the estimation considers vertex pairs with a distance greater than the given value as disconnected, the resulting estimation will always be lower than the actual closeness centrality.

Arguments:

graph:

The graph object.

res:

The result of the computation, a vector containing the closeness centrality scores for the given vertices.

vids:

Vector giving the vertices for which the closeness centrality scores will be computed.

mode:

The type of shortest paths to be used for the calculation in directed graphs. Possible values:

`IGRAPH_OUT`

the lengths of the outgoing paths are calculated.

`IGRAPH_IN`

the lengths of the incoming paths are calculated.

`IGRAPH_ALL`

the directed graph is considered as an undirected one for the computation.

cutoff:

The maximal length of paths that will be considered. If zero or negative, the exact closeness will be calculated (no upper limit on path lengths).

weights:

An optional vector containing edge weights for weighted closeness. Supply a null pointer here for traditional, unweighted closeness.

Returns:

Error code:

`IGRAPH_ENOMEM`

not enough memory for temporary data.

`IGRAPH_EINVVID`

invalid vertex id passed.

`IGRAPH_EINVMODE`

invalid mode argument.

Time complexity: O(n|E|), n is the number of vertices for which the calculation is done and |E| is the number of edges in the graph.

See also:

Other centrality types: `igraph_degree()`, `igraph_betweenness()`.

13.7.2. `igraph_betweenness_estimate` — Estimated betweenness centrality of some vertices.

```
int igraph_betweenness_estimate(const igraph_t *graph, igraph_vector_t *res,
    const igraph_vs_t vids, igraph_bool_t directed,
    igraph_real_t cutoff,
    const igraph_vector_t *weights,
    igraph_bool_t nobigint);
```

The betweenness centrality of a vertex is the number of geodesics going through it. If there are more than one geodesic between two vertices, the value of these geodesics are weighted by one over the number of geodesics. When estimating betweenness centrality, igraph takes into consideration only those paths that are shorter than or equal to a prescribed length. Note that the estimated centrality will always be less than the real one.

Arguments:

`graph`:

The graph object.

res:

The result of the computation, a vector containing the estimated betweenness scores for the specified vertices.

vids:

The vertices of which the betweenness centrality scores will be estimated.

directed:

Logical, if true directed paths will be considered for directed graphs. It is ignored for undirected graphs.

cutoff:

The maximal length of paths that will be considered. If zero or negative, the exact betweenness will be calculated (no upper limit on path lengths).

weights:

An optional vector containing edge weights for calculating weighted betweenness. Supply a null pointer here for unweighted betweenness.

nobigint:

Logical, if true, then we don't use big integers for the calculation, setting this to 1 (=true) should work for most graphs. It is currently ignored for weighted graphs.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data. `IGRAPH_EINVVID`, invalid vertex id passed in *vids*.

Time complexity: O(|V||E|), |V| and |E| are the number of vertices and edges in the graph. Note that the time complexity is independent of the number of vertices for which the score is calculated.

See also:

Other centrality types: `igraph_degree()`, `igraph_closeness()`. See `igraph_edge_betweenness()` for calculating the betweenness score of the edges in a graph.

13.7.3. `igraph_edge_betweenness_estimate` — Estimated betweenness centrality of the edges.

```
int igraph_edge_betweenness_estimate(const igraph_t *graph, igraph_vector_t *result,
                                     igraph_bool_t directed, igraph_real_t cutoff,
         const igraph_vector_t *weights);
```

The betweenness centrality of an edge is the number of geodesics going through it. If there are more than one geodesics between two vertices, the value of these geodesics are weighted by one over the number of geodesics. When estimating betweenness centrality, igraph takes into consideration only those paths that are shorter than or equal to a prescribed length. Note that the estimated centrality will always be less than the real one.

Arguments:

graph:

The graph object.

result:

The result of the computation, vector containing the betweenness scores for the edges.

directed:

Logical, if true directed paths will be considered for directed graphs. It is ignored for undirected graphs.

cutoff:

The maximal length of paths that will be considered. If zero or negative, the exact betweenness will be calculated (no upper limit on path lengths).

weights:

An optional weight vector for weighted betweenness. Supply a null pointer here for unweighted betweenness.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data.

Time complexity: O(|V||E|), |V| and |E| are the number of vertices and edges in the graph.

See also:

Other centrality types: `igraph_degree()`, `igraph_closeness()`. See `igraph_betweenness()` for calculating the betweenness score of the vertices in a graph.

13.8. Centralization

13.8.1. `igraph_centralization` — Calculate the centralization score from the node level scores

```
igraph_real_t igraph_centralization(const igraph_vector_t *scores,
        igraph_real_t theoretical_max,
        igraph_bool_t normalized);
```

For a centrality score defined on the vertices of a graph, it is possible to define a graph level centralization index, by calculating the sum of the deviation from the maximum centrality score. Consequently, the higher the centralization index of the graph, the more centralized the structure is.

In order to make graphs of different sizes comparable, the centralization index is usually normalized to a number between zero and one, by dividing the (unnormalized) centralization score of the most centralized structure with the same number of vertices.

For most centrality indices the most centralized structure is the star graph, a single center connected to all other nodes in the network. There are some variation depending on whether the graph is directed or not, whether loop edges are allowed, etc.

This function simply calculates the graph level index, if the node level scores and the theoretical maximum are given. It is called by all the measure-specific centralization functions.

Arguments:

`scores`:

A vector containing the node-level centrality scores.

theoretical_max:

The graph level centrality score of the most centralized graph with the same number of vertices. Only used if `normalized` set to true.

normalized:

Boolean, whether to normalize the centralization by dividing the supplied theoretical maximum.

Returns:

The graph level index.

See also:

`igraph_centralization_degree()`, `igraph_centralization_betweenness()`, `igraph_centralization_closeness()`, and `igraph_centralization_eigenvector_centrality()` for specific centralization functions.

Time complexity: O(n), the length of the score vector.

Example 13-17. File examples/simple/centralization.c

13.8.2. `igraph_centralization_degree` — Calculate vertex degree and graph centralization

```
int igraph_centralization_degree(const igraph_t *graph, igraph_vector_t *res,
     igraph_neimode_t mode, igraph_bool_t loops,
     igraph_real_t *centralization,
     igraph_real_t *theoretical_max,
     igraph_bool_t normalized);
```

This function calculates the degree of the vertices by passing its arguments to `igraph_degree()`; and it calculates the graph level centralization index based on the results by calling `igraph_centralization()`.

Arguments:

graph:

The input graph.

res:

A vector if you need the node-level degree scores, or a null pointer otherwise.

mode:

Constant the specifies the type of degree for directed graphs. Possible values: `IGRAPH_IN`, `IGRAPH_OUT` and `IGRAPH_ALL`. This argument is ignored for undirected graphs.

loops:

Boolean, whether to consider loop edges when calculating the degree (and the centralization).

centralization:

Pointer to a real number, the centralization score is placed here.

theoretical_max:

Pointer to real number or a null pointer. If not a null pointer, then the theoretical maximum graph centrality score for a graph with the same number vertices is stored here.

normalized:

Boolean, whether to calculate a normalized centralization score. See `igraph_centralization()` for how the normalization is done.

Returns:

Error code.

See also:

`igraph_centralization()`, `igraph_degree()`.

Time complexity: the complexity of `igraph_degree()` plus O(n), the number of vertices queried, for calculating the centralization score.

13.8.3. `igraph_centralization_betweenness` — Calculate vertex betweenness and graph centralization

```
int igraph_centralization_betweenness(const igraph_t *graph,
          igraph_vector_t *res,
          igraph_bool_t directed,
          igraph_bool_t nobigint,
          igraph_real_t *centralization,
          igraph_real_t *theoretical_max,
          igraph_bool_t normalized);
```

This function calculates the betweenness centrality of the vertices by passing its arguments to `igraph_betweenness()`; and it calculates the graph level centralization index based on the results by calling `igraph_centralization()`.

Arguments:

`graph`:

The input graph.

`res`:

A vector if you need the node-level betweenness scores, or a null pointer otherwise.

`directed`:

Boolean, whether to consider directed paths when calculating betweenness.

`nobigint`:

Logical, if true, then we don’t use big integers for the calculation, setting this to zero (=false) should work for most graphs. It is currently ignored for weighted graphs.

`centralization`:

Pointer to a real number, the centralization score is placed here.

`theoretical_max`:

Pointer to real number or a null pointer. If not a null pointer, then the theoretical maximum graph centrality score for a graph with the same number vertices is stored here.

`normalized`:

Boolean, whether to calculate a normalized centralization score. See `igraph_centralization()` for how the normalization is done.

Returns:

Error code.

See also:

`igraph_centralization()`, `igraph_betweenness()`.

Time complexity: the complexity of `igraph_betweenness()` plus O(n), the number of vertices queried, for calculating the centralization score.

13.8.4. `igraph_centralization_closeness` — Calculate vertex closeness and graph centralization

```
int igraph_centralization_closeness(const igraph_t *graph,
        igraph_vector_t *res,
        igraph_neimode_t mode,
        igraph_real_t *centralization,
        igraph_real_t *theoretical_max,
        igraph_bool_t normalized);
```

This function calculates the closeness centrality of the vertices by passing its arguments to `igraph_closeness()`; and it calculates the graph level centralization index based on the results by calling `igraph_centralization()`.

Arguments:

`graph`:

The input graph.

`res`:

A vector if you need the node-level closeness scores, or a null pointer otherwise.

`mode`:

Constant the specifies the type of closeness for directed graphs. Possible values: `IGRAPH_IN`, `IGRAPH_OUT` and `IGRAPH_ALL`. This argument is ignored for undirected graphs. See `igraph_closeness()` argument with the same name for more.

`centralization`:

Pointer to a real number, the centralization score is placed here.

theoretical_max:

Pointer to real number or a null pointer. If not a null pointer, then the theoretical maximum graph centrality score for a graph with the same number vertices is stored here.

normalized:

Boolean, whether to calculate a normalized centralization score. See `igraph_centralization()` for how the normalization is done.

Returns:

Error code.

See also:

`igraph_centralization()`, `igraph_closeness()`.

Time complexity: the complexity of `igraph_closeness()` plus O(n), the number of vertices queried, for calculating the centralization score.

13.8.5. `igraph_centralization_eigenvector_centrality` — Calculate eigenvector centrality scores and graph centralization

```
int igraph_centralization_eigenvector_centrality(
      const igraph_t *graph,
      igraph_vector_t *vector,
      igraph_real_t *value,
      igraph_bool_t directed,
      igraph_bool_t scale,
      igraph_arpack_options_t *options,
      igraph_real_t *centralization,
      igraph_real_t *theoretical_max,
      igraph_bool_t normalized);
```

This function calculates the eigenvector centrality of the vertices by passing its arguments to `igraph_eigenvector_centrality`); and it calculates the graph level centralization index based on the results by calling `igraph_centralization()`.

Arguments:

graph:

The input graph.

vector:

A vector if you need the node-level eigenvector centrality scores, or a null pointer otherwise.

value:

If not a null pointer, then the leading eigenvalue is stored here.

scale:

If not zero then the result will be scaled, such that the absolute value of the maximum centrality is one.

options:

Options to ARPACK. See `igraph_arpack_options_t` for details. Note that the function overwrites the `n` (number of vertices) parameter and it always starts the calculation from a non-random vector calculated based on the degree of the vertices.

centralization:

Pointer to a real number, the centralization score is placed here.

theoretical_max:

Pointer to real number or a null pointer. If not a null pointer, then the theoretical maximum graph centrality score for a graph with the same number vertices is stored here.

normalized:

Boolean, whether to calculate a normalized centralization score. See `igraph_centralization()` for how the normalization is done.

Returns:

Error code.

See also:

`igraph_centralization()`, `igraph_eigenvector_centrality()`.

Time complexity: the complexity of `igraph_eigenvector_centrality()` plus O(|V|), the number of vertices for the calculating the centralization.

13.8.6. `igraph_centralization_degree_tmax` — Theoretical maximum for graph centralization based on degree

```
int igraph_centralization_degree_tmax(const igraph_t *graph,
          igraph_integer_t nodes,
          igraph_neimode_t mode,
          igraph_bool_t loops,
          igraph_real_t *res);
```

This function returns the theoretical maximum graph centrality based on vertex degree.

There are two ways to call this function, the first is to supply a graph as the `graph` argument, and then the number of vertices is taken from this object, and its directedness is considered as well. The `nodes` argument is ignored in this case. The `mode` argument is also ignored if the supplied graph is undirected.

The other way is to supply a null pointer as the `graph` argument. In this case the `nodes` and `mode` arguments are considered.

The most centralized structure is the star. More specifically, for undirected graphs it is the star, for directed graphs it is the in-star or the out-star.

Arguments:

`graph`:

A graph object or a null pointer, see the description above.

`nodes`:

The number of nodes. This is ignored if the `graph` argument is not a null pointer.

`mode`:

Constant, whether the calculation is based on in-degree (`IGRAPH_IN`), out-degree (`IGRAPH_OUT`) or total degree (`IGRAPH_ALL`). This is ignored if the `graph` argument is not a null pointer and the given graph is undirected.

`loops`:

Boolean scalar, whether to consider loop edges in the calculation.

`res`:

Pointer to a real variable, the result is stored here.

Returns:

Error code.

Time complexity: O(1).

See also:

`igraph_centralization_degree()` and `igraph_centralization()`.

13.8.7. `igraph_centralization_betweenness_tmax` — Theoretical maximum for graph centralization based on betweenness

```
int igraph_centralization_betweenness_tmax(const igraph_t *graph,
        igraph_integer_t nodes,
        igraph_bool_t directed,
        igraph_real_t *res);
```

This function returns the theoretical maximum graph centrality based on vertex betweenness.

There are two ways to call this function, the first is to supply a graph as the `graph` argument, and then the number of vertices is taken from this object, and its directedness is considered as well. The `nodes` argument is ignored in this case. The `directed` argument is also ignored if the supplied graph is undirected.

The other way is to supply a null pointer as the `graph` argument. In this case the `nodes` and `directed` arguments are considered.

The most centralized structure is the star.

Arguments:

`graph`:

A graph object or a null pointer, see the description above.

nodes:

The number of nodes. This is ignored if the `graph` argument is not a null pointer.

directed:

Boolean scalar, whether to use directed paths in the betweenness calculation. This argument is ignored if `graph` is not a null pointer and it is undirected.

res:

Pointer to a real variable, the result is stored here.

Returns:

Error code.

Time complexity: O(1).

See also:

`igraph_centralization_betweenness()` and `igraph_centralization()`.

13.8.8. `igraph_centralization_closeness_tmax` — Theoretical maximum for graph centralization based on closeness

```
int igraph_centralization_closeness_tmax(const igraph_t *graph,
      igraph_integer_t nodes,
      igraph_neimode_t mode,
      igraph_real_t *res);
```

This function returns the theoretical maximum graph centrality based on vertex closeness.

There are two ways to call this function, the first is to supply a graph as the `graph` argument, and then the number of vertices is taken from this object, and its directedness is considered as well. The `nodes` argument is ignored in this case. The `mode` argument is also ignored if the supplied graph is undirected.

The other way is to supply a null pointer as the `graph` argument. In this case the `nodes` and `mode` arguments are considered.

The most centralized structure is the star.

Arguments:

graph:

A graph object or a null pointer, see the description above.

nodes:

The number of nodes. This is ignored if the `graph` argument is not a null pointer.

mode:

Constant, specifies what kinf of distances to consider to calculate closeness. See the `mode` argument of `igraph_closeness()` for details. This argument is ignored if `graph` is not a null pointer and it is undirected.

res:

Pointer to a real variable, the result is stored here.

Returns:

Error code.

Time complexity: O(1).

See also:

`igraph_centralization_closeness()` and `igraph_centralization()`.

13.8.9. `igraph_centralization_eigenvector_centrality_tmax` — Theoretical maximum centralization for eigenvector centrality

```
int igraph_centralization_eigenvector_centrality_tmax(
      const igraph_t *graph,
      igraph_integer_t nodes,
      igraph_bool_t directed,
```

```
        igraph_bool_t scale,
        igraph_real_t *res);
```

This function returns the theoretical maximum graph centrality based on vertex eigenvector centrality.

There are two ways to call this function, the first is to supply a graph as the `graph` argument, and then the number of vertices is taken from this object, and its directedness is considered as well. The `nodes` argument is ignored in this case. The `directed` argument is also ignored if the supplied graph is undirected.

The other way is to supply a null pointer as the `graph` argument. In this case the `nodes` and `directed` arguments are considered.

The most centralized directed structure is the in-star. The most centralized undirected structure is the graph with a single edge.

Arguments:

`graph`:

A graph object or a null pointer, see the description above.

`nodes`:

The number of nodes. This is ignored if the `graph` argument is not a null pointer.

`directed`:

Boolean scalar, whether to consider edge directions. This argument is ignored if `graph` is not a null pointer and it is undirected.

`scale`:

Whether to rescale the node-level centrality scores to have a maximum of one.

`res`:

Pointer to a real variable, the result is stored here.

Returns:

Error code.

Time complexity: O(1).

See also:

`igraph_centralization_closeness()` and `igraph_centralization()`.

13.9. Similarity Measures

13.9.1. igraph_bibcoupling — Bibliographic coupling.

```
int igraph_bibcoupling(const igraph_t *graph, igraph_matrix_t *res,
                       const igraph_vs_t vids);
```

The bibliographic coupling of two vertices is the number of other vertices they both cite, `igraph_bibcoupling()` calculates this. The bibliographic coupling score for each given vertex and all other vertices in the graph will be calculated.

Arguments:

graph:

The graph object to analyze.

res:

Pointer to a matrix, the result of the calculation will be stored here. The number of its rows is the same as the number of vertex ids in *vids*, the number of columns is the number of vertices in the graph.

vids:

The vertex ids of the vertices for which the calculation will be done.

Returns:

Error code: `IGRAPH_EINVVID`: invalid vertex id.

Time complexity: O(|V|d^2), |V| is the number of vertices in the graph, d is the (maximum) degree of the vertices in the graph.

See also:

`igraph_cocitation()`

13.9.2. `igraph_cocitation` — Cocitation coupling.

```
int igraph_cocitation(const igraph_t *graph, igraph_matrix_t *res,
                      const igraph_vs_t vids);
```

Two vertices are cocited if there is another vertex citing both of them. `igraph_cocitation()` simply counts how many times two vertices are cocited. The cocitation score for each given vertex and all other vertices in the graph will be calculated.

Arguments:

graph:

The graph object to analyze.

res:

Pointer to a matrix, the result of the calculation will be stored here. The number of its rows is the same as the number of vertex ids in *vids*, the number of columns is the number of vertices in the graph.

vids:

The vertex ids of the vertices for which the calculation will be done.

Returns:

Error code: `IGRAPH_EINVVID`: invalid vertex id.

Time complexity: O(|V|d^2), |V| is the number of vertices in the graph, d is the (maximum) degree of the vertices in the graph.

See also:

`igraph_bibcoupling()`

Example 13-18. File examples/simple/igraph_cocitation.c

13.9.3. `igraph_similarity_jaccard` — Jaccard similarity coefficient for the given vertices.

```
int igraph_similarity_jaccard(const igraph_t *graph, igraph_matrix_t *res,
    const igraph_vs_t vids, igraph_neimode_t mode, igraph_bool_t loops);
```

The Jaccard similarity coefficient of two vertices is the number of common neighbors divided by the number of vertices that are neighbors of at least one of the two vertices being considered. This function calculates the pairwise Jaccard similarities for some (or all) of the vertices.

Arguments:

`graph`:

The graph object to analyze

`res`:

Pointer to a matrix, the result of the calculation will be stored here. The number of its rows and columns is the same as the number of vertex ids in *`vids`*.

`vids`:

The vertex ids of the vertices for which the calculation will be done.

`mode`:

The type of neighbors to be used for the calculation in directed graphs. Possible values:

`IGRAPH_OUT`

the outgoing edges will be considered for each node.

`IGRAPH_IN`

the incoming edges will be considered for each node.

`IGRAPH_ALL`

the directed graph is considered as an undirected one for the computation.

loops:

Whether to include the vertices themselves in the neighbor sets.

Returns:

Error code:

`IGRAPH_ENOMEM`

not enough memory for temporary data.

`IGRAPH_EINVVID`

invalid vertex id passed.

`IGRAPH_EINVMODE`

invalid mode argument.

Time complexity: O(|V|^2 d), |V| is the number of vertices in the vertex iterator given, d is the (maximum) degree of the vertices in the graph.

See also:

`igraph_similarity_dice()`, a measure very similar to the Jaccard coefficient

Example 13-19. File examples/simple/igraph_similarity.c

13.9.4. `igraph_similarity_jaccard_pairs` — Jaccard similarity coefficient for given vertex pairs.

```
int igraph_similarity_jaccard_pairs(const igraph_t *graph, igraph_vector_t *res,
 const igraph_vector_t *pairs, igraph_neimode_t mode, igraph_bool_t loops);
```

The Jaccard similarity coefficient of two vertices is the number of common neighbors divided by the number of vertices that are neighbors of at least one of the two vertices being considered. This function calculates the pairwise Jaccard similarities for a list of vertex pairs.

Arguments:

`graph`:

The graph object to analyze

`res`:

Pointer to a vector, the result of the calculation will be stored here. The number of elements is the same as the number of pairs in *`pairs`*.

`pairs`:

A vector that contains the pairs for which the similarity will be calculated. Each pair is defined by two consecutive elements, i.e. the first and second element of the vector specifies the first pair, the third and fourth element specifies the second pair and so on.

`mode`:

The type of neighbors to be used for the calculation in directed graphs. Possible values:

`IGRAPH_OUT`

the outgoing edges will be considered for each node.

`IGRAPH_IN`

the incoming edges will be considered for each node.

`IGRAPH_ALL`

the directed graph is considered as an undirected one for the computation.

`loops`:

Whether to include the vertices themselves in the neighbor sets.

Returns:

Error code:

`IGRAPH_ENOMEM`

not enough memory for temporary data.

`IGRAPH_EINVVID`

invalid vertex id passed.

`IGRAPH_EINVMODE`

invalid mode argument.

Time complexity: O(nd), n is the number of pairs in the given vector, d is the (maximum) degree of the vertices in the graph.

See also:

`igraph_similarity_jaccard()` to calculate the Jaccard similarity between all pairs of a vertex set, or `igraph_similarity_dice()` and `igraph_similarity_dice_pairs()` for a measure very similar to the Jaccard coefficient

Example 13-20. File examples/simple/igraph_similarity.c

13.9.5. `igraph_similarity_jaccard_es` — Jaccard similarity coefficient for a given edge selector.

```
int igraph_similarity_jaccard_es(const igraph_t *graph, igraph_vector_t *res,
 const igraph_es_t es, igraph_neimode_t mode, igraph_bool_t loops);
```

The Jaccard similarity coefficient of two vertices is the number of common neighbors divided by the number of vertices that are neighbors of at least one of the two vertices being considered. This function calculates the pairwise Jaccard similarities for the endpoints of edges in a given edge selector.

Arguments:

graph:

The graph object to analyze

res:

Pointer to a vector, the result of the calculation will be stored here. The number of elements is the same as the number of edges in *es*.

es:

An edge selector that specifies the edges to be included in the result.

mode:

The type of neighbors to be used for the calculation in directed graphs. Possible values:

`IGRAPH_OUT`

the outgoing edges will be considered for each node.

`IGRAPH_IN`

the incoming edges will be considered for each node.

`IGRAPH_ALL`

the directed graph is considered as an undirected one for the computation.

loops:

Whether to include the vertices themselves in the neighbor sets.

Returns:

Error code:

`IGRAPH_ENOMEM`

not enough memory for temporary data.

`IGRAPH_EINVVID`

invalid vertex id passed.

`IGRAPH_EINVMODE`

invalid mode argument.

Time complexity: O(nd), n is the number of edges in the edge selector, d is the (maximum) degree of the vertices in the graph.

See also:

`igraph_similarity_jaccard()` and `igraph_similarity_jaccard_pairs()` to calculate the Jaccard similarity between all pairs of a vertex set or some selected vertex pairs, or `igraph_similarity_dice()`, `igraph_similarity_dice_pairs()` and `igraph_similarity_dice_es()` for a measure very similar to the Jaccard coefficient

Example 13-21. File examples/simple/igraph_similarity.c

13.9.6. `igraph_similarity_dice` — Dice similarity coefficient.

```
int igraph_similarity_dice(const igraph_t *graph, igraph_matrix_t *res,
    const igraph_vs_t vids, igraph_neimode_t mode, igraph_bool_t loops);
```

The Dice similarity coefficient of two vertices is twice the number of common neighbors divided by the sum of the degrees of the vertices. This function calculates the pairwise Dice similarities for some (or all) of the vertices.

Arguments:

graph:

The graph object to analyze

res:

Pointer to a matrix, the result of the calculation will be stored here. The number of its rows and columns is the same as the number of vertex ids in *vids*.

vids:

The vertex ids of the vertices for which the calculation will be done.

mode:

The type of neighbors to be used for the calculation in directed graphs. Possible values:

`IGRAPH_OUT`

the outgoing edges will be considered for each node.

`IGRAPH_IN`

the incoming edges will be considered for each node.

`IGRAPH_ALL`

the directed graph is considered as an undirected one for the computation.

loops:

Whether to include the vertices themselves as their own neighbors.

Returns:

Error code:

`IGRAPH_ENOMEM`

not enough memory for temporary data.

`IGRAPH_EINVVID`

invalid vertex id passed.

`IGRAPH_EINVMODE`

invalid mode argument.

Time complexity: O(|V|^2 d), |V| is the number of vertices in the vertex iterator given, d is the (maximum) degree of the vertices in the graph.

See also:

`igraph_similarity_jaccard()`, a measure very similar to the Dice coefficient

Example 13-22. File examples/simple/igraph_similarity.c

13.9.7. `igraph_similarity_dice_pairs` — Dice similarity coefficient for given vertex pairs.

```
int igraph_similarity_dice_pairs(const igraph_t *graph, igraph_vector_t *res,
 const igraph_vector_t *pairs, igraph_neimode_t mode, igraph_bool_t loops);
```

The Dice similarity coefficient of two vertices is twice the number of common neighbors divided by the sum of the degrees of the vertices. This function calculates the pairwise Dice similarities for a list of vertex pairs.

Arguments:

graph:

The graph object to analyze

res:

Pointer to a vector, the result of the calculation will be stored here. The number of elements is the same as the number of pairs in *pairs*.

pairs:

A vector that contains the pairs for which the similarity will be calculated. Each pair is defined by two consecutive elements, i.e. the first and second element of the vector specifies the first pair, the third and fourth element specifies the second pair and so on.

mode:

The type of neighbors to be used for the calculation in directed graphs. Possible values:

`IGRAPH_OUT`

the outgoing edges will be considered for each node.

`IGRAPH_IN`

the incoming edges will be considered for each node.

`IGRAPH_ALL`

the directed graph is considered as an undirected one for the computation.

loops:

Whether to include the vertices themselves as their own neighbors.

Returns:

Error code:

`IGRAPH_ENOMEM`

not enough memory for temporary data.

`IGRAPH_EINVVID`

invalid vertex id passed.

`IGRAPH_EINVMODE`

invalid mode argument.

Time complexity: O(nd), n is the number of pairs in the given vector, d is the (maximum) degree of the vertices in the graph.

See also:

`igraph_similarity_dice()` to calculate the Dice similarity between all pairs of a vertex set, or `igraph_similarity_jaccard()`, `igraph_similarity_jaccard_pairs()` and `igraph_similarity_jaccard_es()` for a measure very similar to the Dice coefficient

Example 13-23. File examples/simple/igraph_similarity.c

13.9.8. `igraph_similarity_dice_es` — Dice similarity coefficient for a given edge selector.

```
int igraph_similarity_dice_es(const igraph_t *graph, igraph_vector_t *res,
 const igraph_es_t es, igraph_neimode_t mode, igraph_bool_t loops);
```

The Dice similarity coefficient of two vertices is twice the number of common neighbors divided by the sum of the degrees of the vertices. This function calculates the pairwise Dice similarities for the endpoints of edges in a given edge selector.

Arguments:

graph:

The graph object to analyze

res:

Pointer to a vector, the result of the calculation will be stored here. The number of elements is the same as the number of edges in *es*.

es:

An edge selector that specifies the edges to be included in the result.

mode:

The type of neighbors to be used for the calculation in directed graphs. Possible values:

IGRAPH_OUT

the outgoing edges will be considered for each node.

IGRAPH_IN

the incoming edges will be considered for each node.

IGRAPH_ALL

the directed graph is considered as an undirected one for the computation.

loops:

Whether to include the vertices themselves as their own neighbors.

Returns:

Error code:

IGRAPH_ENOMEM

not enough memory for temporary data.

IGRAPH_EINVVID

invalid vertex id passed.

IGRAPH_EINVMODE

invalid mode argument.

Time complexity: O(nd), n is the number of pairs in the given vector, d is the (maximum) degree of the vertices in the graph.

See also:

`igraph_similarity_dice()` and `igraph_similarity_dice_pairs()` to calculate the Dice similarity between all pairs of a vertex set or some selected vertex pairs, or `igraph_similarity_jaccard()`, `igraph_similarity_jaccard_pairs()` and `igraph_similarity_jaccard_es()` for a measure very similar to the Dice coefficient

Example 13-24. File examples/simple/igraph_similarity.c

13.9.9. `igraph_similarity_inverse_log_weighted` — Vertex similarity based on the inverse logarithm of vertex degrees.

```
int igraph_similarity_inverse_log_weighted(const igraph_t *graph,
  igraph_matrix_t *res, const igraph_vs_t vids, igraph_neimode_t mode);
```

The inverse log-weighted similarity of two vertices is the number of their common neighbors, weighted by the inverse logarithm of their degrees. It is based on the assumption that two vertices should be considered more similar if they share a low-degree common neighbor, since high-degree common neighbors are more likely to appear even by pure chance.

Isolated vertices will have zero similarity to any other vertex. Self-similarities are not calculated.

See the following paper for more details: Lada A. Adamic and Eytan Adar: Friends and neighbors on the Web. Social Networks, 25(3):211-230, 2003.

Arguments:

`graph`:

The graph object to analyze.

res:

Pointer to a matrix, the result of the calculation will be stored here. The number of its rows is the same as the number of vertex ids in *vids*, the number of columns is the number of vertices in the graph.

vids:

The vertex ids of the vertices for which the calculation will be done.

mode:

The type of neighbors to be used for the calculation in directed graphs. Possible values:

`IGRAPH_OUT`

the outgoing edges will be considered for each node. Nodes will be weighted according to their in-degree.

`IGRAPH_IN`

the incoming edges will be considered for each node. Nodes will be weighted according to their out-degree.

`IGRAPH_ALL`

the directed graph is considered as an undirected one for the computation. Every node is weighted according to its undirected degree.

Returns:

Error code: `IGRAPH_EINVVID`: invalid vertex id.

Time complexity: O(|V|d^2), |V| is the number of vertices in the graph, d is the (maximum) degree of the vertices in the graph.

Example 13-25. File examples/simple/igraph_similarity.c

13.10. Spanning Trees

13.10.1. `igraph_minimum_spanning_tree` — Calculates one minimum spanning tree of a graph.

```
int igraph_minimum_spanning_tree(const igraph_t* graph,
    igraph_vector_t* res, const igraph_vector_t* weights);
```

If the graph has more minimum spanning trees (this is always the case, except if it is a forest) this implementation returns only the same one.

Directed graphs are considered as undirected for this computation.

If the graph is not connected then its minimum spanning forest is returned. This is the set of the minimum spanning trees of each component.

Arguments:

graph:

The graph object.

res:

An initialized vector, the IDs of the edges that constitute a spanning tree will be returned here. Use `igraph_subgraph_edges()` to extract the spanning tree as a separate graph object.

weights:

A vector containing the weights of the edges in the same order as the simple edge iterator visits them (i.e. in increasing order of edge IDs).

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data.

Time complexity: O(|V|+|E|) for the unweighted case, O(|E| log |V|) for the weighted case. |V| is the number of vertices, |E| the number of edges in the graph.

See also:

`igraph_minimum_spanning_tree_unweighted()` and `igraph_minimum_spanning_tree_prim()` if you only need the tree as a separate graph object.

Example 13-26. File examples/simple/igraph_minimum_spanning_tree.c

13.10.2. `igraph_minimum_spanning_tree_unweighted` — Calculates one minimum spanning tree of an unweighted graph.

```
int igraph_minimum_spanning_tree_unweighted(const igraph_t *graph,
          igraph_t *mst);
```

If the graph has more minimum spanning trees (this is always the case, except if it is a forest) this implementation returns only the same one.

Directed graphs are considered as undirected for this computation.

If the graph is not connected then its minimum spanning forest is returned. This is the set of the minimum spanning trees of each component.

Arguments:

graph:

The graph object.

mst:

The minimum spanning tree, another graph object. Do *not* initialize this object before passing it to this function, but be sure to call `igraph_destroy()` on it if you don't need it any more.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data.

Time complexity: O(|V|+|E|), |V| is the number of vertices, |E| the number of edges in the graph.

See also:

`igraph_minimum_spanning_tree_prim()` for weighted graphs, `igraph_minimum_spanning_tree()` if you need the IDs of the edges that constitute the spanning tree.

13.10.3. `igraph_minimum_spanning_tree_prim` — Calculates one minimum spanning tree of a weighted graph.

```
int igraph_minimum_spanning_tree_prim(const igraph_t *graph, igraph_t *mst,
          const igraph_vector_t *weights);
```

This function uses Prim's method for carrying out the computation, see Prim, R.C.: Shortest connection networks and some generalizations, Bell System Technical Journal, Vol. 36, 1957, 1389--1401.

If the graph has more than one minimum spanning tree, the current implementation returns always the same one.

Directed graphs are considered as undirected for this computation.

If the graph is not connected then its minimum spanning forest is returned. This is the set of the minimum spanning trees of each component.

Arguments:

graph:

The graph object.

mst:

The result of the computation, a graph object containing the minimum spanning tree of the graph. Do *not* initialize this object before passing it to this function, but be sure to call `igraph_destroy()` on it if you don't need it any more.

weights:

A vector containing the weights of the edges in the same order as the simple edge iterator visits them (i.e. in increasing order of edge IDs).

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory. `IGRAPH_EINVAL`, length of weight vector does not match number of edges.

Time complexity: O(|E| log |V|), |V| is the number of vertices, |E| the number of edges in the graph.

See also:

`igraph_minimum_spanning_tree_unweighted()` for unweighted graphs, `igraph_minimum_spanning_tree()` if you need the IDs of the edges that constitute the spanning tree.

Example 13-27. File examples/simple/igraph_minimum_spanning_tree.c

13.11. Transitivity or Clustering Coefficient

13.11.1. `igraph_transitivity_undirected` — Calculates the transitivity (clustering coefficient) of a graph.

```
int igraph_transitivity_undirected(const igraph_t *graph,
       igraph_real_t *res,
       igraph_transitivity_mode_t mode);
```

The transitivity measures the probability that two neighbors of a vertex are connected. More precisely, this is the ratio of the triangles and connected triples in the graph, the result is a single real number. Directed graphs are considered as undirected ones.

Note that this measure is different from the local transitivity measure (see `igraph_transitivity_local_undirected()`) as it calculates a single value for the whole graph. See the following reference for more details:

S. Wasserman and K. Faust: Social Network Analysis: Methods and Applications. Cambridge: Cambridge University Press, 1994.

Clustering coefficient is an alternative name for transitivity.

Arguments:

`graph`:

The graph object.

`res`:

Pointer to a real variable, the result will be stored here.

`mode`:

Defines how to treat graphs with no connected triples. `IGRAPH_TRANSITIVITY_NAN` returns `NaN` in this case, `IGRAPH_TRANSITIVITY_ZERO` returns zero.

Returns:

Error code: `IGRAPH_ENOMEM`: not enough memory for temporary data.

See also:

`igraph_transitivity_local_undirected()`, `igraph_transitivity_avglocal_undirected()`.

Time complexity: O(|V|*d^2), |V| is the number of vertices in the graph, d is the average node degree.

Example 13-28. File examples/simple/igraph_transitivity.c

13.11.2. `igraph_transitivity_local_undirected` — Calculates the local transitivity (clustering coefficient) of a graph.

```
int igraph_transitivity_local_undirected(const igraph_t *graph,
      igraph_vector_t *res,
      const igraph_vs_t vids,
      igraph_transitivity_mode_t mode);
```

The transitivity measures the probability that two neighbors of a vertex are connected. In case of the local transitivity, this probability is calculated separately for each vertex.

Note that this measure is different from the global transitivity measure (see `igraph_transitivity_undirected()`) as it calculates a transitivity value for each vertex individually. See the following reference for more details:

D. J. Watts and S. Strogatz: Collective dynamics of small-world networks. Nature 393(6684):440-442 (1998).

Clustering coefficient is an alternative name for transitivity.

Arguments:

`graph`:

The input graph, it can be directed but direction of the edges will be ignored.

`res`:

Pointer to an initialized vector, the result will be stored here. It will be resized as needed.

`vids`:

Vertex set, the vertices for which the local transitivity will be calculated.

`mode`:

Defines how to treat vertices with degree less than two. `IGRAPH_TRANSITIVITY_NAN` returns `NaN` for these vertices, `IGRAPH_TRANSITIVITY_ZERO` returns zero.

Returns:

Error code.

See also:

`igraph_transitivity_undirected()`,
`igraph_transitivity_avglocal_undirected()`.

Time complexity: O(n*d^2), n is the number of vertices for which the transitivity is calculated, d is the average vertex degree.

13.11.3. `igraph_transitivity_avglocal_undirected` — Average local transitivity (clustering coefficient).

```
int igraph_transitivity_avglocal_undirected(const igraph_t *graph,
         igraph_real_t *res,
         igraph_transitivity_mode_t mode);
```

The transitivity measures the probability that two neighbors of a vertex are connected. In case of the average local transitivity, this probability is calculated for each vertex and then the average is taken. Vertices with less than two neighbors require special treatment, they will either be left out from the calculation or they will be considered as having zero transitivity, depending on the `mode` argument.

Note that this measure is different from the global transitivity measure (see `igraph_transitivity_undirected()`) as it simply takes the average local transitivity across the whole network. See the following reference for more details:

D. J. Watts and S. Strogatz: Collective dynamics of small-world networks. Nature 393(6684):440-442 (1998).

Clustering coefficient is an alternative name for transitivity.

Arguments:

`graph`:

The input graph, directed graphs are considered as undirected ones.

`res`:

Pointer to a real variable, the result will be stored here.

mode:

Defines how to treat vertices with degree less than two. `IGRAPH_TRANSITIVITY_NAN` leaves them out from averaging, `IGRAPH_TRANSITIVITY_ZERO` includes them with zero transitivity. The result will be `NaN` if the mode is `IGRAPH_TRANSITIVITY_NAN` and there are no vertices with more than one neighbor.

Returns:

Error code.

See also:

`igraph_transitivity_undirected()`, `igraph_transitivity_local_undirected()`.

Time complexity: O(|V|*d^2), |V| is the number of vertices in the graph and d is the average degree.

13.11.4. `igraph_transitivity_barrat` — Weighted transitivity, as defined by A. Barrat.

```
int igraph_transitivity_barrat(const igraph_t *graph,
          igraph_vector_t *res,
          const igraph_vs_t vids,
          const igraph_vector_t *weights,
       igraph_transitivity_mode_t mode);
```

This is a local transitivity, i.e. a vertex-level index. For a given vertex `i`, from all triangles in which it participates we consider the weight of the edges incident on `i`. The transitivity is the sum of these weights divided by twice the strength of the vertex (see `igraph_strength()`) and the degree of the vertex minus one. See Alain Barrat, Marc Barthelemy, Romualdo Pastor-Satorras, Alessandro Vespignani: The architecture of complex weighted networks, Proc. Natl. Acad. Sci. USA 101, 3747 (2004) at http://arxiv.org/abs/cond-mat/0311416 for the exact formula.

Arguments:

graph:

The input graph, edge directions are ignored for directed graphs. Note that the function does NOT work for non-simple graphs.

res:

Pointer to an initialized vector, the result will be stored here. It will be resized as needed.

vids:

The vertices for which the calculation is performed.

weights:

Edge weights. If this is a null pointer, then a warning is given and `igraph_transitivity_local_undirected()` is called.

mode:

Defines how to treat vertices with zero strength. `IGRAPH_TRANSITIVITY_NAN` says that the transitivity of these vertices is `NaN`, `IGRAPH_TRANSITIVITY_ZERO` says it is zero.

Returns:

Error code.

Time complexity: O(|V|*d^2), |V| is the number of vertices in the graph, d is the average node degree.

See also:

`igraph_transitivity_undirected()`, `igraph_transitivity_local_undirected()` and `igraph_transitivity_avglocal_undirected()` for other kinds of (non-weighted) transitivity.

13.12. Directedness conversion

13.12.1. `igraph_to_directed` — Convert an undirected graph to a directed one

```
int igraph_to_directed(igraph_t *graph,
          igraph_to_directed_t mode);
```

If the supplied graph is directed, this function does nothing.

Arguments:

graph:

The graph object to convert.

mode:

Constant, specifies the details of how exactly the conversion is done. Possible values: `IGRAPH_TO_DIRECTED_ARBITRARY`: the number of edges in the graph stays the same, an arbitrarily directed edge is created for each undirected edge; `IGRAPH_TO_DIRECTED_MUTUAL`: two directed edges are created for each undirected edge, one in each direction.

Returns:

Error code.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

13.12.2. `igraph_to_undirected` — Convert a directed graph to an undirected one.

```
int igraph_to_undirected(igraph_t *graph,
    igraph_to_undirected_t mode,
    const igraph_attribute_combination_t *edge_comb);
```

If the supplied graph is undirected, this function does nothing.

Arguments:

graph:

The graph object to convert.

mode:

Constant, specifies the details of how exactly the conversion is done. Possible values: `IGRAPH_TO_UNDIRECTED_EACH`: the number of edges remains constant, an undirected edge is created for each directed one, this version might create graphs with multiple edges; `IGRAPH_TO_UNDIRECTED_COLLAPSE`: one undirected edge will be created for each pair of vertices which are connected with at least one directed edge, no multiple edges will be created. `IGRAPH_TO_UNDIRECTED_MUTUAL` creates an undirected edge for each pair of mutual edges in the directed graph. Non-mutual edges are lost. This mode might create multiple edges.

edge_comb:

What to do with the edge attributes. See the igraph manual section about attributes for details.

Returns:

Error code.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

Example 13-29. File examples/simple/igraph_to_undirected.c

13.13. Spectral properties

13.13.1. `igraph_laplacian` — Returns the Laplacian matrix of a graph

```
int igraph_laplacian(const igraph_t *graph, igraph_matrix_t *res,
       igraph_sparsemat_t *sparseres,
                       igraph_bool_t normalized,
                       const igraph_vector_t *weights);
```

The graph Laplacian matrix is similar to an adjacency matrix but contains -1's instead of 1's and the vertex degrees are included in the diagonal. So the result for edge i--j is -1 if i!=j and is equal to the degree of vertex i if i==j. igraph_laplacian will work on a directed graph; in this case, the diagonal will contain the out-degrees. Loop edges will be ignored.

The normalized version of the Laplacian matrix has 1 in the diagonal and -1/sqrt(d[i]d[j]) if there is an edge from i to j.

The first version of this function was written by Vincent Matossian.

Arguments:

graph:

Pointer to the graph to convert.

res:

Pointer to an initialized matrix object, the result is stored here. It will be resized if needed. If it is a null pointer, then it is ignored. At least one of *res* and *sparseres* must be a non-null pointer.

sparseres:

Pointer to an initialized sparse matrix object, the result is stored here, if it is not a null pointer. At least one of *res* and *sparseres* must be a non-null pointer.

normalized:

Whether to create a normalized Laplacian matrix.

weights:

An optional vector containing edge weights, to calculate the weighted Laplacian matrix. Set it to a null pointer to calculate the unweighted Laplacian.

Returns:

Error code.

Time complexity: O(|V||V|), |V| is the number of vertices in the graph.

Example 13-30. File examples/simple/igraph_laplacian.c

13.14. Non-simple graphs: multiple and loop edges

13.14.1. `igraph_is_simple` — Decides whether the input graph is a simple graph.

```
int igraph_is_simple(const igraph_t *graph, igraph_bool_t *res);
```

A graph is a simple graph if it does not contain loop edges and multiple edges.

Arguments:

graph:

The input graph.

res:

Pointer to a boolean constant, the result is stored here.

Returns:

Error code.

See also:

`igraph_is_loop()` and `igraph_is_multiple()` to find the loops and multiple edges, `igraph_simplify()` to get rid of them, or `igraph_has_multiple()` to decide whether there is at least one multiple edge.

Time complexity: O(|V|+|E|).

13.14.2. `igraph_is_loop` — Find the loop edges in a graph.

```
int igraph_is_loop(const igraph_t *graph, igraph_vector_bool_t *res,
     igraph_es_t es);
```

A loop edge is an edge from a vertex to itself.

Arguments:

graph:

The input graph.

res:

Pointer to an initialized boolean vector for storing the result, it will be resized as needed.

es:

The edges to check, for all edges supply `igraph_ess_all()` here.

Returns:

Error code.

See also:

`igraph_simplify()` to get rid of loop edges.

Time complexity: O(e), the number of edges to check.

Example 13-31. File examples/simple/igraph_is_loop.c

13.14.3. `igraph_is_multiple` — Find the multiple edges in a graph.

```
int igraph_is_multiple(const igraph_t *graph, igraph_vector_bool_t *res,
         igraph_es_t es);
```

An edge is a multiple edge if there is another edge with the same head and tail vertices in the graph.

Note that this function returns true only for the second or more appearances of the multiple edges.

Arguments:

graph:

The input graph.

res:

Pointer to a boolean vector, the result will be stored here. It will be resized as needed.

es:

The edges to check. Supply `igraph_ess_all()` if you want to check all edges.

Returns:

Error code.

See also:

`igraph_count_multiple()`, `igraph_has_multiple()` and `igraph_simplify()`.

Time complexity: O(e*d), e is the number of edges to check and d is the average degree (out-degree in directed graphs) of the vertices at the tail of the edges.

Example 13-32. File examples/simple/igraph_is_multiple.c

13.14.4. `igraph_has_multiple` — Check whether the graph has at least one multiple edge.

```
int igraph_has_multiple(const igraph_t *graph, igraph_bool_t *res);
```

An edge is a multiple edge if there is another edge with the same head and tail vertices in the graph.

Arguments:

`graph`:

The input graph.

`res`:

Pointer to a boolean variable, the result will be stored here.

Returns:

Error code.

See also:

`igraph_count_multiple()`, `igraph_is_multiple()` and `igraph_simplify()`.

Time complexity: O(e*d), e is the number of edges to check and d is the average degree (out-degree in directed graphs) of the vertices at the tail of the edges.

Example 13-33. File examples/simple/igraph_has_multiple.c

13.14.5. `igraph_count_multiple` — Count the number of appearances of the edges in a graph.

```
int igraph_count_multiple(const igraph_t *graph, igraph_vector_t *res, igraph_es_t es);
```

If the graph has no multiple edges then the result vector will be filled with ones. (An edge is a multiple edge if there is another edge with the same head and tail vertices in the graph.)

Arguments:

`graph`:

The input graph.

`res`:

Pointer to a vector, the result will be stored here. It will be resized as needed.

`es`:

The edges to check. Supply `igraph_ess_all()` if you want to check all edges.

Returns:

Error code.

See also:

`igraph_is_multiple()` and `igraph_simplify()`.

Time complexity: O(e*d), e is the number of edges to check and d is the average degree (out-degree in directed graphs) of the vertices at the tail of the edges.

13.14.6. `igraph_simplify` — Removes loop and/or multiple edges from the graph.

```
int igraph_simplify(igraph_t *graph, igraph_bool_t multiple,
      igraph_bool_t loops,
      const igraph_attribute_combination_t *edge_comb);
```

Arguments:

graph:

The graph object.

multiple:

Logical, if true, multiple edges will be removed.

loops:

Logical, if true, loops (self edges) will be removed.

edge_comb:

What to do with the edge attributes. See the igraph manual section about attributes for details.

Returns:

Error code: `IGRAPH_ENOMEM` if we are out of memory.

Time complexity: O(|V|+|E|).

Example 13-34. File examples/simple/igraph_simplify.c

13.15. Mixing patterns

13.15.1. `igraph_assortativity_nominal` — Assortativity of a graph based on vertex categories

```
int igraph_assortativity_nominal(const igraph_t *graph,
     const igraph_vector_t *types,
     igraph_real_t *res,
     igraph_bool_t directed);
```

Assuming the vertices of the input graph belong to different categories, this function calculates the assortativity coefficient of the graph. The assortativity coefficient is between minus one and one and it is one if all connections stay within categories, it is minus one, if the network is perfectly disassortative. For a randomly connected network it is (asymptotically) zero.

See equation (2) in M. E. J. Newman: Mixing patterns in networks, Phys. Rev. E 67, 026126 (2003) (http://arxiv.org/abs/cond-mat/0209450) for the proper definition.

Arguments:

`graph`:

The input graph, it can be directed or undirected.

`types`:

Vector giving the vertex types. They are assumed to be integer numbers, starting with zero.

`res`:

Pointer to a real variable, the result is stored here.

`directed`:

Boolean, it gives whether to consider edge directions in a directed graph. It is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|E|+t), |E| is the number of edges, t is the number of vertex types.

See also:

`igraph_assortativity` if the vertex types are defines by numeric values (e.g. vertex degree), instead of categories.

Example 13-35. File examples/simple/assortativity.c

13.15.2. `igraph_assortativity` — Assortativity based on numeric properties of vertices

```
int igraph_assortativity(const igraph_t *graph,
    const igraph_vector_t *types1,
    const igraph_vector_t *types2,
    igraph_real_t *res,
    igraph_bool_t directed);
```

This function calculates the assortativity coefficient of the input graph. This coefficient is basically the correlation between the actual connectivity patterns of the vertices and the pattern expected from the distribution of the vertex types.

See equation (21) in M. E. J. Newman: Mixing patterns in networks, Phys. Rev. E 67, 026126 (2003) (http://arxiv.org/abs/cond-mat/0209450) for the proper definition. The actual calculation is performed using equation (26) in the same paper for directed graphs, and equation (4) in M. E. J. Newman: Assortative mixing in networks, Phys. Rev. Lett. 89, 208701 (2002) (http://arxiv.org/abs/cond-mat/0205405/) for undirected graphs.

Arguments:

`graph`:

The input graph, it can be directed or undirected.

`types1`:

The vertex values, these can be arbitrary numeric values.

`types2`:

A second value vector to be using for the incoming edges when calculating assortativity for a directed graph. Supply a null pointer here if you want to use the same values for outgoing and incoming edges. This argument is ignored (with a warning) if it is not a null pointer and undirected assortativity coefficient is being calculated.

res:

Pointer to a real variable, the result is stored here.

directed:

Boolean, whether to consider edge directions for directed graphs. It is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|E|), linear in the number of edges of the graph.

See also:

`igraph_assortativity_nominal()` if you have discrete vertex categories instead of numeric labels, and `igraph_assortativity_degree()` for the special case of assortativity based on vertex degree.

Example 13-36. File examples/simple/assortativity.c

13.15.3. `igraph_assortativity_degree` — Assortativity of a graph based on vertex degree

```
int igraph_assortativity_degree(const igraph_t *graph,
    igraph_real_t *res,
    igraph_bool_t directed);
```

Assortativity based on vertex degree, please see the discussion at the documentation of `igraph_assortativity()` for details.

Arguments:

graph:

The input graph, it can be directed or undirected.

res:

Pointer to a real variable, the result is stored here.

directed:

Boolean, whether to consider edge directions for directed graphs. This argument is ignored for undirected graphs. Supply 1 (=TRUE) here to do the natural thing, i.e. use directed version of the measure for directed graphs and the undirected version for undirected graphs.

Returns:

Error code.

Time complexity: O(|E|+|V|), |E| is the number of edges, |V| is the number of vertices.

See also:

`igraph_assortativity()` for the general function calculating assortativity for any kind of numeric vertex values.

Example 13-37. File examples/simple/assortativity.c

13.16. K-Cores

13.16.1. `igraph_coreness` — Finding the coreness of the vertices in a network.

```
int igraph_coreness(const igraph_t *graph, igraph_vector_t *cores,
```

```
    igraph_neimode_t mode);
```

The k-core of a graph is a maximal subgraph in which each vertex has at least degree k. (Degree here means the degree in the subgraph of course.). The coreness of a vertex is the highest order of a k-core containing the vertex.

This function implements the algorithm presented in Vladimir Batagelj, Matjaz Zaversnik: An O(m) Algorithm for Cores Decomposition of Networks.

Arguments:

graph:

The input graph.

cores:

Pointer to an initialized vector, the result of the computation will be stored here. It will be resized as needed. For each vertex it contains the highest order of a core containing the vertex.

mode:

For directed graph it specifies whether to calculate in-cores, out-cores or the undirected version. It is ignored for undirected graphs. Possible values: `IGRAPH_ALL` undirected version, `IGRAPH_IN` in-cores, `IGRAPH_OUT` out-cores.

Returns:

Error code.

Time complexity: O(|E|), the number of edges.

13.17. Topological sorting, directed acyclic graphs

13.17.1. `igraph_is_dag` — Checks whether a graph is a directed acyclic graph (DAG) or not.

```
int igraph_is_dag(const igraph_t* graph, igraph_bool_t *res);
```

A directed acyclic graph (DAG) is a directed graph with no cycles.

Arguments:

graph:

The input graph.

res:

Pointer to a boolean constant, the result is stored here.

Returns:

Error code.

Time complexity: O(|V|+|E|), where |V| and |E| are the number of vertices and edges in the original input graph.

See also:

`igraph_topological_sorting()` to get a possible topological sorting of a DAG.

13.17.2. `igraph_topological_sorting` — Calculate a possible topological sorting of the graph.

```
int igraph_topological_sorting(const igraph_t* graph, igraph_vector_t *res,
          igraph_neimode_t mode);
```

A topological sorting of a directed acyclic graph is a linear ordering of its nodes where each node comes before all nodes to which it has edges. Every DAG has at least one topological sort, and may have many. This function returns a possible topological sort among them. If the graph is not acyclic (it has at least one cycle), a partial topological sort is returned and a warning is issued.

Arguments:

graph:

The input graph.

res:

Pointer to a vector, the result will be stored here. It will be resized if needed.

mode:

Specifies how to use the direction of the edges. For `IGRAPH_OUT`, the sorting order ensures that each node comes before all nodes to which it has edges, so nodes with no incoming edges go first. For `IGRAPH_IN`, it is quite the opposite: each node comes before all nodes from which it receives edges. Nodes with no outgoing edges go first.

Returns:

Error code.

Time complexity: O(|V|+|E|), where |V| and |E| are the number of vertices and edges in the original input graph.

See also:

`igraph_is_dag()` if you are only interested in whether a given graph is a DAG or not, or `igraph_feedback_arc_set()` to find a set of edges whose removal makes the graph a DAG.

Example 13-38. File examples/simple/igraph_topological_sorting.c

13.17.3. `igraph_feedback_arc_set` — Calculates a feedback arc set of the graph using different

```
int igraph_feedback_arc_set(const igraph_t *graph, igraph_vector_t *result,
        const igraph_vector_t *weights, igraph_fas_algorithm_t algo);
```

algorithms.

A feedback arc set is a set of edges whose removal makes the graph acyclic. We are usually interested in *minimum* feedback arc sets, i.e. sets of edges whose total weight is minimal among all the feedback arc sets.

For undirected graphs, the problem is simple: one has to find a maximum weight spanning tree and then remove all the edges not in the spanning tree. For directed graphs, this is an NP-hard problem, and various heuristics are usually used to find an approximate solution to the problem. This function implements a few of these heuristics.

Arguments:

`graph`:

The graph object.

`result`:

An initialized vector, the result will be returned here.

`weights`:

Weight vector or NULL if no weights are specified.

`algo`:

The algorithm to use to solve the problem if the graph is directed. Possible values:

`IGRAPH_FAS_EXACT_IP`

Finds a *minimum* feedback arc set using integer programming (IP). The complexity of this algorithm is exponential of course.

`IGRAPH_FAS_APPROX_EADES`

Finds a feedback arc set using the heuristic of Eades, Lin and Smyth (1993). This is guaranteed to be smaller than |E|/2 - |V|/6, and it is linear in the number of edges (i.e. O(|E|)). For more details, see Eades P, Lin X and Smyth WF: A fast and effective heuristic for the feedback arc set problem. In: Proc Inf Process Lett 319-323, 1993.

Returns:

Error code: `IGRAPH_EINVAL` if an unknown method was specified or the weight vector is invalid.

Example 13-39. File examples/simple/igraph_feedback_arc_set.c

Example 13-40. File examples/simple/igraph_feedback_arc_set_ip.c

Time complexity: depends on *algo*, see the time complexities there.

13.18. Maximum cardinality search, graph decomposition, chordal graphs

13.18.1. `igraph_maximum_cardinality_search` — Maximum cardinality search

```
int igraph_maximum_cardinality_search(const igraph_t *graph,
          igraph_vector_t *alpha,
          igraph_vector_t *alpham1);
```

This function implements the maximum cardinality search algorithm discussed in Robert E Tarjan and Mihalis Yannakakis: Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. SIAM Journal of Computation 13, 566--579, 1984.

Arguments:

graph:

The input graph. Can be directed, but the direction of the edges is ignored.

alpha:

Pointer to an initialized vector, the result is stored here. It will be resized, as needed. Upon return it contains the rank of the each vertex.

alpham1:

Pointer to an initialized vector or a `NULL` pointer. If not `NULL`, then the inverse of *alpha* is stored here.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in terms of the number of vertices and edges.

See also:

`igraph_is_chordal()`.

13.18.2. `igraph_is_chordal` — Decides whether a graph is chordal

```
int igraph_is_chordal(const igraph_t *graph,
        const igraph_vector_t *alpha,
        const igraph_vector_t *alpham1,
        igraph_bool_t *chordal,
        igraph_vector_t *fill_in,
        igraph_t *newgraph);
```

A graph is chordal if each of its cycles of four or more nodes has a chord, which is an edge joining two nodes that are not adjacent in the cycle. An equivalent definition is that any chordless cycles have at most three nodes. If either *`alpha`* or *`alpha1`* is given, then the other is calculated by taking simply the inverse. If neither are given, then `igraph_maximum_cardinality_search()` is called to calculate them.

Arguments:

`graph`:

The input graph, it might be directed, but edge direction is ignored.

`alpha`:

Either an alpha vector coming from `igraph_maximum_cardinality_search()` (on the same graph), or a null pointer.

`alpham1`:

Either an inverse alpha vector coming from `igraph_maximum_cardinality_search()` (on the same graph) or a null pointer.

`chordal`:

Pointer to a boolean, the result is stored here.

fill_in:

Pointer to an initialized vector, or a null pointer. If not a null pointer, then the fill-in of the graph is stored here. The fill-in is the set of edges that are needed to make the graph chordal. The vector is resized as needed.

newgraph:

Pointer to an uninitialized graph, or a null pointer. If not a null pointer, then a new triangulated graph is created here. This essentially means adding the fill-in edges to the original graph.

Returns:

Error code.

Time complexity: O(n).

See also:

`igraph_maximum_cardinality_search()`.

13.19. Matchings

13.19.1. `igraph_is_matching` — Checks whether the given matching is valid for the given graph.

```
int igraph_is_matching(const igraph_t* graph,
    const igraph_vector_bool_t* types, const igraph_vector_long_t* matching,
    igraph_bool_t* result);
```

This function checks a matching vector and verifies whether its length matches the number of vertices in the given graph, its values are between -1 (inclusive) and the number of vertices (exclusive), and whether there exists a corresponding edge in the graph for every matched vertex pair. For bipartite graphs, it also verifies whether the matched vertices are in different parts of the graph.

Arguments:

graph:

The input graph. It can be directed but the edge directions will be ignored.

types:

If the graph is bipartite and you are interested in bipartite matchings only, pass the vertex types here. If the graph is non-bipartite, simply pass `NULL`.

matching:

The matching itself. It must be a vector where element i contains the ID of the vertex that vertex i is matched to, or -1 if vertex i is unmatched.

result:

Pointer to a boolean variable, the result will be returned here.

See also:

`igraph_is_maximal_matching()` if you are also interested in whether the matching is maximal (i.e. non-extendable).

Time complexity: O(|V|+|E|) where |V| is the number of vertices and |E| is the number of edges.

Example 13-41. File examples/simple/igraph_maximum_bipartite_matching.c

13.19.2. `igraph_is_maximal_matching` — Checks whether a matching in a graph is maximal.

```
int igraph_is_maximal_matching(const igraph_t* graph,
    const igraph_vector_bool_t* types, const igraph_vector_long_t* matching,
    igraph_bool_t* result);
```

A matching is maximal if and only if there exists no unmatched vertex in a graph such that one of its neighbors is also unmatched.

Arguments:

graph:

The input graph. It can be directed but the edge directions will be ignored.

types:

If the graph is bipartite and you are interested in bipartite matchings only, pass the vertex types here. If the graph is non-bipartite, simply pass `NULL`.

matching:

The matching itself. It must be a vector where element i contains the ID of the vertex that vertex i is matched to, or -1 if vertex i is unmatched.

result:

Pointer to a boolean variable, the result will be returned here.

See also:

`igraph_is_matching()` if you are only interested in whether a matching vector is valid for a given graph.

Time complexity: O(|V|+|E|) where |V| is the number of vertices and |E| is the number of edges.

Example 13-42. File examples/simple/igraph_maximum_bipartite_matching.c

13.19.3. `igraph_maximum_bipartite_matching` — Calculates a maximum matching in a bipartite graph.

```
int igraph_maximum_bipartite_matching(const igraph_t* graph,
    const igraph_vector_bool_t* types, igraph_integer_t* matching_size,
    igraph_real_t* matching_weight, igraph_vector_long_t* matching,
    const igraph_vector_t* weights, igraph_real_t eps);
```

A matching in a bipartite graph is a partial assignment of vertices of the first kind to vertices of the second kind such that each vertex of the first kind is matched to at most one vertex of the second kind and vice versa, and matched vertices must be connected by an edge in the graph. The size (or cardinality) of a matching is the number of edges. A matching is a maximum matching if there exists no other matching with larger cardinality. For weighted graphs, a maximum matching is a matching whose edges have the largest possible total weight among all possible matchings.

Maximum matchings in bipartite graphs are found by the push-relabel algorithm with greedy initialization and a global relabeling after every n/2 steps where n is the number of vertices in the graph.

References: Cherkassky BV, Goldberg AV, Martin P, Setubal JC and Stolfi J: Augment or push: A computational study of bipartite matching and unit-capacity flow algorithms. ACM Journal of Experimental Algorithmics 3, 1998.

Kaya K, Langguth J, Manne F and Ucar B: Experiments on push-relabel-based maximum cardinality matching algorithms for bipartite graphs. Technical Report TR/PA/11/33 of the Centre Europeen de Recherche et de Formation Avancee en Calcul Scientifique, 2011.

Arguments:

graph:

The input graph. It can be directed but the edge directions will be ignored.

types:

Boolean vector giving the vertex types of the graph.

matching_size:

The size of the matching (i.e. the number of matched vertex pairs will be returned here). It may be `NULL` if you don't need this.

matching_weight:

The weight of the matching if the edges are weighted, or the size of the matching again if the edges are unweighted. It may be `NULL` if you don't need this.

matching:

The matching itself. It must be a vector where element i contains the ID of the vertex that vertex i is matched to, or -1 if vertex i is unmatched.

weights:

A null pointer (=no edge weights), or a vector giving the weights of the edges. Note that the algorithm is stable only for integer weights.

eps:

A small real number used in equality tests in the weighted bipartite matching algorithm. Two real numbers are considered equal in the algorithm if their difference is smaller than `eps`. This is required to avoid the accumulation of numerical errors. It is advised to pass a value derived from the `DBL_EPSILON` constant in `float`.h here. If you are running the algorithm with no `weights` vector, this argument is ignored.

Returns:

Error code.

Time complexity: O(sqrt(|V|) |E|) for unweighted graphs (according to the technical report referenced above), O(|V||E|) for weighted graphs.

Example 13-43. File examples/simple/igraph_maximum_bipartite_matching.c

13.20. Line graphs

13.20.1. `igraph_linegraph` — Create the line graph of a graph.

```
int igraph_linegraph(const igraph_t *graph, igraph_t *linegraph);
```

The line graph L(G) of a G undirected graph is defined as follows. L(G) has one vertex for each edge in G and two vertices in L(G) are connected by an edge if their corresponding edges share an end point.

The line graph L(G) of a G directed graph is slightly different, L(G) has one vertex for each edge in G and two vertices in L(G) are connected by a directed edge if the target of the first vertex's corresponding edge is the same as the source of the second vertex's corresponding edge.

Edge *i* in the original graph will correspond to vertex *i* in the line graph.

The first version of this function was contributed by Vincent Matossian, thanks.

Arguments:

`graph`:

The input graph, may be directed or undirected.

linegraph:

Pointer to an uninitialized graph object, the result is stored here.

Returns:

Error code.

Time complexity: O(|V|+|E|), the number of edges plus the number of vertices.

13.21. Unfolding a graph into a tree

13.21.1. `igraph_unfold_tree` — Unfolding a graph into a tree, by possibly multiplicating its vertices.

```
int igraph_unfold_tree(const igraph_t *graph, igraph_t *tree,
          igraph_neimode_t mode, const igraph_vector_t *roots,
          igraph_vector_t *vertex_index);
```

A graph is converted into a tree (or forest, if it is unconnected), by performing a breadth-first search on it, and replicating vertices that were found a second, third, etc. time.

Arguments:

graph:

The input graph, it can be either directed or undirected.

tree:

Pointer to an uninitialized graph object, the result is stored here.

mode:

For directed graphs; whether to follow paths along edge directions (`IGRAPH_OUT`), or the opposite (`IGRAPH_IN`), or ignore edge directions completely (`IGRAPH_ALL`). It is ignored for undirected graphs.

roots:

A numeric vector giving the root vertex, or vertices (if the graph is not connected), to start from.

vertex_index:

Pointer to an initialized vector, or a null pointer. If not a null pointer, then a mapping from the vertices in the new graph to the ones in the original is created here.

Returns:

Error code.

Time complexity: O(n+m), linear in the number vertices and edges.

13.22. Other Operations

13.22.1. `igraph_density` — Calculate the density of a graph.

```
int igraph_density(const igraph_t *graph, igraph_real_t *res,
     igraph_bool_t loops);
```

The density of a graph is simply the ratio number of edges and the number of possible edges. Note that density is ill-defined for graphs with multiple and/or loop edges, so consider calling `igraph_simplify()` on the graph if you know that it contains multiple or loop edges.

Arguments:

graph:

The input graph object.

res:

Pointer to a real number, the result will be stored here.

loops:

Logical constant, whether to include loops in the calculation. If this constant is TRUE then loop edges are thought to be possible in the graph (this does not necessarily mean that the graph really contains any loops). If this is FALSE then the result is only correct if the graph does not contain loops.

Returns:

Error code.

Time complexity: O(1).

13.22.2. `igraph_reciprocity` — Calculates the reciprocity of a directed graph.

```
int igraph_reciprocity(const igraph_t *graph, igraph_real_t *res,
         igraph_bool_t ignore_loops,
         igraph_reciprocity_t mode);
```

The measure of reciprocity defines the proportion of mutual connections, in a directed graph. It is most commonly defined as the probability that the opposite counterpart of a directed edge is also included in the graph. In adjacency matrix notation: `sum(i, j, (A.*A')ij) / sum(i, j, Aij)` , where `A.*A'` is the element-wise product of matrix `A` and its transpose. This measure is calculated if the *mode* argument is `IGRAPH_RECIPROCITY_DEFAULT`.

Prior to igraph version 0.6, another measure was implemented, defined as the probability of mutual connection between a vertex pair if we know that there is a (possibly non-mutual) connection between them. In other words, (unordered) vertex pairs are classified into three groups: (1) disconnected, (2) non-reciprocally connected, (3) reciprocally connected. The result is the size of group (3), divided by the sum of group sizes (2)+(3). This measure is calculated if *mode* is `IGRAPH_RECIPROCITY_RATIO`.

Arguments:

graph:

The graph object.

res:

Pointer to an `igraph_real_t` which will contain the result.

ignore_loops:

Whether to ignore loop edges.

mode:

Type of reciprocity to calculate, possible values are `IGRAPH_RECIPROCITY_DEFAULT` and `IGRAPH_RECIPROCITY_RATIO`, please see their description above.

Returns:

Error code: `IGRAPH_EINVAL`: graph has no edges `IGRAPH_ENOMEM`: not enough memory for temporary data.

Time complexity: O(|V|+|E|), |V| is the number of vertices, |E| is the number of edges.

Example 13-44. File examples/simple/igraph_reciprocity.c

13.22.3. `igraph_diversity` — Structural diversity index of the vertices

```
int igraph_diversity(igraph_t *graph, const igraph_vector_t *weights,
       igraph_vector_t *res, const igraph_vs_t vids);
```

This measure was defined in Nathan Eagle, Michael Macy and Rob Claxton: Network Diversity and Economic Development, Science 328, 1029--1031, 2010.

It is simply the (normalized) Shannon entropy of the incident edges' weights. D(i)=H(i)/log(k[i]), and H(i) = -sum(p[i,j] log(p[i,j]), j=1..k[i]), where p[i,j]=w[i,j]/sum(w[i,l], l=1..k[i]), k[i] is the (total) degree of vertex i, and w[i,j] is the weight of the edge(s) between vertex i and j.

Arguments:

`The`:

input graph, edge directions are ignored.

`weights`:

The edge weights, in the order of the edge ids, must have appropriate length.

`res`:

An initialized vector, the results are stored here.

`vids`:

Vector with the vertex ids for which to calculate the measure.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear.

13.22.4. `igraph_is_mutual` — Check whether the edges of a directed graph are mutual.

```
int igraph_is_mutual(igraph_t *graph, igraph_vector_bool_t *res, igraph_es_t es);
```

An (A,B) edge is mutual if the graph contains the (B,A) edge, too.

An undirected graph only has mutual edges, by definition.

Edge multiplicity is not considered here, e.g. if there are two (A,B) edges and one (B,A) edge, then all three are considered to be mutual.

Arguments:

graph:

The input graph.

res:

Pointer to an initialized vector, the result is stored here.

es:

The sequence of edges to check. Supply `igraph_ess_all()` for all edges, see `igraph_ess_all()`.

Returns:

Error code.

Time complexity: O(n log(d)), n is the number of edges supplied, d is the maximum in-degree of the vertices that are targets of the supplied edges. An upper limit of the time complexity is O(n log(|E|)), |E| is the number of edges in the graph.

13.22.5. `igraph_avg_nearest_neighbor_degree` — Average nearest neighbor degree.

```
int igraph_avg_nearest_neighbor_degree(const igraph_t *graph,
            igraph_vs_t vids,
            igraph_vector_t *knn,
            igraph_vector_t *knnk,
            const igraph_vector_t *weights);
```

Calculates the average degree of the neighbors for each vertex, and optionally, the same quantity in the function of vertex degree.

For isolate vertices *knn* is set to `IGRAPH_NAN`. The same is done in *knnk* for vertex degrees that don't appear in the graph.

Arguments:

graph:

The input graph, it can be directed but the directedness of the edges is ignored.

vids:

The vertices for which the calculation is performed.

knn:

Pointer to an initialized vector, the result will be stored here. It will be resized as needed. Supply a NULL pointer here, if you only want to calculate `knnk`.

knnk:

Pointer to an initialized vector, the average nearest neighbor degree in the function of vertex degree is stored here. The first (zeroth) element is for degree one vertices, etc. Supply a NULL pointer here if you don't want to calculate this.

weights:

Optional edge weights. Supply a null pointer here for the non-weighted version. If this is not a null pointer, then the strength of the vertices is used instead of the normal vertex degree, see `igraph_strength()`.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

Example 13-45. File examples/simple/igraph_knn.c

13.22.6. `igraph_get_adjacency` — Returns the adjacency matrix of a graph

```
int igraph_get_adjacency(const igraph_t *graph, igraph_matrix_t *res,
    igraph_get_adjacency_t type, igraph_bool_t eids);
```

The result is an incidence matrix, it contains numbers greater than one if there are multiple edges in the graph.

Arguments:

graph:

Pointer to the graph to convert

res:

Pointer to an initialized matrix object, it will be resized if needed.

type:

Constant giving the type of the adjacency matrix to create for undirected graphs. It is ignored for directed graphs. Possible values:

`IGRAPH_GET_ADJACENCY_UPPER`

the upper right triangle of the matrix is used.

`IGRAPH_GET_ADJACENCY_LOWER`

the lower left triangle of the matrix is used.

`IGRAPH_GET_ADJACENCY_BOTH`

the whole matrix is used, a symmetric matrix is returned.

type:

eids Logical, if true, then the edges ids plus one are stored in the adjacency matrix, instead of the number of edges between the two vertices. (The plus one is needed, since edge ids start from zero, and zero means no edge in this case.)

Returns:

Error code: `IGRAPH_EINVAL` invalid type argument.

See also:

igraph_get_adjacency_sparse if you want a sparse matrix representation

Time complexity: O(|V||V|), |V| is the number of vertices in the graph.

13.22.7. `igraph_get_stochastic` — Stochastic adjacency matrix of a graph

```
int igraph_get_stochastic(const igraph_t *graph,
     igraph_matrix_t *matrix,
     igraph_bool_t column_wise);
```

Stochastic matrix of a graph. The stochastic matrix of a graph is its adjacency matrix, normalized row-wise or column-wise, such that the sum of each row (or column) is one.

Arguments:

graph:

The input graph.

sparsemat:

Pointer to an initialized matrix, the result is stored here.

column_wise:

Whether to normalize column-wise. For undirected graphs this argument does not have any effect.

Returns:

Error code.

Time complexity: O(|V||V|), quadratic in the number of vertices.

See also:

igraph_get_stochastic_sparsemat(), the sparse version of this function.

13.22.8. `igraph_get_stochastic_sparsemat` — Stochastic adjacency matrix of a graph

```
int igraph_get_stochastic_sparsemat(const igraph_t *graph,
        igraph_sparsemat_t *sparsemat,
        igraph_bool_t column_wise);
```

Stochastic matrix of a graph. The stochastic matrix of a graph is its adjacency matrix, normalized row-wise or column-wise, such that the sum of each row (or column) is one.

Arguments:

graph:

The input graph.

sparsemat:

Pointer to an uninitialized sparse matrix, the result is stored here.

column_wise:

Whether to normalize column-wise. For undirected graphs this argument does not have any effect.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

See also:

igraph_get_stochastic(), the dense version of this function.

13.22.9. `igraph_get_edgelist` — Returns the list of edges in a graph

```
int igraph_get_edgelist(const igraph_t *graph, igraph_vector_t *res, igraph_bool_t bycol);
```

The order of the edges is given by the edge ids.

Arguments:

graph:

Pointer to the graph object

res:

Pointer to an initialized vector object, it will be resized.

bycol:

Logical, if true, the edges will be returned columnwise, eg. the first edge is `res[0]->res[|E|]`, the second is `res[1]->res[|E|+1]`, etc.

Returns:

Error code.

Time complexity: O(|E|), the number of edges in the graph.

13.22.10. `igraph_contract_vertices` — Replace multiple vertices with a single one.

```
int igraph_contract_vertices(igraph_t *graph,
        const igraph_vector_t *mapping,
        const igraph_attribute_combination_t
        *vertex_comb);
```

This function creates a new graph, by merging several vertices into one. The vertices in the new graph correspond to sets of vertices in the input graph.

Arguments:

graph:

The input graph, it can be directed or undirected.

mapping:

A vector giving the mapping. For each vertex in the original graph, it should contain its id in the new graph.

vertex_comb:

What to do with the vertex attributes. See the igraph manual section about attributes for details.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number or vertices plus edges.

Chapter 14. Graph visitors

14.1. Breadth-first search

14.1.1. `igraph_bfs` — Breadth-first search

```
int igraph_bfs(const igraph_t *graph,
        igraph_integer_t root, const igraph_vector_t *roots,
        igraph_neimode_t mode, igraph_bool_t unreachable,
        const igraph_vector_t *restricted,
        igraph_vector_t *order, igraph_vector_t *rank,
        igraph_vector_t *father,
        igraph_vector_t *pred, igraph_vector_t *succ,
        igraph_vector_t *dist, igraph_bfshandler_t *callback,
        void *extra);
```

A simple breadth-first search, with a lot of different results and the possibility to call a callback whenever a vertex is visited. It is allowed to supply null pointers as the output arguments the user is not interested in, in this case they will be ignored.

If not all vertices can be reached from the supplied root vertex, then additional root vertices will be used, in the order of their vertex ids.

Arguments:

`graph`:

The input graph.

`root`:

The id of the root vertex. It is ignored if the `roots` argument is not a null pointer.

`roots`:

Pointer to an initialized vector, or a null pointer. If not a null pointer, then it is a vector containing root vertices to start the BFS from. The vertices are considered in the order they appear. If a root vertex was already found while searching from another one, then no search is conducted from it.

`mode`:

For directed graphs, it defines which edges to follow. `IGRAPH_OUT` means following the direction of the edges, `IGRAPH_IN` means the opposite, and `IGRAPH_ALL` ignores the direction of the edges. This parameter is ignored for undirected graphs.

unreachable:

Logical scalar, whether the search should visit the vertices that are unreachable from the given root node(s). If true, then additional searches are performed until all vertices are visited.

restricted:

If not a null pointer, then it must be a pointer to a vector containing vertex ids. The BFS is carried out only on these vertices.

order:

If not null pointer, then the vertex ids of the graph are stored here, in the same order as they were visited.

rank:

If not a null pointer, then the rank of each vertex is stored here.

father:

If not a null pointer, then the id of the father of each vertex is stored here.

pred:

If not a null pointer, then the id of vertex that was visited before the current one is stored here. If there is no such vertex (the current vertex is the root of a search tree), then -1 is stored.

succ:

If not a null pointer, then the id of the vertex that was visited after the current one is stored here. If there is no such vertex (the current one is the last in a search tree), then -1 is stored.

dist:

If not a null pointer, then the distance from the root of the current search tree is stored here.

callback:

If not null, then it should be a pointer to a function of type `igraph_bfshandler_t`. This function will be called, whenever a new vertex is visited.

extra:

Extra argument to pass to the callback function.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

Example 14-1. File examples/simple/igraph_bfs.c

Example 14-2. File examples/simple/igraph_bfs2.c

14.1.2. `igraph_bfshandler_t` — Callback type for BFS function

```
typedef igraph_bool_t igraph_bfshandler_t(const igraph_t *graph,
        igraph_integer_t vid,
        igraph_integer_t pred,
        igraph_integer_t succ,
        igraph_integer_t rank,
        igraph_integer_t dist,
        void *extra);
```

`igraph_bfs()` is able to call a callback function, whenever a new vertex is found, while doing the breadth-first search. This callback function must be of type `igraph_bfshandler_t`. It has the following arguments:

Arguments:

`graph`:

The graph that that algorithm is working on. Of course this must not be modified.

`vid`:

The id of the vertex just found by the breadth-first search.

`pred`:

The id of the previous vertex visited. It is -1 if there is no previous vertex, because the current vertex is the root is a search tree.

`succ`:

The id of the next vertex that will be visited. It is -1 if there is no next vertex, because the current vertex is the last one in a search tree.

rank:

The rank of the current vertex, it starts with zero.

dist:

The distance (number of hops) of the current vertex from the root of the current search tree.

extra:

The extra argument that was passed to `igraph_bfs()`.

Returns:

A logical value, if TRUE (=non-zero), that is interpreted as a request to stop the BFS and return to the caller. If a BFS is terminated like this, then all elements of the result vectors that were not yet calculated at the point of the termination contain `IGRAPH_NAN`.

See also:

`igraph_bfs()`

14.2. Depth-first search

14.2.1. `igraph_dfs` — Depth-first search

```
int igraph_dfs(const igraph_t *graph, igraph_integer_t root,
        igraph_neimode_t mode, igraph_bool_t unreachable,
        igraph_vector_t *order,
        igraph_vector_t *order_out, igraph_vector_t *father,
        igraph_vector_t *dist, igraph_dfshandler_t *in_callback,
        igraph_dfshandler_t *out_callback,
        void *extra);
```

A simple depth-first search, with the possibility to call a callback whenever a vertex is discovered and/or whenever a subtree is finished. It is allowed to supply null pointers as the output arguments the user is not interested in, in this case they will be ignored.

If not all vertices can be reached from the supplied root vertex, then additional root vertices will be used, in the order of their vertex ids.

Arguments:

graph:

The input graph.

root:

The id of the root vertex.

mode:

For directed graphs, it defines which edges to follow. `IGRAPH_OUT` means following the direction of the edges, `IGRAPH_IN` means the opposite, and `IGRAPH_ALL` ignores the direction of the edges. This parameter is ignored for undirected graphs.

unreachable:

Logical scalar, whether the search should visit the vertices that are unreachable from the given root node(s). If true, then additional searches are performed until all vertices are visited.

order:

If not null pointer, then the vertex ids of the graph are stored here, in the same order as they were discovered.

order_out:

If not a null pointer, then the vertex ids of the graphs are stored here, in the order of the completion of their subtree.

father:

If not a null pointer, then the id of the father of each vertex is stored here.

dist:

If not a null pointer, then the distance from the root of the current search tree is stored here.

in_callback:

If not null, then it should be a pointer to a function of type `igraph_dfshandler_t`. This function will be called, whenever a new vertex is discovered.

out_callback:

If not null, then it should be a pointer to a function of type `igraph_dfshandler_t`. This function will be called, whenever the subtree of a vertex is completed.

extra:

Extra argument to pass to the callback function(s).

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

14.2.2. `igraph_dfshandler_t` — Callback type for the DFS function

```
typedef igraph_bool_t igraph_dfshandler_t(const igraph_t *graph,
        igraph_integer_t vid,
        igraph_integer_t dist,
        void *extra);
```

`igraph_dfs()` is able to call a callback function, whenever a new vertex is discovered, and/or whenever a subtree is completed. These callbacks must be of type `igraph_dfshandler_t`. They have the following arguments:

Arguments:

`graph`:

The graph that that algorithm is working on. Of course this must not be modified.

`vid`:

The id of the vertex just found by the depth-first search.

`dist`:

The distance (number of hops) of the current vertex from the root of the current search tree.

`extra`:

The extra argument that was passed to `igraph_dfs()`.

Returns:

A logical value, if TRUE (=non-zero), that is interpreted as a request to stop the DFS and return to the caller. If a DFS is terminated like this, then all elements of the result vectors that were not yet calculated at the point of the termination contain `IGRAPH_NAN`.

See also:

`igraph_dfs()`

Chapter 15. Cliques and Independent Vertex Sets

These functions calculate various graph properties related to cliques and independent vertex sets.

15.1. Cliques

15.1.1. `igraph_cliques` — Find all or some cliques in a graph

```
int igraph_cliques(const igraph_t *graph, igraph_vector_ptr_t *res,
                   igraph_integer_t min_size, igraph_integer_t max_size);
```

Cliques are fully connected subgraphs of a graph.

If you are only interested in the size of the largest clique in the graph, use `igraph_clique_number()` instead.

The current implementation of this function searches for maximal independent vertex sets (see `igraph_maximal_independent_vertex_sets()`) in the complementer graph using the algorithm published in: S. Tsukiyama, M. Ide, H. Ariyoshi and I. Shirawaka. A new algorithm for generating all the maximal independent sets. SIAM J Computing, 6:505--517, 1977.

Arguments:

`graph`:

The input graph.

`res`:

Pointer to a pointer vector, the result will be stored here, ie. `res` will contain pointers to `igraph_vector_t` objects which contain the indices of vertices involved in a clique. The pointer vector will be resized if needed but note that the objects in the pointer vector will not be freed.

`min_size`:

Integer giving the minimum size of the cliques to be returned. If negative or zero, no lower bound will be used.

max_size:

Integer giving the maximum size of the cliques to be returned. If negative or zero, no upper bound will be used.

Returns:

Error code.

See also:

`igraph_largest_cliques()` and `igraph_clique_number()`.

Time complexity: TODO

Example 15-1. File examples/simple/igraph_cliques.c

15.1.2. `igraph_largest_cliques` — Finds the largest clique(s) in a graph.

```
int igraph_largest_cliques(const igraph_t *graph, igraph_vector_ptr_t *res);
```

A clique is largest (quite intuitively) if there is no other clique in the graph which contains more vertices.

Note that this is not necessarily the same as a maximal clique, ie. the largest cliques are always maximal but a maximal clique is not always largest.

The current implementation of this function searches for maximal cliques using `igraph_maximal_cliques()` and drops those that are not the largest.

The implementation of this function changed between igraph 0.5 and 0.6, so the order of the cliques and the order of vertices within the cliques will almost surely be different between these two versions.

Arguments:

`graph`:

The input graph.

`res`:

Pointer to an initialized pointer vector, the result will be stored here. It will be resized as needed. Note that vertices of a clique may be returned in arbitrary order.

Returns:

Error code.

See also:

`igraph_cliques()`, `igraph_maximal_cliques()`

Time complexity: O(3^(|V|/3)) worst case.

15.1.3. `igraph_maximal_cliques` — Find all maximal cliques of a graph

```
int igraph_maximal_cliques(const igraph_t *graph, igraph_vector_ptr_t *res,
        igraph_integer_t min_size, igraph_integer_t max_size);
```

A maximal clique is a clique which can't be extended any more by adding a new vertex to it.

If you are only interested in the size of the largest clique in the graph, use `igraph_clique_number()` instead.

The current implementation uses the Bron-Kerbosch algorithm to find the maximal cliques, see: C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. Communications of the ACM 16(9):575-577, 1973.

The implementation of this function changed between igraph 0.5 and 0.6, so the order of the cliques and the order of vertices within the cliques will almost surely be different between these two versions.

Arguments:

`graph`:

The input graph.

`res`:

Pointer to a pointer vector, the result will be stored here, ie. `res` will contain pointers to `igraph_vector_t` objects which contain the indices of vertices involved in a clique. The pointer vector will be resized if needed but note that the objects in the pointer vector will not be freed. Note that vertices of a clique may be returned in arbitrary order.

`min_size`:

Integer giving the minimum size of the cliques to be returned. If negative or zero, no lower bound will be used.

`max_size`:

Integer giving the maximum size of the cliques to be returned. If negative or zero, no upper bound will be used.

Returns:

Error code.

See also:

`igraph_maximal_independent_vertex_sets()`, `igraph_clique_number()`

Time complexity: O(3^(|V|/3)) worst case.

Example 15-2. File examples/simple/igraph_maximal_cliques.c

15.1.4. `igraph_clique_number` — Find the clique number of the graph

```
int igraph_clique_number(const igraph_t *graph, igraph_integer_t *no);
```

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

Arguments:

graph:

The input graph.

no:

The clique number will be returned to the `igraph_integer_t` pointed by this variable.

Returns:

Error code.

See also:

`igraph_cliques()`, `igraph_largest_cliques()`.

Time complexity: O(3^(|V|/3)) worst case.

15.2. Independent Vertex Sets

15.2.1. `igraph_independent_vertex_sets` — Find all independent vertex sets in a graph

```
int igraph_independent_vertex_sets(const igraph_t *graph,
       igraph_vector_ptr_t *res,
```

```
        igraph_integer_t min_size,
        igraph_integer_t max_size);
```

A vertex set is considered independent if there are no edges between them.

If you are interested in the size of the largest independent vertex set, use `igraph_independence_number()` instead.

The current implementation was ported to igraph from the Very Nauty Graph Library by Keith Briggs and uses the algorithm from the paper S. Tsukiyama, M. Ide, H. Ariyoshi and I. Shirawaka. A new algorithm for generating all the maximal independent sets. SIAM J Computing, 6:505--517, 1977.

Arguments:

graph:

The input graph.

res:

Pointer to a pointer vector, the result will be stored here, ie. `res` will contain pointers to `igraph_vector_t` objects which contain the indices of vertices involved in an independent vertex set. The pointer vector will be resized if needed but note that the objects in the pointer vector will not be freed.

min_size:

Integer giving the minimum size of the sets to be returned. If negative or zero, no lower bound will be used.

max_size:

Integer giving the maximum size of the sets to be returned. If negative or zero, no upper bound will be used.

Returns:

Error code.

See also:

`igraph_largest_independent_vertex_sets()`, `igraph_independence_number()`.

Time complexity: TODO

Example 15-3. File examples/simple/igraph_independent_sets.c

15.2.2. `igraph_largest_independent_vertex_sets` — Finds the largest independent vertex set(s) in a graph.

```
int igraph_largest_independent_vertex_sets(const igraph_t *graph,
        igraph_vector_ptr_t *res);
```

An independent vertex set is largest if there is no other independent vertex set with more vertices in the graph.

The current implementation was ported to igraph from the Very Nauty Graph Library by Keith Briggs and uses the algorithm from the paper S. Tsukiyama, M. Ide, H. Ariyoshi and I. Shirawaka. A new algorithm for generating all the maximal independent sets. SIAM J Computing, 6:505--517, 1977.

Arguments:

`graph`:

The input graph.

`res`:

Pointer to a pointer vector, the result will be stored here. It will be resized as needed.

Returns:

Error code.

See also:

`igraph_independent_vertex_sets()`, `igraph_maximal_independent_vertex_sets()`.

Time complexity: TODO

15.2.3. `igraph_maximal_independent_vertex_sets` — Find all maximal independent vertex sets of a graph

```
int igraph_maximal_independent_vertex_sets(const igraph_t *graph,
        igraph_vector_ptr_t *res);
```

A maximal independent vertex set is an independent vertex set which can't be extended any more by adding a new vertex to it.

The algorithm used here is based on the following paper: S. Tsukiyama, M. Ide, H. Ariyoshi and I. Shirawaka. A new algorithm for generating all the maximal independent sets. SIAM J Computing, 6:505--517, 1977.

The implementation was originally written by Kevin O'Neill and modified by K M Briggs in the Very Nauty Graph Library. I simply re-wrote it to use igraph's data structures.

If you are interested in the size of the largest independent vertex set, use `igraph_independence_number()` instead.

Arguments:

`graph`:

The input graph.

`res`:

Pointer to a pointer vector, the result will be stored here, ie. `res` will contain pointers to `igraph_vector_t` objects which contain the indices of vertices involved in an independent vertex set. The pointer vector will be resized if needed but note that the objects in the pointer vector will not be freed.

Returns:

Error code.

See also:

`igraph_maximal_cliques()`, `igraph_independence_number()`

Time complexity: TODO.

15.2.4. `igraph_independence_number` — Find the independence number of the graph

```
int igraph_independence_number(const igraph_t *graph, igraph_integer_t *no);
```

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

The current implementation was ported to igraph from the Very Nauty Graph Library by Keith Briggs and uses the algorithm from the paper S. Tsukiyama, M. Ide, H. Ariyoshi and I. Shirawaka. A new algorithm for generating all the maximal independent sets. SIAM J Computing, 6:505--517, 1977.

Arguments:

`graph`:

The input graph.

`no`:

The independence number will be returned to the `igraph_integer_t` pointed by this variable.

Returns:

Error code.

See also:

`igraph_independent_vertex_sets()`.

Time complexity: TODO.

Chapter 16. Graph Isomorphism

16.1. The simple interface

igraph provides four set of functions to deal with graph isomorphism problems.

The `igraph_isomorphic()` and `igraph_subisomorphic()` functions make up the first set (in addition with the `igraph_permute_vertices()` function). These functions choose the algorithm which is best for the supplied input graph. (The choice is not very sophisticated though, see their documentation for details.)

The VF2 graph (and subgraph) isomorphism algorithm is implemented in igraph, these functions are the second set. See `igraph_isomorphic_vf2()` and `igraph_subisomorphic_vf2()` for starters.

Functions for the BLISS algorithm constitute the third set, see `igraph_isomorphic_bliss()`. This implementation only works for undirected graphs.

Finally, the isomorphism classes of all graphs with three and four vertices are precomputed and stored in igraph, so for these small graphs there is a very simple fast way to decide isomorphism. See `igraph_isomorphic_34()`.

16.1.1. `igraph_permute_vertices` — Permute the vertices

```
int igraph_permute_vertices(const igraph_t *graph, igraph_t *res,
       const igraph_vector_t *permutation);
```

This function creates a new graph from the input graph by permuting its vertices according to the specified mapping. Call this function with the output of `igraph_canonical_permutation()` to create the canonical form of a graph.

Arguments:

graph:

The input graph.

res:

Pointer to an uninitialized graph object. The new graph is created here.

permutation:

The permutation to apply. Vertex 0 is mapped to the first element of the vector, vertex 1 to the second, etc. Note that it is not checked that the vector contains every element only once, and no range checking is performed either.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in terms of the number of vertices and edges.

16.1.2. `igraph_isomorphic` — Decides whether two graphs are isomorphic

```
int igraph_isomorphic(const igraph_t *graph1, const igraph_t *graph2,
        igraph_bool_t *iso);
```

From Wikipedia: The graph isomorphism problem or GI problem is the graph theory problem of determining whether, given two graphs G1 and G2, it is possible to permute (or relabel) the vertices of one graph so that it is equal to the other. Such a permutation is called a graph isomorphism.

This function decides which graph isomorphism algorithm to be used based on the input graphs. Right now it does the following:

1. If one graph is directed and the other undirected then an error is triggered.
2. If the two graphs does not have the same number of vertices and edges it returns with `FALSE`.
3. Otherwise, if the graphs have three or four vertices then an O(1) algorithm is used with precomputed data.
4. Otherwise, if the graphs are directed then VF2 is used, see `igraph_isomorphic_vf2()`.
5. Otherwise BLISS is used, see `igraph_isomorphic_bliss()`.

Please call the VF2 and BLISS functions directly if you need something more sophisticated, e.g. you need the isomorphic mapping.

Arguments:

graph1:

The first graph.

graph2:

The second graph.

iso:

Pointer to a logical variable, will be set to TRUE (1) if the two graphs are isomorphic, and FALSE (0) otherwise.

Returns:

Error code.

See also:

`igraph_isoclass()`, `igraph_isoclass_subgraph()`, `igraph_isoclass_create()`.

Time complexity: exponential.

16.1.3. `igraph_subisomorphic` — Decide subgraph isomorphism

```
int igraph_subisomorphic(const igraph_t *graph1, const igraph_t *graph2,
    igraph_bool_t *iso);
```

Check whether *graph2* is isomorphic to a subgraph of *graph1*. Currently this function just calls `igraph_subisomorphic_vf2()` for all graphs.

Arguments:

graph1:

The first input graph, may be directed or undirected. This is supposed to be the bigger graph.

graph2:

The second input graph, it must have the same directedness as *graph2*, or an error is triggered. This is supposed to be the smaller graph.

`iso`:

Pointer to a boolean, the result is stored here.

Returns:

Error code.

Time complexity: exponential.

16.2. The BLISS algorithm

BLISS is a successor of the famous NAUTY algorithm and implementation. While using the same ideas in general, with better heuristics and data structure BLISS outperforms NAUTY on most graphs.

BLISS was developed and implemented by Tommi Junttila and Petteri Kaski at Helsinki University of Technology, Finland. See Tommi Juntilla's homepage at http://www.tcs.hut.fi/~tjunttil/ and the publication at http://www.siam.org/proceedings/alenex/2007/alx07_013junttilat.pdf for more information.

BLISS version 0.35 is included in igraph.

16.2.1. `igraph_bliss_sh_t` — Splitting heuristics for BLISS

```
typedef enum { IGRAPH_BLISS_F=0, IGRAPH_BLISS_FL,
        IGRAPH_BLISS_FS, IGRAPH_BLISS_FM,
        IGRAPH_BLISS_FLM, IGRAPH_BLISS_FSM } igraph_bliss_sh_t;
```

Values:

`IGRAPH_BLISS_F`:

First non-singleton cell.

`IGRAPH_BLISS_FL`:

First largest non-singleton cell.

`IGRAPH_BLISS_FS`:

First smallest non-singleton cell.

`IGRAPH_BLISS_FM`:

First maximally non-trivially connected non-singleton cell.

`IGRAPH_BLISS_FLM`:

Largest maximally non-trivially connected non-singleton cell.

`IGRAPH_BLISS_FSM`:

Smallest maximally non-trivially connected non-singletion cell.

16.2.2. `igraph_bliss_info_t` — Information about a BLISS run

```
typedef struct igraph_bliss_info_t {
  unsigned long nof_nodes;
  unsigned long nof_leaf_nodes;
  unsigned long nof_bad_nodes;
  unsigned long nof_canupdates;
  unsigned long max_level;
  char *group_size;
} igraph_bliss_info_t;
```

Some secondary information found by the BLISS algorithm is stored here. It is useful if you wany to study the internal working of the algorithm.

Values:

`nof_nodes`:

The number of nodes in the search tree.

`nof_leaf_nodes`:

The number of leaf nodes in the search tree.

`nof_bad_nodes`:

Number of bad nodes.

`nof_canupdates`:

Number of canrep updates.

`max_level`:

Maximum level.

`group_size`:

The size of the automorphism group of the graph, given as a string. It should be deallocated via `free()` if not needed any more.

See http://www.tcs.hut.fi/Software/bliss/index.html for details about the algorithm and these parameters.

16.2.3. `igraph_canonical_permutation` — Canonical permutation using BLISS

```
int igraph_canonical_permutation(const igraph_t *graph, igraph_vector_t *labeling,
     igraph_bliss_sh_t sh, igraph_bliss_info_t *info);
```

This function computes the canonical permutation which transforms the graph into a canonical form by using the BLISS algorithm.

Arguments:

`graph`:

The input graph, it is treated as undirected and the multiple edges are ignored.

`labeling`:

Pointer to a vector, the result is stored here. The permutation takes vertex 0 to the first element of the vector, vertex 1 to the second, etc. The vector will be resized as needed.

`sh`:

The split heuristics to be used in BLISS. See `igraph_bliss_sh_t`.

`info`:

If not `NULL` then information on BLISS internals is stored here. See `igraph_bliss_info_t`.

Returns:

Error code.

Time complexity: exponential, in practice it is fast for many graphs.

16.2.4. `igraph_isomorphic_bliss` — Graph isomorphism via BLISS

```
int igraph_isomorphic_bliss(const igraph_t *graph1, const igraph_t *graph2,
       igraph_bool_t *iso, igraph_vector_t *map12,
       igraph_vector_t *map21,
       igraph_bliss_sh_t sh1, igraph_bliss_sh_t sh2,
       igraph_bliss_info_t *info1, igraph_bliss_info_t *info2);
```

This function uses the BLISS graph isomorphism algorithm, a successor of the famous NAUTY algorithm and implementation. BLISS is open source and licensed according to the GNU GPL. See http://www.tcs.hut.fi/Software/bliss/index.html for details. Currently the 0.35 version of BLISS is included in igraph.

Arguments:

graph1:

The first input graph, it is assumed to be undirected, directed graphs are treated as undirected too. The algorithm eliminates multiple edges from the graph first.

graph2:

The second input graph, it is assumed to be undirected, directed graphs are treated as undirected too. The algorithm eliminates multiple edges from the graph first.

iso:

Pointer to a boolean, the result is stored here.

map12:

A vector or `NULL` pointer. If not `NULL` then an isomorphic mapping from *graph1* to *graph2* is stored here. If the input graphs are not isomorphic then this vector is cleared, i.e. it will have length zero.

map21:

Similar to *map12*, but for the mapping from *graph2* to *graph1*.

sh1:

Splitting heuristics to be used for the first graph. See `igraph_bliss_sh_t`.

sh2:

Splitting heuristics to be used for the second graph. See `igraph_bliss_sh_t`.

info1:

If not `NULL`, information about the canonization of the first input graph is stored here. See `igraph_bliss_info_t` for details.

info2:

Same as *info1*, but for the second graph.

Returns:

Error code.

Time complexity: exponential, but in practice it is quite fast.

16.2.5. `igraph_automorphisms` — Number of automorphisms using BLISS

```
int igraph_automorphisms(const igraph_t *graph,
    igraph_bliss_sh_t sh, igraph_bliss_info_t *info);
```

The number of automorphisms of a graph is computed using BLISS. The result is returned as part of the *info* structure, in tag `group_size`. It is returned as a string, as it can be very high even for relatively small graphs. If the GNU MP library is used then this number is exact, otherwise a long double is used and it is only approximate. See also `igraph_bliss_info_t`.

Arguments:

graph:

The input graph, it is treated as undirected and the multiple edges are ignored.

sh:

The split heuristics to be used in BLISS. See `igraph_bliss_sh_t`.

info:

The result is stored here, in particular in the `group_size` tag of *info*.

Returns:

Error code.

Time complexity: exponential, in practice it is fast for many graphs.

16.3. The VF2 algorithm

16.3.1. `igraph_isomorphic_vf2` — Isomorphism via VF2

```
int igraph_isomorphic_vf2(const igraph_t *graph1, const igraph_t *graph2,
     const igraph_vector_int_t *vertex_color1,
     const igraph_vector_int_t *vertex_color2,
     const igraph_vector_int_t *edge_color1,
     const igraph_vector_int_t *edge_color2,
     igraph_bool_t *iso, igraph_vector_t *map12,
     igraph_vector_t *map21,
     igraph_isocompat_t *node_compat_fn,
     igraph_isocompat_t *edge_compat_fn,
     void *arg);
```

This function performs the VF2 algorithm via calling `igraph_isomorphic_function_vf2()`.

Note that this function cannot be used for deciding subgraph isomorphism, use `igraph_subisomorphic_vf2()` for that.

Arguments:

`graph1`:

The first graph, may be directed or undirected.

`graph2`:

The second graph. It must have the same directedness as *`graph1`*, otherwise an error is reported.

`vertex_color1`:

An optional color vector for the first graph. If color vectors are given for both graphs, then the isomorphism is calculated on the colored graphs; i.e. two vertices can match only if their color also matches. Supply a null pointer here if your graphs are not colored.

vertex_color2:

An optional color vector for the second graph. See the previous argument for explanation.

edge_color1:

An optional edge color vector for the first graph. The matching edges in the two graphs must have matching colors as well. Supply a null pointer here if your graphs are not edge-colored.

edge_color2:

The edge color vector for the second graph.

iso:

Pointer to a logical constant, the result of the algorithm will be placed here.

map12:

Pointer to an initialized vector or a NULL pointer. If not a NULL pointer then the mapping from *graph1* to *graph2* is stored here. If the graphs are not isomorphic then the vector is cleared (ie. has zero elements).

map21:

Pointer to an initialized vector or a NULL pointer. If not a NULL pointer then the mapping from *graph2* to *graph1* is stored here. If the graphs are not isomorphic then the vector is cleared (ie. has zero elements).

node_compat_fn:

A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two nodes are compatible.

edge_compat_fn:

A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two edges are compatible.

arg:

Extra argument to supply to functions *node_compat_fn* and *edge_compat_fn*.

Returns:

Error code.

See also:

`igraph_subisomorphic_vf2()`, `igraph_count_isomorphisms_vf2()`, `igraph_get_isomorphisms_vf2()`, Time complexity: exponential, what did you expect?

Example 16-1. File examples/simple/igraph_isomorphic_vf2.c

16.3.2. `igraph_count_isomorphisms_vf2` — Number of isomorphisms via VF2

```
int igraph_count_isomorphisms_vf2(const igraph_t *graph1, const igraph_t *graph2,
      const igraph_vector_int_t *vertex_color1,
      const igraph_vector_int_t *vertex_color2,
      const igraph_vector_int_t *edge_color1,
      const igraph_vector_int_t *edge_color2,
      igraph_integer_t *count,
      igraph_isocompat_t *node_compat_fn,
      igraph_isocompat_t *edge_compat_fn,
      void *arg);
```

This function counts the number of isomorphic mappings between two graphs. It uses the generic `igraph_isomorphic_function_vf2()` function.

Arguments:

`graph1`:

The first input graph, may be directed or undirected.

`graph2`:

The second input graph, it must have the same directedness as *`graph1`*, or an error will be reported.

`vertex_color1`:

An optional color vector for the first graph. If color vectors are given for both graphs, then the isomorphism is calculated on the colored graphs; i.e. two vertices can match only if their color also matches. Supply a null pointer here if your graphs are not colored.

`vertex_color2`:

An optional color vector for the second graph. See the previous argument for explanation.

edge_color1:

An optional edge color vector for the first graph. The matching edges in the two graphs must have matching colors as well. Supply a null pointer here if your graphs are not edge-colored.

edge_color2:

The edge color vector for the second graph.

count:

Point to an integer, the result will be stored here.

node_compat_fn:

A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two nodes are compatible.

edge_compat_fn:

A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two edges are compatible.

arg:

Extra argument to supply to functions *node_compat_fn* and *edge_compat_fn*.

Returns:

Error code.

Time complexity: exponential.

16.3.3. `igraph_get_isomorphisms_vf2` — Collect the isomorphic mappings

```
int igraph_get_isomorphisms_vf2(const igraph_t *graph1,
    const igraph_t *graph2,
    const igraph_vector_int_t *vertex_color1,
    const igraph_vector_int_t *vertex_color2,
    const igraph_vector_int_t *edge_color1,
    const igraph_vector_int_t *edge_color2,
    igraph_vector_ptr_t *maps,
    igraph_isocompat_t *node_compat_fn,
    igraph_isocompat_t *edge_compat_fn,
    void *arg);
```

This function finds all the isomorphic mappings between two graphs. It uses the `igraph_isomorphic_function_vf2()` function. Call the function with the same graph as *graph1* and *graph2* to get automorphisms.

Arguments:

graph1:

The first input graph, may be directed or undirected.

graph2:

The second input graph, it must have the same directedness as *graph1*, or an error will be reported.

vertex_color1:

An optional color vector for the first graph. If color vectors are given for both graphs, then the isomorphism is calculated on the colored graphs; i.e. two vertices can match only if their color also matches. Supply a null pointer here if your graphs are not colored.

vertex_color2:

An optional color vector for the second graph. See the previous argument for explanation.

edge_color1:

An optional edge color vector for the first graph. The matching edges in the two graphs must have matching colors as well. Supply a null pointer here if your graphs are not edge-colored.

edge_color2:

The edge color vector for the second graph.

maps:

Pointer vector. On return it is empty if the input graphs are no isomorphic. Otherwise it contains pointers to igraph_vector_t objects, each vector is an isomorphic mapping of *graph2* to *graph1*. Please note that you need to 1) Destroy the vectors via `igraph_vector_destroy()`, 2) free them via `free()` and then 3) call `igraph_vector_ptr_destroy()` on the pointer vector to deallocate all memory when *maps* is no longer needed.

node_compat_fn:

A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two nodes are compatible.

edge_compat_fn:

A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two edges are compatible.

arg:

Extra argument to supply to functions *node_compat_fn* and *edge_compat_fn*.

Returns:

Error code.

Time complexity: exponential.

16.3.4. `igraph_isohandler_t` — Callback type, called when an isomorphism was found

```
typedef igraph_bool_t igraph_isohandler_t(const igraph_vector_t *map12,
       const igraph_vector_t *map21, void *arg);
```

See the details at the documentation of `igraph_isomorphic_function_vf2()`.

Arguments:

map12:

The mapping from the first graph to the second.

map21:

The mapping from the second graph to the first, the inverse of *map12* basically.

arg:

This extra argument was passed to `igraph_isomorphic_function_vf2()` when it was called.

Returns:

Boolean, whether to continue with the isomorphism search.

16.3.5. `igraph_isocompat_t` — Callback type, called to check whether two vertices or edges are compatible

```
typedef igraph_bool_t igraph_isocompat_t(const igraph_t *graph1,
      const igraph_t *graph2,
      const igraph_integer_t g1_num,
```

```
const igraph_integer_t g2_num,
void *arg);
```

VF2 (subgraph) isomorphism functions can be restricted by defining relations on the vertices and/or edges of the graphs, and then checking whether the vertices (edges) match according to these relations.

This feature is implemented by two callbacks, one for vertices, one for edges. Every time igraph tries to match a vertex (edge) of the first (sub)graph to a vertex of the second graph, the vertex (edge) compatibility callback is called. The callback returns a logical value, giving whether the two vertices match.

Both callback functions are of type `igraph_isocompat_t`.

Arguments:

graph1:

The first graph.

graph2:

The second graph.

g1_num:

The id of a vertex or edge in the first graph.

g2_num:

The id of a vertex or edge in the second graph.

arg:

Extra argument to pass to the callback functions.

Returns:

Logical scalar, whether vertex (or edge) *g1_num* in *graph1* is compatible with vertex (or edge) *g2_num* in *graph2*.

16.3.6. `igraph_isomorphic_function_vf2` — The generic VF2 interface

```
int igraph_isomorphic_function_vf2(const igraph_t *graph1, const igraph_t *graph2,
       const igraph_vector_int_t *vertex_color1,
       const igraph_vector_int_t *vertex_color2,
       const igraph_vector_int_t *edge_color1,
       const igraph_vector_int_t *edge_color2,
       igraph_vector_t *map12,
       igraph_vector_t *map21,
       igraph_isohandler_t *isohandler_fn,
       igraph_isocompat_t *node_compat_fn,
       igraph_isocompat_t *edge_compat_fn,
       void *arg);
```

This function is an implementation of the VF2 isomorphism algorithm, see P. Foggia, C. Sansone, M. Vento, An Improved algorithm for matching large graphs, Proc. of the 3rd IAPR-TC-15 International Workshop on Graph-based Representations, Italy, 2001.

For using it you need to define a callback function of type `igraph_isohandler_t`. This function will be called whenever VF2 finds an isomorphism between the two graphs. The mapping between the two graphs will be also provided to this function. If the callback returns a nonzero value then the search is continued, otherwise it stops.

Arguments:

graph1:

The first input graph.

graph2:

The second input graph.

vertex_color1:

An optional color vector for the first graph. If color vectors are given for both graphs, then the isomorphism is calculated on the colored graphs; i.e. two vertices can match only if their color also matches. Supply a null pointer here if your graphs are not colored.

vertex_color2:

An optional color vector for the second graph. See the previous argument for explanation.

edge_color1:

An optional edge color vector for the first graph. The matching edges in the two graphs must have matching colors as well. Supply a null pointer here if your graphs are not edge-colored.

edge_color2:

The edge color vector for the second graph.

map12:

Pointer to an initialized vector or `NULL`. If not `NULL` and the supplied graphs are isomorphic then the permutation taking *graph1* to *graph* is stored here. If not `NULL` and the graphs are not isomorphic then a zero-length vector is returned.

map21:

This is the same as *map12*, but for the permutation taking *graph2* to *graph1*.

isohandler_fn:

The callback function to be called if an isomorphism is found. See also `igraph_isohandler_t`.

node_compat_fn:

A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two nodes are compatible.

edge_compat_fn:

A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two edges are compatible.

arg:

Extra argument to supply to functions *isohandler_fn*, *node_compat_fn* and *edge_compat_fn*.

Returns:

Error code.

Time complexity: exponential.

16.3.7. `igraph_subisomorphic_vf2` — Decide subgraph isomorphism using VF2

```
int igraph_subisomorphic_vf2(const igraph_t *graph1, const igraph_t *graph2,
        const igraph_vector_int_t *vertex_color1,
```

```
const igraph_vector_int_t *vertex_color2,
const igraph_vector_int_t *edge_color1,
const igraph_vector_int_t *edge_color2,
igraph_bool_t *iso, igraph_vector_t *map12,
igraph_vector_t *map21,
igraph_isocompat_t *node_compat_fn,
igraph_isocompat_t *edge_compat_fn,
void *arg);
```

Decides whether a subgraph of *graph1* is isomorphic to *graph2*. It uses `igraph_subisomorphic_function_vf2()`.

Arguments:

graph1:

The first input graph, may be directed or undirected. This is supposed to be the larger graph.

graph2:

The second input graph, it must have the same directedness as *graph1*. This is supposed to be the smaller graph.

vertex_color1:

An optional color vector for the first graph. If color vectors are given for both graphs, then the subgraph isomorphism is calculated on the colored graphs; i.e. two vertices can match only if their color also matches. Supply a null pointer here if your graphs are not colored.

vertex_color2:

An optional color vector for the second graph. See the previous argument for explanation.

edge_color1:

An optional edge color vector for the first graph. The matching edges in the two graphs must have matching colors as well. Supply a null pointer here if your graphs are not edge-colored.

edge_color2:

The edge color vector for the second graph.

iso:

Pointer to a boolean. The result of the decision problem is stored here.

map12:

Pointer to a vector or `NULL`. If not `NULL`, then an isomorphic mapping from *graph1* to *graph2* is stored here.

map21:

Pointer to a vector ot `NULL`. If not `NULL`, then an isomorphic mapping from *graph2* to *graph1* is stored here.

node_compat_fn:

A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two nodes are compatible.

edge_compat_fn:

A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two edges are compatible.

arg:

Extra argument to supply to functions *node_compat_fn* and *edge_compat_fn*.

Returns:

Error code.

Time complexity: exponential.

16.3.8. `igraph_count_subisomorphisms_vf2` — Number of subgraph isomorphisms using VF2

```
int igraph_count_subisomorphisms_vf2(const igraph_t *graph1, const igraph_t *graph2,
         const igraph_vector_int_t *vertex_color1,
         const igraph_vector_int_t *vertex_color2,
         const igraph_vector_int_t *edge_color1,
         const igraph_vector_int_t *edge_color2,
         igraph_integer_t *count,
         igraph_isocompat_t *node_compat_fn,
         igraph_isocompat_t *edge_compat_fn,
         void *arg);
```

Count the number of isomorphisms between subgraphs of *graph1* and *graph2*. This function uses `igraph_subisomorphic_function_vf2()`.

Arguments:

graph1:

The first input graph, may be directed or undirected. This is supposed to be the larger graph.

graph2:

The second input graph, it must have the same directedness as *graph1*. This is supposed to be the smaller graph.

vertex_color1:

An optional color vector for the first graph. If color vectors are given for both graphs, then the subgraph isomorphism is calculated on the colored graphs; i.e. two vertices can match only if their color also matches. Supply a null pointer here if your graphs are not colored.

vertex_color2:

An optional color vector for the second graph. See the previous argument for explanation.

edge_color1:

An optional edge color vector for the first graph. The matching edges in the two graphs must have matching colors as well. Supply a null pointer here if your graphs are not edge-colored.

edge_color2:

The edge color vector for the second graph.

count:

Pointer to an integer. The number of subgraph isomorphisms is stored here.

node_compat_fn:

A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two nodes are compatible.

edge_compat_fn:

A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two edges are compatible.

arg:

Extra argument to supply to functions *node_compat_fn* and *edge_compat_fn*.

Returns:

Error code.

Time complexity: exponential.

16.3.9. `igraph_get_subisomorphisms_vf2` — Return all subgraph isomorphic mappings

```
int igraph_get_subisomorphisms_vf2(const igraph_t *graph1,
       const igraph_t *graph2,
       const igraph_vector_int_t *vertex_color1,
       const igraph_vector_int_t *vertex_color2,
       const igraph_vector_int_t *edge_color1,
       const igraph_vector_int_t *edge_color2,
       igraph_vector_ptr_t *maps,
       igraph_isocompat_t *node_compat_fn,
       igraph_isocompat_t *edge_compat_fn,
       void *arg);
```

This function collects all isomorphic mappings of *graph2* to a subgraph of *graph1*. It uses the `igraph_subisomorphic_function_vf2()` function.

Arguments:

graph1:

The first input graph, may be directed or undirected. This is supposed to be the larger graph.

graph2:

The second input graph, it must have the same directedness as *graph1*. This is supposed to be the smaller graph.

vertex_color1:

An optional color vector for the first graph. If color vectors are given for both graphs, then the subgraph isomorphism is calculated on the colored graphs; i.e. two vertices can match only if their color also matches. Supply a null pointer here if your graphs are not colored.

vertex_color2:

An optional color vector for the second graph. See the previous argument for explanation.

edge_color1:

An optional edge color vector for the first graph. The matching edges in the two graphs must have matching colors as well. Supply a null pointer here if your graphs are not edge-colored.

edge_color2:

The edge color vector for the second graph.

maps:

Pointer vector. On return it contains pointers to igraph_vector_t objects, each vector is an isomorphic mapping of *graph2* to a subgraph of *graph1*. Please note that you need to 1) Destroy the vectors via `igraph_vector_destroy()`, 2) free them via `free()` and then 3) call `igraph_vector_ptr_destroy()` on the pointer vector to deallocate all memory when *maps* is no longer needed.

node_compat_fn:

A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two nodes are compatible.

edge_compat_fn:

A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two edges are compatible.

arg:

Extra argument to supply to functions *node_compat_fn* and *edge_compat_fn*.

Returns:

Error code.

Time complexity: exponential.

16.3.10. `igraph_subisomorphic_function_vf2` — Generic VF2 function for subgraph isomorphism problems

```
int igraph_subisomorphic_function_vf2(const igraph_t *graph1,
          const igraph_t *graph2,
          const igraph_vector_int_t *vertex_color1,
          const igraph_vector_int_t *vertex_color2,
          const igraph_vector_int_t *edge_color1,
          const igraph_vector_int_t *edge_color2,
          igraph_vector_t *map12,
          igraph_vector_t *map21,
          igraph_isohandler_t *isohandler_fn,
          igraph_isocompat_t *node_compat_fn,
          igraph_isocompat_t *edge_compat_fn,
          void *arg);
```

This function is the pair of `igraph_isomorphic_function_vf2()`, for subgraph isomorphism problems. It searches for subgraphs of *graph1* which are isomorphic to *graph2*. When it founds an isomorphic mapping it calls the supplied callback *isohandler_fn*. The mapping (and its inverse) and the additional *arg* argument are supplied to the callback.

Arguments:

graph1:

The first input graph, may be directed or undirected. This is supposed to be the larger graph.

graph2:

The second input graph, it must have the same directedness as *graph1*. This is supposed to be the smaller graph.

vertex_color1:

An optional color vector for the first graph. If color vectors are given for both graphs, then the subgraph isomorphism is calculated on the colored graphs; i.e. two vertices can match only if their color also matches. Supply a null pointer here if your graphs are not colored.

vertex_color2:

An optional color vector for the second graph. See the previous argument for explanation.

edge_color1:

An optional edge color vector for the first graph. The matching edges in the two graphs must have matching colors as well. Supply a null pointer here if your graphs are not edge-colored.

edge_color2:

The edge color vector for the second graph.

map12:

Pointer to a vector or `NULL`. If not `NULL`, then an isomorphic mapping from *graph1* to *graph2* is stored here.

map21:

Pointer to a vector ot `NULL`. If not `NULL`, then an isomorphic mapping from *graph2* to *graph1* is stored here.

isohandler_fn:

A pointer to a function of type `igraph_isohandler_t`. This will be called whenever a subgraph isomorphism is found. If the function returns with a non-zero value then the search is continued, otherwise it stops and the function returns.

node_compat_fn:

A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two nodes are compatible.

edge_compat_fn:

A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two edges are compatible.

arg:

Extra argument to supply to functions *isohandler_fn*, *node_compat_fn* and *edge_compat_fn*.

Returns:

Error code.

Time complexity: exponential.

16.4. Functions for graphs with 3 or 4 vertices

16.4.1. `igraph_isomorphic_34` — Graph isomorphism for 3-4 vertices

```
int igraph_isomorphic_34(const igraph_t *graph1, const igraph_t *graph2,
    igraph_bool_t *iso);
```

This function uses precomputed indices to decide isomorphism problems for graphs with only 3 or 4 vertices.

Arguments:

graph1:

The first input graph.

graph2:

The second input graph. Must have the same directedness as *graph1*.

iso:

Pointer to a boolean, the result is stored here.

Returns:

Error code.

Time complexity: O(1).

16.4.2. `igraph_isoclass` — Determine the isomorphism class of a graph with 3 or 4 vertices

```
int igraph_isoclass(const igraph_t *graph, igraph_integer_t *isoclass);
```

All graphs with a given number of vertices belong to a number of isomorphism classes, with every graph in a given class being isomorphic to each other.

This function gives the isomorphism class (a number) of a graph. Two graphs have the same isomorphism class if and only if they are isomorphic.

The first isomorphism class is numbered zero and it is the empty graph, the last isomorphism class is the full graph. The number of isomorphism class for directed graphs with three vertices is 16 (between 0 and 15), for undirected graph it is only 4. For graphs with four vertices it is 218 (directed) and 11 (undirected).

Arguments:

graph:

The graph object.

isoclass:

Pointer to an integer, the isomorphism class will be stored here.

Returns:

Error code.

See also:

`igraph_isomorphic()`, `igraph_isoclass_subgraph()`, `igraph_isoclass_create()`, `igraph_motifs_randesu()`.

Because of some limitations this function works only for graphs with three of four vertices.

Time complexity: O(|E|), the number of edges in the graph.

16.4.3. `igraph_isoclass_subgraph` — The isomorphism class of a subgraph of a graph.

```
int igraph_isoclass_subgraph(const igraph_t *graph, igraph_vector_t *vids,
        igraph_integer_t *isoclass);
```

This function is only implemented for subgraphs with three or four vertices.

Arguments:

`graph`:

The graph object.

`vids`:

A vector containing the vertex ids to be considered as a subgraph. Each vertex id should be included at most once.

`isoclass`:

Pointer to an integer, this will be set to the isomorphism class.

Returns:

Error code.

See also:

`igraph_isoclass()`, `igraph_isomorphic()`, `igraph_isoclass_create()`.

Time complexity: O((d+n)*n), d is the average degree in the network, and n is the number of vertices in `vids`.

16.4.4. `igraph_isoclass_create` — Creates a graph from the given isomorphism class.

```
int igraph_isoclass_create(igraph_t *graph, igraph_integer_t size,
      igraph_integer_t number, igraph_bool_t directed);
```

This function is implemented only for graphs with three or four vertices.

Arguments:

`graph`:

Pointer to an uninitialized graph object.

`size`:

The number of vertices to add to the graph.

`number`:

The isomorphism class.

`directed`:

Logical constant, whether to create a directed graph.

Returns:

Error code.

See also:

`igraph_isoclass()`, `igraph_isoclass_subgraph()`, `igraph_isomorphic()`.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph to create.

Chapter 17. Graph Motifs, Dyad Census and Triad Census

This section deals with functions which find small induced subgraphs in a graph. These were first defined for subgraphs of two and three vertices by Holland and Leinhardt, and named dyad census and triad census.

17.1. `igraph_dyad_census` — Calculating the dyad census as defined by Holland and Leinhardt

```
int igraph_dyad_census(const igraph_t *graph, igraph_integer_t *mut,
         igraph_integer_t *asym, igraph_integer_t *null);
```

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

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

Arguments:

`graph`:

The input graph, a warning is given if undirected as the results are undefined for undirected graphs.

`mut`:

Pointer to an integer, the number of mutual dyads is stored here.

`asym`:

Pointer to an integer, the number of asymmetric dyads is stored here.

`null`:

Pointer to an integer, the number of null dyads is stored here.

Returns:

Error code.

See also:

`igraph_reciprocity()`, `igraph_triad_census()`.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

17.2. `igraph_triad_census` — Triad census, as defined by Davis and Leinhardt

```
int igraph_triad_census(const igraph_t *graph, igraph_vector_t *res);
```

Calculating the triad census means classifying every triple of vertices in a directed graph. A triple can be in one of 16 states:

`003`

A, B, C, the empty graph.

`012`

A->B, C, a graph with a single directed edge.

`102`

A<->B, C, a graph with a mutual connection between two vertices.

`021D`

A<-B->C, the binary out-tree.

`021U`

A->B<-C, the binary in-tree.

`021C`

A->B->C, the directed line.

`111D`

A<->B<-C.

`111U`

A<->B->C.

`030T`

A->B<-C, A->C.

`030C`

A<-B<-C, A->C.

`201`

A<->B<->C.

`120D`

A<-B->C, A<->C.

`120U`

A->B<-C, A<->C.

`120C`

A->B->C, A<->C.

`210`

A->B<->C, A<->C.

`300`

A<->B<->C, A<->C, the complete graph.

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

This function calls `igraph_motifs_randesu()` which is an implementation of the FANMOD motif finder tool, see `igraph_motifs_randesu()` for details. Note that the order of the triads is not the same for `igraph_triad_census()` and `igraph_motifs_randesu()`.

Arguments:

`graph`:

The input graph. A warning is given for undirected graphs, as the result is undefined for those.

`res`:

Pointer to an initialized vector, the result is stored here in the same order as given in the list above. Note that this order is different than the one used by `igraph_motifs_randesu()`.

Returns:

Error code.

See also:

`igraph_motifs_randesu()`, `igraph_dyad_census()`.

Time complexity: TODO.

17.3. Graph motifs

17.3.1. `igraph_motifs_randesu` — Count the number of motifs in a graph

```
int igraph_motifs_randesu(const igraph_t *graph, igraph_vector_t *hist,
     int size, const igraph_vector_t *cut_prob);
```

Motifs are small connected subgraphs of a given structure in a graph. It is argued that the motif profile (ie. the number of different motifs in the graph) is characteristic for different types of networks and network function is related to the motifs in the graph.

This function is able to find the different motifs of size three and four (ie. the number of different subgraphs with three and four vertices) in the network.

In a big network the total number of motifs can be very large, so it takes a lot of time to find all of them, a sampling method can be used. This function is capable of doing sampling via the `cut_prob` argument. This argument gives the probability that a branch of the motif search tree will not be explored. See S. Wernicke and F. Rasche: FANMOD: a tool for fast network motif detection, Bioinformatics 22(9), 1152--1153, 2006 for details.

Set the `cut_prob` argument to a zero vector for finding all motifs.

Directed motifs will be counted in directed graphs and undirected motifs in undirected graphs.

Arguments:

graph:

The graph to find the motifs in.

hist:

The result of the computation, it gives the number of motifs found for each isomorphism class. See `igraph_isoclass()` for help about isomorphism classes. Note that this function does *not* count isomorphism classes that are not connected and will report NaN (more precisely `IGRAPH_NAN`) for them.

size:

The size of the motifs to search for. Only three and four are implemented currently. The limitation is not in the motif finding code, but the graph isomorphism code.

cut_prob:

Vector of probabilities for cutting the search tree at a given level. The first element is the first level, etc. Supply all zeros here (of length `size`) to find all motifs in a graph.

Returns:

Error code.

See also:

`igraph_motifs_randesu_estimate()` for estimating the number of motifs in a graph, this can help to set the `cut_prob` parameter; `igraph_motifs_randesu_no()` to calculate the total number of motifs of a given size in a graph; `igraph_motifs_randesu_callback()` for calling a callback function for every motif found.

Time complexity: TODO.

Example 17-1. File examples/simple/igraph_motifs_randesu.c

17.3.2. `igraph_motifs_randesu_no` — Count the total number of motifs in a graph

```
int igraph_motifs_randesu_no(const igraph_t *graph, igraph_integer_t *no,
        int size, const igraph_vector_t *cut_prob);
```

This function counts the total number of motifs in a graph without assigning isomorphism classes to them.

Directed motifs will be counted in directed graphs and undirected motifs in undirected graphs.

Arguments:

graph:

The graph object to study.

no:

Pointer to an integer type, the result will be stored here.

size:

The size of the motifs to count.

cut_prob:

Vector giving the probabilities that a branch of the search tree will be cut at a given level.

Returns:

Error code.

See also:

`igraph_motifs_randesu()`, `igraph_motifs_randesu_estimate()`.

Time complexity: TODO.

17.3.3. `igraph_motifs_randesu_estimate` — Estimate the total number of motifs in a graph

```
int igraph_motifs_randesu_estimate(const igraph_t *graph, igraph_integer_t *est,
       int size, const igraph_vector_t *cut_prob,
       igraph_integer_t sample_size,
       const igraph_vector_t *parsample);
```

This function is useful for large graphs for which it is not feasible to count all the different motifs, because there is very many of them.

The total number of motifs is estimated by taking a sample of vertices and counts all motifs in which these vertices are included. (There is also a `cut_prob` parameter which gives the probabilities to cut a branch of the search tree.)

Directed motifs will be counted in directed graphs and undirected motifs in undirected graphs.

Arguments:

`graph`:

The graph object to study.

`est`:

Pointer to an integer type, the result will be stored here.

`size`:

The size of the motif to look for.

`cut_prob`:

Vector giving the probabilities to cut a branch of the search tree and omit counting the motifs in that branch. It contains a probability for each level. Supply `size` zeros here to count all the motifs in the sample.

`sample_size`:

The number of vertices to use as the sample. This parameter is only used if the `parsample` argument is a null pointer.

`parsample`:

Either pointer to an initialized vector or a null pointer. If a vector then the vertex ids in the vector are used as a sample. If a null pointer then the `sample_size` argument is used to create a sample of vertices drawn with uniform probability.

Returns:

Error code.

See also:

`igraph_motifs_randesu()`, `igraph_motifs_randesu_no()`.

Time complexity: TODO.

17.3.4. `igraph_motifs_randesu_callback` — Finds motifs in a graph and calls a function for each of them

```
int igraph_motifs_randesu_callback(const igraph_t *graph, int size,
  const igraph_vector_t *cut_prob, igraph_motifs_handler_t *callback,
  void* extra);
```

Similarly to `igraph_motifs_randesu()`, this function is able to find the different motifs of size three and four (ie. the number of different subgraphs with three and four vertices) in the network. However, instead of counting them, the function will call a callback function for each motif found to allow further tests or post-processing.

The `cut_prob` argument also allows sampling the motifs, just like for `igraph_motifs_randesu()`. Set the `cut_prob` argument to a zero vector for finding all motifs.

Arguments:

graph:

The graph to find the motifs in.

size:

The size of the motifs to search for. Only three and four are implemented currently. The limitation is not in the motif finding code, but the graph isomorphism code.

cut_prob:

Vector of probabilities for cutting the search tree at a given level. The first element is the first level, etc. Supply all zeros here (of length `size`) to find all motifs in a graph.

callback:

A pointer to a function of type `igraph_motifs_handler_t`. This function will be called whenever a new motif is found.

extra:

Extra argument to pass to the callback function.

Returns:

Error code.

Time complexity: TODO.

Example 17-2. File examples/simple/igraph_motifs_randesu.c

17.3.5. `igraph_motifs_handler_t` — Callback type for `igraph_motifs_randesu_callback`

```
typedef igraph_bool_t igraph_motifs_handler_t(const igraph_t *graph,
           igraph_vector_t *vids,
           int isoclass,
           void* extra);
```

`igraph_motifs_randesu_callback()` calls a specified callback function whenever a new motif is found during a motif search. This callback function must be of type `igraph_motifs_handler_t`. It has the following arguments:

Arguments:

graph:

The graph that that algorithm is working on. Of course this must not be modified.

vids:

The IDs of the vertices in the motif that has just been found. This vector is owned by the motif search algorithm, so do not modify or destroy it; make a copy of it if you need it later.

isoclass:

The isomorphism class of the motif that has just been found. Use `igraph_isoclass` or `igraph_isoclass_subgraph` to find out which isomorphism class belongs to a given motif.

extra:

The extra argument that was passed to `igraph_motifs_randesu_callback()`.

Returns:

A logical value, if TRUE (=non-zero), that is interpreted as a request to stop the motif search and return to the caller.

See also:

`igraph_motifs_randesu_callback()`

Chapter 18. Generating Layouts for Graph Drawing

18.1. 2D layout generators

Layout generator functions (or at least most of them) try to place the vertices and edges of a graph on a 2D plane or in 3D space in a way which visually pleases the human eye.

They take a graph object and a number of parameters as arguments and return an igraph_matrix_t, in which each row gives the coordinates of a vertex.

18.1.1. `igraph_layout_random` — Places the vertices uniform randomly on a plane.

```
int igraph_layout_random(const igraph_t *graph, igraph_matrix_t *res);
```

Arguments:

graph:

Pointer to an initialized graph object.

res:

Pointer to an initialized matrix object. This will contain the result and will be resized as needed.

Returns:

Error code. The current implementation always returns with success.

Time complexity: O(|V|), the number of vertices.

18.1.2. `igraph_layout_circle` — Places the vertices uniformly on a circle, in the order of vertex ids.

```
int igraph_layout_circle(const igraph_t *graph, igraph_matrix_t *res);
```

Arguments:

graph:

Pointer to an initialized graph object.

res:

Pointer to an initialized matrix object. This will contain the result and will be resized as needed.

Returns:

Error code.

Time complexity: O(|V|), the number of vertices.

18.1.3. `igraph_layout_star` — Generate a star-like layout

```
int igraph_layout_star(const igraph_t *graph, igraph_matrix_t *res,
          igraph_integer_t center, const igraph_vector_t *order);
```

Arguments:

graph:

The input graph.

res:

Pointer to an initialized matrix object. This will contain the result and will be resized as needed.

center:

The id of the vertex to put in the center.

order:

A numeric vector giving the order of the vertices (including the center vertex!). If a null pointer, then the vertices are placed in increasing vertex id order.

Returns:

Error code.

Time complexity: O(|V|), linear in the number of vertices.

See also:

`igraph_layout_circle()` and other layout generators.

18.1.4. `igraph_layout_grid` — Places the vertices on a regular grid on the plane.

```
int igraph_layout_grid(const igraph_t *graph, igraph_matrix_t *res, long int width);
```

Arguments:

graph:

Pointer to an initialized graph object.

res:

Pointer to an initialized matrix object. This will contain the result and will be resized as needed.

width:

The number of vertices in a single row of the grid. When zero or negative, the width of the grid will be the square root of the number of vertices, rounded up if needed.

Returns:

Error code. The current implementation always returns with success.

Time complexity: O(|V|), the number of vertices.

18.1.5. `igraph_layout_graphopt` — Optimizes vertex layout via the graphopt algorithm.

```
int igraph_layout_graphopt(const igraph_t *graph, igraph_matrix_t *res,
      igraph_integer_t niter,
      igraph_real_t node_charge, igraph_real_t node_mass,
      igraph_real_t spring_length,
      igraph_real_t spring_constant,
      igraph_real_t max_sa_movement,
      igraph_bool_t use_seed);
```

This is a port of the graphopt layout algorithm by Michael Schmuhl. graphopt version 0.4.1 was rewritten in C and the support for layers was removed (might be added later) and a code was a bit reorganized to avoid some unnecessary steps is the node charge (see below) is zero.

graphopt uses physical analogies for defining attracting and repelling forces among the vertices and then the physical system is simulated until it reaches an equilibrium. (There is no simulated annealing or anything like that, so a stable fixed point is not guaranteed.)

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

Arguments:

`graph`:

The input graph.

`res`:

Pointer to an initialized matrix, the result will be stored here and its initial contents is used the starting point of the simulation if the *`use_seed`* argument is true. Note that in this case the matrix should have the proper size, otherwise a warning is issued and the supplied values are ignored. If no starting positions are given (or they are invalid) then a random staring position is used. The matrix will be resized if needed.

niter:

Integer constant, the number of iterations to perform. Should be a couple of hundred in general. If you have a large graph then you might want to only do a few iterations and then check the result. If it is not good enough you can feed it in again in the *res* argument. The original graphopt default if 500.

node_charge:

The charge of the vertices, used to calculate electric repulsion. The original graphopt default is 0.001.

node_mass:

The mass of the vertices, used for the spring forces. The original graphopt defaults to 30.

spring_length:

The length of the springs, an integer number. The original graphopt defaults to zero.

spring_constant:

The spring constant, the original graphopt defaults to one.

max_sa_movement:

Real constant, it gives the maximum amount of movement allowed in a single step along a single axis. The original graphopt default is 5.

use_seed:

Logical scalar, whether to use the positions in *res* as a starting configuration. See also *res* above.

Returns:

Error code.

Time complexity: O(n (|V|^2+|E|)), n is the number of iterations, |V| is the number of vertices, |E| the number of edges. If *node_charge* is zero then it is only O(n|E|).

18.1.6. The DrL layout generator

DrL is a sophisticated layout generator developed and implemented by Shawn Martin et al. As of October 2012 the original DrL homepage is unfortunately not available. You can read more about this algorithm in the following technical report: Martin, S., Brown, W.M., Klavans, R., Boyack, K.W., DrL: Distributed Recursive (Graph) Layout. SAND Reports, 2008. 2936: p. 1-10.

Only a subset of the complete DrL functionality is included in igraph, parallel runs and recursive, multi-level layouting is not supported.

The parameters of the layout are stored in an `igraph_layout_drl_options_t` structure, this can be initialized by calling the function `igraph_layout_drl_options_init()`. The fields of this structure can then be adjusted by hand if needed. The layout is calculated by an `igraph_layout_drl()` call.

18.1.6.1. `igraph_layout_drl_options_t` — Parameters for the DrL layout generator

```
typedef struct igraph_layout_drl_options_t {
  igraph_real_t    edge_cut;
  igraph_integer_t init_iterations;
  igraph_real_t    init_temperature;
  igraph_real_t    init_attraction;
  igraph_real_t    init_damping_mult;
  igraph_integer_t liquid_iterations;
  igraph_real_t    liquid_temperature;
  igraph_real_t    liquid_attraction;
  igraph_real_t    liquid_damping_mult;
  igraph_integer_t expansion_iterations;
  igraph_real_t    expansion_temperature;
  igraph_real_t    expansion_attraction;
  igraph_real_t    expansion_damping_mult;
  igraph_integer_t cooldown_iterations;
  igraph_real_t    cooldown_temperature;
  igraph_real_t    cooldown_attraction;
  igraph_real_t    cooldown_damping_mult;
  igraph_integer_t crunch_iterations;
  igraph_real_t    crunch_temperature;
  igraph_real_t    crunch_attraction;
  igraph_real_t    crunch_damping_mult;
  igraph_integer_t simmer_iterations;
  igraph_real_t    simmer_temperature;
  igraph_real_t    simmer_attraction;
  igraph_real_t    simmer_damping_mult;
} igraph_layout_drl_options_t;
```

Values:

`edge_cut`:

The edge cutting parameter. Edge cutting is done in the late stages of the algorithm in order to achieve less dense layouts. Edges are cut if there is a lot of stress on them (a large value in the objective function sum). The edge cutting parameter is a value between 0 and 1 with 0 representing no edge cutting and 1 representing maximal edge cutting. The default value is 32/40.

`init_iterations`:

Number of iterations, initial phase.

`init_temperature`:

Start temperature, initial phase.

`init_attraction`:

Attraction, initial phase.

`init_damping_mult`:

Damping factor, initial phase.

`liquid_iterations`:

Number of iterations in the liquid phase.

`liquid_temperature`:

Start temperature in the liquid phase.

`liquid_attraction`:

Attraction in the liquid phase.

`liquid_damping_mult`:

Multiplicatie damping factor, liquid phase.

`expansion_iterations`:

Number of iterations in the expansion phase.

`expansion_temperature`:

Start temperature in the expansion phase.

`expansion_attraction`:

Attraction, expansion phase.

`expansion_damping_mult`:

Damping factor, expansion phase.

`cooldown_iterations`:

Number of iterations in the cooldown phase.

`cooldown_temperature`:

Start temperature in the cooldown phase.

`cooldown_attraction`:

Attraction in the cooldown phase.

`cooldown_damping_mult`:

Damping fact int the cooldown phase.

`crunch_iterations`:

Number of iterations in the crunch phase.

`crunch_temperature`:

Start temperature in the crunch phase.

`crunch_attraction`:

Attraction in the crunch phase.

`crunch_damping_mult`:

Damping factor in the crunch phase.

`simmer_iterations`:

Number of iterations in the simmer phase.

`simmer_temperature`:

Start temperature in te simmer phase.

`simmer_attraction`:

Attraction in the simmer phase.

`simmer_damping_mult`:

Multiplicative damping factor in the simmer phase.

18.1.6.2. `igraph_layout_drl_default_t` — Predefined parameter templates for the DrL layout generator

```
typedef enum { IGRAPH_LAYOUT_DRL_DEFAULT=0,
        IGRAPH_LAYOUT_DRL_COARSEN,
        IGRAPH_LAYOUT_DRL_COARSEST,
        IGRAPH_LAYOUT_DRL_REFINE,
        IGRAPH_LAYOUT_DRL_FINAL } igraph_layout_drl_default_t;
```

These constants can be used to initialize a set of DrL parameters. These can then be modified according to the user's needs.

Values:

`IGRAPH_LAYOUT_DRL_DEFAULT`:

The deafult parameters.

`IGRAPH_LAYOUT_DRL_COARSEN`:

Slightly modified parameters to get a coarser layout.

`IGRAPH_LAYOUT_DRL_COARSEST`:

An even coarser layout.

`IGRAPH_LAYOUT_DRL_REFINE`:

Refine an already calculated layout.

`IGRAPH_LAYOUT_DRL_FINAL`:

Finalize an already refined layout.

18.1.6.3. `igraph_layout_drl_options_init` — Initialize parameters for the DrL layout generator

```
int igraph_layout_drl_options_init(igraph_layout_drl_options_t *options,
       igraph_layout_drl_default_t templ);
```

This function can be used to initialize the struct holding the parameters for the DrL layout generator. There are a number of predefined templates available, it is a good idea to start from one of these by modifying some parameters.

Arguments:

`options`:

The struct to initialize.

`templ`:

The template to use. Currently the following templates are supplied: `IGRAPH_LAYOUT_DRL_DEFAULT`, `IGRAPH_LAYOUT_DRL_COARSEN`,

IGRAPH_LAYOUT_DRL_COARSEST, IGRAPH_LAYOUT_DRL_REFINE and
IGRAPH_LAYOUT_DRL_FINAL.

Returns:

Error code.

Time complexity: O(1).

18.1.6.4. `igraph_layout_drl` — The DrL layout generator

```
int igraph_layout_drl(const igraph_t *graph, igraph_matrix_t *res,
        igraph_bool_t use_seed,
        igraph_layout_drl_options_t *options,
        const igraph_vector_t *weights,
        const igraph_vector_bool_t *fixed);
```

This function implements the force-directed DrL layout generator. Please see more in the following technical report: Martin, S., Brown, W.M., Klavans, R., Boyack, K.W., DrL: Distributed Recursive (Graph) Layout. SAND Reports, 2008. 2936: p. 1-10.

Arguments:

graph:

The input graph.

use_seed:

Logical scalar, if true, then the coordinates supplied in the *res* argument are used as starting points.

res:

Pointer to a matrix, the result layout is stored here. It will be resized as needed.

options:

The parameters to pass to the layout generator.

weights:

Edge weights, pointer to a vector. If this is a null pointer then every edge will have the same weight.

fixed:

Pointer to a logical vector, or a null pointer. This can be used to fix the position of some vertices. Vertices for which it is true will not be moved, but stay at the coordinates given in the *res* matrix. This argument is ignored if it is a null pointer or if use_seed is false.

Returns:

Error code.

Time complexity: ???.

18.1.6.5. `igraph_layout_drl_3d` — The DrL layout generator, 3d version.

```
int igraph_layout_drl_3d(const igraph_t *graph, igraph_matrix_t *res,
    igraph_bool_t use_seed,
    igraph_layout_drl_options_t *options,
    const igraph_vector_t *weights,
    const igraph_vector_bool_t *fixed);
```

This function implements the force-directed DrL layout generator. Please see more in the technical report: Martin, S., Brown, W.M., Klavans, R., Boyack, K.W., DrL: Distributed Recursive (Graph) Layout. SAND Reports, 2008. 2936: p. 1-10.

This function uses a modified DrL generator that does the layout in three dimensions.

Arguments:

graph:

The input graph.

use_seed:

Logical scalar, if true, then the coordinates supplied in the *res* argument are used as starting points.

res:

Pointer to a matrix, the result layout is stored here. It will be resized as needed.

options:

The parameters to pass to the layout generator.

weights:

Edge weights, pointer to a vector. If this is a null pointer then every edge will have the same weight.

fixed:

Pointer to a logical vector, or a null pointer. This can be used to fix the position of some vertices. Vertices for which it is true will not be moved, but stay at the coordinates given in the *res* matrix. This argument is ignored if it is a null pointer or if use_seed is false.

Returns:

Error code.

Time complexity: ???.

See also:

`igraph_layout_drl()` for the standard 2d version.

18.1.7. `igraph_layout_fruchterman_reingold` — Places the vertices on a plane according to the Fruchterman-Reingold algorithm.

```
int igraph_layout_fruchterman_reingold(const igraph_t *graph, igraph_matrix_t *res,
            igraph_integer_t niter, igraph_real_t maxdelta,
            igraph_real_t area, igraph_real_t coolexp,
            igraph_real_t repulserad, igraph_bool_t use_seed,
            const igraph_vector_t *weight,
            const igraph_vector_t *minx,
            const igraph_vector_t *maxx,
            const igraph_vector_t *miny,
            const igraph_vector_t *maxy);
```

This is a force-directed layout, see Fruchterman, T.M.J. and Reingold, E.M.: Graph Drawing by Force-directed Placement. Software -- Practice and Experience, 21/11, 1129--1164, 1991. This function was ported from the SNA R package.

Arguments:

graph:

Pointer to an initialized graph object.

res:

Pointer to an initialized matrix object. This will contain the result and will be resized as needed.

niter:

The number of iterations to do. A reasonable default value is 500.

maxdelta:

The maximum distance to move a vertex in an iteration. A reasonable default value is the number of vertices.

area:

The area parameter of the algorithm. A reasonable default is the square of the number of vertices.

coolexp:

The cooling exponent of the simulated annealing. A reasonable default is 1.5.

repulserad:

Determines the radius at which vertex-vertex repulsion cancels out attraction of adjacent vertices. A reasonable default is *area* times the number of vertices.

use_seed:

Logical, if true the supplied values in the *res* argument are used as an initial layout, if false a random initial layout is used.

weight:

Pointer to a vector containing edge weights, the attraction along the edges will be multiplied by these. It will be ignored if it is a null-pointer.

minx:

Pointer to a vector, or a `NULL` pointer. If not a `NULL` pointer then the vector gives the minimum "x" coordinate for every vertex.

maxx:

Same as *minx*, but the maximum "x" coordinates.

miny:

Pointer to a vector, or a `NULL` pointer. If not a `NULL` pointer then the vector gives the minimum "y" coordinate for every vertex.

maxy:

Same as *miny*, but the maximum “y” coordinates.

Returns:

Error code.

Time complexity: O(|V|^2) in each iteration, |V| is the number of vertices in the graph.

18.1.8. `igraph_layout_kamada_kawai` — Places the vertices on a plane according the Kamada-Kawai algorithm.

```
int igraph_layout_kamada_kawai(const igraph_t *graph, igraph_matrix_t *res,
          igraph_integer_t niter, igraph_real_t sigma,
          igraph_real_t initemp, igraph_real_t coolexp,
          igraph_real_t kkconst, igraph_bool_t use_seed,
          const igraph_vector_t *minx,
          const igraph_vector_t *maxx,
          const igraph_vector_t *miny,
          const igraph_vector_t *maxy);
```

This is a force directed layout, see Kamada, T. and Kawai, S.: An Algorithm for Drawing General Undirected Graphs. Information Processing Letters, 31/1, 7--15, 1989. This function was ported from the SNA R package.

Arguments:

graph:

A graph object.

res:

Pointer to an initialized matrix object. This will contain the result (x-positions in column zero and y-positions in column one) and will be resized if needed.

niter:

The number of iterations to perform. A reasonable default value is 1000.

sigma:

Sets the base standard deviation of position change proposals. A reasonable default value is the number of vertices / 4.

initemp:

Sets the initial temperature for the annealing. A reasonable default value is 10.

coolexp:

The cooling exponent of the annealing. A reasonable default value is 0.99.

kkconst:

The Kamada-Kawai vertex attraction constant. Typical value: (number of vertices)^2

use_seed:

Boolean, whether to use the values supplied in the *res* argument as the initial configuration. If zero then a random initial configuration is used.

minx:

Pointer to a vector, or a NULL pointer. If not a NULL pointer then the vector gives the minimum "x" coordinate for every vertex.

maxx:

Same as *minx*, but the maximum "x" coordinates.

miny:

Pointer to a vector, or a NULL pointer. If not a NULL pointer then the vector gives the minimum "y" coordinate for every vertex.

maxy:

Same as *miny*, but the maximum "y" coordinates.

Returns:

Error code.

Time complexity: O(|V|^2) for each iteration, |V| is the number of vertices in the graph.

18.1.9. `igraph_layout_mds` — Place the vertices on a plane using multidimensional scaling.

```
int igraph_layout_mds(const igraph_t* graph, igraph_matrix_t *res,
                      const igraph_matrix_t *dist, long int dim,
                      igraph_arpack_options_t *options);
```

This layout requires a distance matrix, where the intersection of row i and column j specifies the desired distance between vertex i and vertex j. The algorithm will try to place the vertices in a space having a given number of dimensions in a way that approximates the distance relations prescribed in the distance matrix. igraph uses the classical multidimensional scaling by Torgerson; for more details, see Cox & Cox: Multidimensional Scaling (1994), Chapman and Hall, London.

If the input graph is disconnected, igraph will decompose it first into its subgraphs, lay out the subgraphs one by one using the appropriate submatrices of the distance matrix, and then merge the layouts using `igraph_layout_merge_dla`. Since `igraph_layout_merge_dla` works for 2D layouts only, you cannot run the MDS layout on disconnected graphs for more than two dimensions.

Arguments:

graph:

A graph object.

res:

Pointer to an initialized matrix object. This will contain the result and will be resized if needed.

dist:

The distance matrix. It must be symmetric and this function does not check whether the matrix is indeed symmetric. Results are unspecified if you pass a non-symmetric matrix here. You can set this parameter to null; in this case, the shortest path lengths between vertices will be used as distances.

dim:

The number of dimensions in the embedding space. For 2D layouts, supply 2 here.

options:

This argument is currently ignored, it was used for ARPACK, but LAPACK is used now for calculating the eigenvectors.

Returns:

Error code.

Added in version 0.6.

Time complexity: usually around O(|V|^2 dim).

18.1.10. `igraph_layout_grid_fruchterman_reingold` — Force based layout generator for large graphs.

```
int igraph_layout_grid_fruchterman_reingold(const igraph_t *graph,
               igraph_matrix_t *res,
               igraph_integer_t niter, igraph_real_t maxdelta,
               igraph_real_t area, igraph_real_t coolexp,
               igraph_real_t repulserad,
               igraph_real_t cellsize,
               igraph_bool_t use_seed,
               const igraph_vector_t *weight);
```

This algorithm is the same as the Fruchterman-Reingold layout generator, but it partitions the 2d space to a grid and and vertex repulsion is calculated only for vertices nearby.

Arguments:

`graph`:

The graph object.

`res`:

The result, the coordinates in a matrix. The parameter should point to an initialized matrix object and will be resized.

`niter`:

The number of iterations to do. A reasonable default value is 500.

`maxdelta`:

The maximum distance to move a vertex in an iteration. A reasonable default value is the number of vertices.

`area`:

The area parameter of the algorithm. A reasonable default is the square of the number of vertices.

coolexp:

The cooling exponent of the simulated annealing. A reasonable default is 1.5.

repulserad:

Determines the radius at which vertex-vertex repulsion cancels out attraction of adjacent vertices. A reasonable default is *area* times the number of vertices.

cellsize:

The size of the grid cells. A reasonable default is the fourth root of *area* (or the square root of the number of vertices if *area* is also left at its default value)

use_seed:

Logical, if true, the coordinates passed in *res* (should have the appropriate size) will be used for the first iteration.

weight:

Pointer to a vector containing edge weights, the attraction along the edges will be multiplied by these. It will be ignored if it is a null-pointer.

Returns:

Error code.

Added in version 0.2.

Time complexity: ideally (constant number of vertices in each cell) O(niter*(|V|+|E|)), in the worst case O(niter*(|V|^2+|E|)).

18.1.11. `igraph_layout_lgl` — Force based layout algorithm for large graphs.

```
int igraph_layout_lgl(const igraph_t *graph, igraph_matrix_t *res,
        igraph_integer_t maxit, igraph_real_t maxdelta,
        igraph_real_t area, igraph_real_t coolexp,
        igraph_real_t repulserad, igraph_real_t cellsize,
        igraph_integer_t proot);
```

This is a layout generator similar to the Large Graph Layout algorithm and program (http://bioinformatics.icmb.utexas.edu/lgl/). But unlike LGL, this version uses a Fruchterman-Reingold style simulated annealing algorithm for placing the vertices. The speedup is achieved by placing the vertices on a grid and calculating the repulsion only for vertices which are closer to each other than a limit.

Arguments:

graph:

> The (initialized) graph object to place.

res:

> Pointer to an initialized matrix object to hold the result. It will be resized if needed.

maxit:

> The maximum number of cooling iterations to perform for each layout step. A reasonable default is 150.

maxdelta:

> The maximum length of the move allowed for a vertex in a single iteration. A reasonable default is the number of vertices.

area:

> This parameter gives the area of the square on which the vertices will be placed. A reasonable default value is the number of vertices squared.

coolexp:

> The cooling exponent. A reasonable default value is 1.5.

repulserad:

> Determines the radius at which vertex-vertex repulsion cancels out attraction of adjacent vertices. A reasonable default value is *area* times the number of vertices.

cellsize:

> The size of the grid cells, one side of the square. A reasonable default value is the fourth root of *area* (or the square root of the number of vertices if *area* is also left at its default value).

proot:

> The root vertex, this is placed first, its neighbors in the first iteration, second neighbors in the second, etc. If negative then a random vertex is chosen.

Returns:

> Error code.

Added in version 0.2.

Time complexity: ideally O(dia*maxit*(|V|+|E|)), |V| is the number of vertices, dia is the diameter of the graph, worst case complexity is still O(dia*maxit*(|V|^2+|E|)), this is the case when all vertices happen to be in the same grid cell.

18.1.12. `igraph_layout_reingold_tilford` — Reingold-Tilford layout for tree graphs

```
int igraph_layout_reingold_tilford(const igraph_t *graph,
       igraph_matrix_t *res,
       igraph_neimode_t mode,
       const igraph_vector_t *roots,
       const igraph_vector_t *rootlevel);
```

Arranges the nodes in a tree where the given node is used as the root. The tree is directed downwards and the parents are centered above its children. For the exact algorithm, see:

Reingold, E and Tilford, J: Tidier drawing of trees. IEEE Trans. Softw. Eng., SE-7(2):223--228, 1981

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

Arguments:

`graph`:

The graph object.

`res`:

The result, the coordinates in a matrix. The parameter should point to an initialized matrix object and will be resized.

`mode`:

Specifies which edges to consider when building the tree. If it is `IGRAPH_OUT` then only the outgoing, if it is `IGRAPH_IN` then only the incoming edges of a parent are considered. If it is `IGRAPH_ALL` then all edges are used (this was the behavior in igraph 0.5 and before). This parameter also influences how the root vertices are calculated, if they are not given. See the *`roots`* parameter.

roots:

The index of the root vertex or root vertices. If this is a non-empty vector then the supplied vertex ids are used as the roots of the trees (or a single tree if the graph is connected). If it is a null pointer of a pointer to an empty vector, then the root vertices are automatically calculated based on topological sorting, performed with the opposite mode than the *mode* argument. After the vertices have been sorted, one is selected from each component.

rootlevel:

This argument can be useful when drawing forests which are not trees (i.e. they are unconnected and have tree components). It specifies the level of the root vertices for every tree in the forest. It is only considered if not a null pointer and the *roots* argument is also given (and it is not a null pointer of an empty vector).

Returns:

Error code.

Added in version 0.2.

See also:

`igraph_layout_reingold_tilford_circular()`.

Example 18-1. File examples/simple/igraph_layout_reingold_tilford.c

18.1.13. `igraph_layout_reingold_tilford_circular` — Circular Reingold-Tilford layout for trees

```
int igraph_layout_reingold_tilford_circular(const igraph_t *graph,
          igraph_matrix_t *res,
          igraph_neimode_t mode,
          const igraph_vector_t *roots,
          const igraph_vector_t *rootlevel);
```

This layout is almost the same as `igraph_layout_reingold_tilford()`, but the tree is drawn in a circular way, with the root vertex in the center.

Arguments:

graph:

The graph object.

res:

The result, the coordinates in a matrix. The parameter should point to an initialized matrix object and will be resized.

mode:

Specifies which edges to consider when building the tree. If it is `IGRAPH_OUT` then only the outgoing, if it is `IGRAPH_IN` then only the incoming edges of a parent are considered. If it is `IGRAPH_ALL` then all edges are used (this was the behavior in igraph 0.5 and before). This parameter also influences how the root vertices are calculated, if they are not given. See the *roots* parameter.

roots:

The index of the root vertex or root vertices. If this is a non-empty vector then the supplied vertex ids are used as the roots of the trees (or a single tree if the graph is connected). If it is a null pointer of a pointer to an empty vector, then the root vertices are automatically calculated based on topological sorting, performed with the opposite mode than the *mode* argument. After the vertices have been sorted, one is selected from each component.

rootlevel:

This argument can be useful when drawing forests which are not trees (i.e. they are unconnected and have tree components). It specifies the level of the root vertices for every tree in the forest. It is only considered if not a null pointer and the *roots* argument is also given (and it is not a null pointer of an empty vector). Note that if you supply a null pointer here and the graph has multiple components, all of the root vertices will be mapped to the origin of the coordinate system, which does not really make sense.

Returns:

Error code.

See also:

`igraph_layout_reingold_tilford()`.

18.1.14. `igraph_layout_sugiyama` — Sugiyama layout algorithm for layered directed acyclic graphs.

```
int igraph_layout_sugiyama(const igraph_t *graph, igraph_matrix_t *res,
        igraph_t *extd_graph, igraph_vector_t *extd_to_orig_eids,
        const igraph_vector_t* layers, igraph_real_t hgap, igraph_real_t vgap,
        long int maxiter, const igraph_vector_t *weights);
```

This layout algorithm is designed for directed acyclic graphs where each vertex is assigned to a layer. Layers are indexed from zero, and vertices of the same layer will be placed on the same horizontal line. The X coordinates of vertices within each layer are decided by the heuristic proposed by Sugiyama et al to minimize edge crossings.

You can also try to lay out undirected graphs, graphs containing cycles, or graphs without an a priori layered assignment with this algorithm. igraph will try to eliminate cycles and assign vertices to layers, but there is no guarantee on the quality of the layout in such cases.

The Sugiyama layout may introduce "bends" on the edges in order to obtain a visually more pleasing layout. This is achieved by adding dummy nodes to edges spanning more than one layer. The resulting layout assigns coordinates not only to the nodes of the original graph but also to the dummy nodes. The layout algorithm will also return the extended graph with the dummy nodes. An edge in the original graph may either be mapped to a single edge in the extended graph or a *path* that starts and ends in the original source and target vertex and passes through multiple dummy vertices. In such cases, the user may also request the mapping of the edges of the extended graph back to the edges of the original graph.

For more details, see K. Sugiyama, S. Tagawa and M. Toda, "Methods for Visual Understanding of Hierarchical Systems". IEEE Transactions on Systems, Man and Cybernetics 11(2):109-125, 1981.

Arguments:

`graph`:

Pointer to an initialized graph object.

`res`:

Pointer to an initialized matrix object. This will contain the result and will be resized as needed. The first |V| rows of the layout will contain the coordinates of the original graph, the remaining rows contain the positions of the dummy nodes. Therefore, you can use the result both with *`graph`* or with *`extended_graph`*.

`extended_graph`:

Pointer to an uninitialized graph object or `NULL`. The extended graph with the added dummy nodes will be returned here. In this graph, each edge points downwards to lower layers, spans exactly one layer and the first |V| vertices coincide with the vertices of the original graph.

`extd_to_orig_eids`:

Pointer to a vector or `NULL`. If not `NULL`, the mapping from the edge IDs of the extended graph back to the edge IDs of the original graph will be stored here.

`layers`:

The layer index for each vertex or `NULL` if the layers should be determined automatically by igraph.

`hgap`:

The preferred minimum horizontal gap between vertices in the same layer.

`vgap`:

The distance between layers.

`maxiter`:

Maximum number of iterations in the crossing minimization stage. 100 is a reasonable default; if you feel that you have too many edge crossings, increase this.

`weights`:

Weights of the edges. These are used only if the graph contains cycles; igraph will tend to reverse edges with smaller weights when breaking the cycles.

18.2. 3D layout generators

18.2.1. `igraph_layout_random_3d` — Random layout in 3D

```
int igraph_layout_random_3d(const igraph_t *graph, igraph_matrix_t *res);
```

Arguments:

`graph`:

The graph to place.

res:

Pointer to an initialized matrix object. It will be resized to hold the result.

Returns:

Error code. The current implementation always returns with success.

Added in version 0.2.

Time complexity: O(|V|), the number of vertices.

18.2.2. `igraph_layout_sphere` — Places vertices (more or less) uniformly on a sphere.

```
int igraph_layout_sphere(const igraph_t *graph, igraph_matrix_t *res);
```

The algorithm was described in the following paper: Distributing many points on a sphere by E.B. Saff and A.B.J. Kuijlaars, *Mathematical Intelligencer* 19.1 (1997) 5--11.

Arguments:

graph:

Pointer to an initialized graph object.

res:

Pointer to an initialized matrix object. This will contain the result and will be resized as needed.

Returns:

Error code. The current implementation always returns with success.

Added in version 0.2.

Time complexity: O(|V|), the number of vertices in the graph.

18.2.3. `igraph_layout_grid_3d` — Places the vertices on a regular grid in the 3D space.

```
int igraph_layout_grid_3d(const igraph_t *graph, igraph_matrix_t *res,
    long int width, long int height);
```

Arguments:

graph:

Pointer to an initialized graph object.

res:

Pointer to an initialized matrix object. This will contain the result and will be resized as needed.

width:

The number of vertices in a single row of the grid. When zero or negative, the width is determined automatically.

height:

The number of vertices in a single column of the grid. When zero or negative, the height is determined automatically.

Returns:

Error code. The current implementation always returns with success.

Time complexity: O(|V|), the number of vertices.

18.2.4. `igraph_layout_fruchterman_reingold_3d` — 3D Fruchterman-Reingold algorithm.

```
int igraph_layout_fruchterman_reingold_3d(const igraph_t *graph,
       igraph_matrix_t *res,
       igraph_integer_t niter, igraph_real_t maxdelta,
       igraph_real_t volume, igraph_real_t coolexp,
       igraph_real_t repulserad,
       igraph_bool_t use_seed,
```

```
const igraph_vector_t *weight,
const igraph_vector_t *minx,
const igraph_vector_t *maxx,
const igraph_vector_t *miny,
const igraph_vector_t *maxy,
const igraph_vector_t *minz,
const igraph_vector_t *maxz);
```

This is the 3D version of the force based Fruchterman-Reingold layout (see `igraph_layout_fruchterman_reingold` for the 2D version

This function was ported from the SNA R package.

Arguments:

graph:

Pointer to an initialized graph object.

res:

Pointer to an initialized matrix object. This will contain the result and will be resized as needed.

niter:

The number of iterations to do. A reasonable default value is 500.

maxdelta:

The maximum distance to move a vertex in an iteration. A reasonable default value is the number of vertices.

volume:

The volume parameter of the algorithm. A reasonable default is the number of vertices^3.

coolexp:

The cooling exponent of the simulated annealing. A reasonable default is 1.5.

repulserad:

Determines the radius at which vertex-vertex repulsion cancels out attraction of adjacent vertices. A reasonable default is *volume* times the number of vertices.

use_seed:

Logical, if true the supplied values in the *res* argument are used as an initial layout, if false a random initial layout is used.

weight:

Pointer to a vector containing edge weights, the attraction along the edges will be multiplied by these. It will be ignored if it is a null-pointer.

minx:

Pointer to a vector, or a `NULL` pointer. If not a `NULL` pointer then the vector gives the minimum “x” coordinate for every vertex.

maxx:

Same as *minx*, but the maximum “x” coordinates.

miny:

Pointer to a vector, or a `NULL` pointer. If not a `NULL` pointer then the vector gives the minimum “y” coordinate for every vertex.

maxy:

Same as *miny*, but the maximum “y” coordinates.

minz:

Pointer to a vector, or a `NULL` pointer. If not a `NULL` pointer then the vector gives the minimum “z” coordinate for every vertex.

maxz:

Same as *minz*, but the maximum “z” coordinates.

Returns:

Error code.

Added in version 0.2.

Time complexity: O(|V|^2) in each iteration, |V| is the number of vertices in the graph.

18.2.5. `igraph_layout_kamada_kawai_3d` — 3D version of the force based Kamada-Kawai layout.

```
int igraph_layout_kamada_kawai_3d(const igraph_t *graph, igraph_matrix_t *res,
      igraph_integer_t niter, igraph_real_t sigma,
      igraph_real_t initemp, igraph_real_t coolexp,
      igraph_real_t kkconst, igraph_bool_t use_seed,
      igraph_bool_t fixz,
```

```
const igraph_vector_t *minx,
const igraph_vector_t *maxx,
const igraph_vector_t *miny,
const igraph_vector_t *maxy,
const igraph_vector_t *minz,
const igraph_vector_t *maxz);
```

The pair of the `igraph_layout_kamada_kawai` 2D layout generator

This function was ported from the SNA R package.

Arguments:

graph:

A graph object.

res:

Pointer to an initialized matrix object. This will contain the result and will be resized if needed.

niter:

The number of iterations to perform. A reasonable default value is 1000.

sigma:

Sets the base standard deviation of position change proposals. A reasonable default value is the number of vertices / 4.

initemp:

Sets the initial temperature for the annealing. A reasonable default value is 10.

coolexp:

The cooling exponent of the annealing. A reasonable default value is 0.99.

kkconst:

The Kamada-Kawai vertex attraction constant. Typical value: (number of vertices)^2

use_seed:

Boolean, whether to use the values cupplied in the *res* argument as the initial configuration. If zero then a random initial configuration is used.

fixz:

Logical, whether to fix the third coordinate of the input matrix.

Returns:

Error code.

Added in version 0.2.

Time complexity: O(|V|^2) for each iteration, |V| is the number of vertices in the graph.

18.3. Merging layouts

18.3.1. `igraph_layout_merge_dla` — Merge multiple layouts by using a DLA algorithm

```
int igraph_layout_merge_dla(igraph_vector_ptr_t *thegraphs,
       igraph_vector_ptr_t *coords,
       igraph_matrix_t *res);
```

First each layout is covered by a circle. Then the layout of the largest graph is placed at the origin. Then the other layouts are placed by the DLA algorithm, larger ones first and smaller ones last.

Arguments:

`thegraphs`:

Pointer vector containing the graph object of which the layouts will be merged.

`coords`:

Pointer vector containing matrix objects with the 2d layouts of the graphs in *`thegraphs`*.

`res`:

Pointer to an initialized matrix object, the result will be stored here. It will be resized if needed.

Returns:

Error code.

Added in version 0.2. This function is experimental.

Time complexity: TODO.

Chapter 19. Reading and Writing Graphs from and to Files

These functions can write a graph to a file, or read a graph from a file.

Note that as **igraph** uses the traditional C streams, it is possible to read/write files from/to memory, at least on GNU operating systems supporting “non-standard” streams.

19.1. Simple edge list and similar formats

19.1.1. `igraph_read_graph_edgelist` — Reads an edge list from a file and creates a graph.

```
int igraph_read_graph_edgelist(igraph_t *graph, FILE *instream,
          igraph_integer_t n, igraph_bool_t directed);
```

This format is simply a series of even number integers separated by whitespace. The one edge (ie. two integers) per line format is thus not required (but recommended for readability). Edges of directed graphs are assumed to be in from, to order.

Arguments:

`graph`:

Pointer to an uninitialized graph object.

`instream`:

Pointer to a stream, it should be readable.

`n`:

The number of vertices in the graph. If smaller than the largest integer in the file it will be ignored. It is thus safe to supply zero here.

`directed`:

Logical, if true the graph is directed, if false it will be undirected.

Returns:

Error code: `IGRAPH_PARSEERROR`: if there is a problem reading the file, or the file is syntactically incorrect.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges. It is assumed that reading an integer requires O(1) time.

19.1.2. `igraph_write_graph_edgelist` — Writes the edge list of a graph to a file.

```
int igraph_write_graph_edgelist(const igraph_t *graph, FILE *outstream);
```

One edge is written per line, separated by a single space. For directed graphs edges are written in from, to order.

Arguments:

graph:

The graph object to write.

outstream:

Pointer to a stream, it should be writable.

Returns:

Error code: `IGRAPH_EFILE` if there is an error writing the file.

Time complexity: O(|E|), the number of edges in the graph. It is assumed that writing an integer to the file requires O(1) time.

19.1.3. `igraph_read_graph_ncol` — Reads a `.ncol` file used by LGL.

```
int igraph_read_graph_ncol(igraph_t *graph, FILE *instream,
      igraph_strvector_t *predefnames,
      igraph_bool_t names,
      igraph_add_weights_t weights,
      igraph_bool_t directed);
```

Also useful for creating graphs from “named” (and optionally weighted) edge lists.

This format is used by the Large Graph Layout program (http://bioinformatics.icmb.utexas.edu/lgl/), and it is simply a symbolic weighted edge list. It is a simple text file with one edge per line. An edge is defined by two symbolic vertex names separated by whitespace. (The symbolic vertex names themselves cannot contain whitespace. They might follow by an optional number, this will be the weight of the edge; the number can be negative and can be in scientific notation. If there is no weight specified to an edge it is assumed to be zero.

The resulting graph is always undirected. LGL cannot deal with files which contain multiple or loop edges, this is however not checked here, as **igraph** is happy with these.

Arguments:

graph:

Pointer to an uninitialized graph object.

instream:

Pointer to a stream, it should be readable.

predefnames:

Pointer to the symbolic names of the vertices in the file. If `NULL` is given here then vertex ids will be assigned to vertex names in the order of their appearance in the \c .ncol file. If it is not `NULL` and some unknown vertex names are found in the \c .ncol file then new vertex ids will be assigned to them.

names:

Logical value, if TRUE the symbolic names of the vertices will be added to the graph as a vertex attribute called “name”.

weights:

Whether to add the weights of the edges to the graph as an edge attribute called “weight”. `IGRAPH_ADD_WEIGHTS_YES` adds the weights (even if they are not present in the file, in this case they are assumed to be zero). `IGRAPH_ADD_WEIGHTS_NO` does not add any edge attribute. `IGRAPH_ADD_WEIGHTS_IF_PRESENT` adds the attribute if and only if there is at least one explicit edge weight in the input file.

directed:

Whether to create a directed graph. As this format was originally used only for undirected graphs there is no information in the file about the directedness of the graph. Set this parameter to `IGRAPH_DIRECTED` or `IGRAPH_UNDIRECTED` to create a directed or undirected graph.

Returns:

Error code: `IGRAPH_PARSEERROR`: if there is a problem reading the file, or the file is syntactically incorrect.

Time complexity: O(|V|+|E|log(|V|)) if we neglect the time required by the parsing. As usual |V| is the number of vertices, while |E| is the number of edges.

See also:

`igraph_read_graph_lgl()`, `igraph_write_graph_ncol()`

19.1.4. `igraph_write_graph_ncol` — Writes the graph to a file in `.ncol` format

```
int igraph_write_graph_ncol(const igraph_t *graph, FILE *outstream,
       const char *names, const char *weights);
```

`.ncol` is a format used by LGL, see `igraph_read_graph_ncol()` for details.

Note that having multiple or loop edges in an `.ncol` file breaks the LGL software but **igraph** does not check for this condition.

Arguments:

graph:

The graph to write.

outstream:

The stream object to write to, it should be writable.

names:

The name of the vertex attribute, if symbolic names are written to the file. If not, supply 0 here.

weights:

The name of the edge attribute, if they are also written to the file. If you don't want weights, supply 0 here.

Returns:

Error code: `IGRAPH_EFILE` if there is an error writing the file.

Time complexity: O(|E|), the number of edges. All file operations are expected to have time complexity O(1).

See also:

`igraph_read_graph_ncol()`, `igraph_write_graph_lgl()`

19.1.5. `igraph_read_graph_lgl` — Reads a graph from an `.lgl` file

```
int igraph_read_graph_lgl(igraph_t *graph, FILE *instream,
     igraph_bool_t names,
     igraph_add_weights_t weights,
     igraph_bool_t directed);
```

The `.lgl` format is used by the Large Graph Layout visualization software (http://bioinformatics.icmb.utexas.edu/lgl/), it can describe undirected optionally weighted graphs. From the LGL manual:

> The second format is the LGL file format (`.lgl` file suffix). This is yet another graph file format that tries to be as stingy as possible with space, yet keeping the edge file in a human readable (not binary) format. The format itself is like the following:

```
 # vertex1name
vertex2name [optionalWeight]
vertex3name [optionalWeight]
```

Here, the first vertex of an edge is preceded with a pound sign '#'. Then each vertex that shares an edge with that vertex is listed one per line on subsequent lines.

LGL cannot handle loop and multiple edges or directed graphs, but in **igraph** it is not an error to have multiple and loop edges.

Arguments:

graph:

Pointer to an uninitialized graph object.

instream:

A stream, it should be readable.

names:

Logical value, if TRUE the symbolic names of the vertices will be added to the graph as a vertex attribute called "name".

weights:

Whether to add the weights of the edges to the graph as an edge attribute called "weight". `IGRAPH_ADD_WEIGHTS_YES` adds the weights (even if they are not present in the file, in this case they are assumed to be zero). `IGRAPH_ADD_WEIGHTS_NO` does not add any edge attribute. `IGRAPH_ADD_WEIGHTS_IF_PRESENT` adds the attribute if and only if there is at least one explicit edge weight in the input file.

directed:

Whether to create a directed graph. As this format was originally used only for undirected graphs there is no information in the file about the directedness of the graph. Set this parameter to `IGRAPH_DIRECTED` or `IGRAPH_UNDIRECTED` to create a directed or undirected graph.

Returns:

Error code: `IGRAPH_PARSEERROR`: if there is a problem reading the file, or the file is syntactically incorrect.

Time complexity: O(|V|+|E|log(|V|)) if we neglect the time required by the parsing. As usual |V| is the number of vertices, while |E| is the number of edges.

See also:

`igraph_read_graph_ncol()`, `igraph_write_graph_lgl()`

Example 19-1. File examples/simple/igraph_read_graph_lgl.c

19.1.6. `igraph_write_graph_lgl` — Writes the graph to a file in `.lgl` format

```
int igraph_write_graph_lgl(const igraph_t *graph, FILE *outstream,
      const char *names, const char *weights,
      igraph_bool_t isolates);
```

`.lgl` is a format used by LGL, see `igraph_read_graph_lgl()` for details.

Note that having multiple or loop edges in an `.lgl` file breaks the LGL software but **igraph** does not check for this condition.

Arguments:

`graph`:

The graph to write.

`outstream`:

The stream object to write to, it should be writable.

`names`:

The name of the vertex attribute, if symbolic names are written to the file. If not supply 0 here.

`weights`:

The name of the edge attribute, if they are also written to the file. If you don't want weights supply 0 here.

`isolates`:

Logical, if TRUE isolated vertices are also written to the file. If FALSE they will be omitted.

Returns:

Error code: `IGRAPH_EFILE` if there is an error writing the file.

Time complexity: O(|E|), the number of edges if *isolates* is FALSE, O(|V|+|E|) otherwise. All file operations are expected to have time complexity O(1).

See also:

`igraph_read_graph_lgl()`, `igraph_write_graph_ncol()`

Example 19-2. File examples/simple/igraph_write_graph_lgl.c

19.1.7. `igraph_read_graph_dimacs` — Read a graph in DIMACS format.

```
int igraph_read_graph_dimacs(igraph_t *graph, FILE *instream,
        igraph_strvector_t *problem,
        igraph_vector_t *label,
        igraph_integer_t *source,
        igraph_integer_t *target,
        igraph_vector_t *capacity,
        igraph_bool_t directed);
```

This function reads the DIMACS file format, more specifically the version for network flow problems, see the files at ftp://dimacs.rutgers.edu/pub/netflow/general-info/

This is a line-oriented text file (ASCII) format. The first character of each line defines the type of the line. If the first character is `c` the line is a comment line and it is ignored. There is one problem line (`p` in the file, it must appear before any node and arc descriptor lines. The problem line has three fields separated by spaces: the problem type (`min` , `max` or `asn`), the number of vertices and number of edges in the graph. Exactly two node identification lines are expected (`n`), one for the source, one for the target vertex. These have two fields: the id of the vertex and the type of the vertex, either `s`

(=source) or `t` (=target). Arc lines start with `a` and have three fields: the source vertex, the target vertex and the edge capacity.

Vertex ids are numbered from 1.

Arguments:

`graph`:

Pointer to an uninitialized graph object.

`instream`:

The file to read from.

`source`:

Pointer to an integer, the id of the source node will be stored here. (The igraph vertex id, which is one less than the actual number in the file.) It is ignored if `NULL`.

`target`:

Pointer to an integer, the (igraph) id of the target node will be stored here. It is ignored if `NULL`.

`capacity`:

Pointer to an initialized vector, the capacity of the edges will be stored here if not `NULL`.

`directed`:

Boolean, whether to create a directed graph.

Returns:

Error code.

Time complexity: O(|V|+|E|+c), the number of vertices plus the number of edges, plus the size of the file in characters.

See also:

`igraph_write_graph_dimacs()`

19.1.8. `igraph_write_graph_dimacs` — Write a graph in DIMACS format.

```
int igraph_write_graph_dimacs(const igraph_t *graph, FILE *outstream,
          long int source, long int target,
          const igraph_vector_t *capacity);
```

This function writes a graph to an output stream in DIMACS format, describing a maximum flow problem. See ftp://dimacs.rutgers.edu/pub/netflow/general-info/

This file format is discussed in the documentation of `igraph_read_graph_dimacs()`, see that for more information.

Arguments:

`graph`:

The graph to write to the stream.

`outstream`:

The stream.

`source`:

Integer, the id of the source vertex for the maximum flow.

`target`:

Integer, the id of the target vertex.

`capacity`:

Pointer to an initialized vector containing the edge capacity values.

Returns:

Error code.

Time complexity: O(|E|), the number of edges in the graph.

See also:

igraph_read_graph_dimacs()

19.2. Binary formats

19.2.1. `igraph_read_graph_graphdb` — Read a graph in the binary graph database format.

```
int igraph_read_graph_graphdb(igraph_t *graph, FILE *instream,
         igraph_bool_t directed);
```

This is a binary format, used in the graph database for isomorphism testing (http://amalfi.dis.unina.it/graph/) From the graph database homepage (http://amalfi.dis.unina.it/graph/db/doc/graphdbat-2.html):

> The graphs are stored in a compact binary format, one graph per file. The file is composed of 16 bit words, which are represented using the so-called little-endian convention, i.e. the least significant byte of the word is stored first.
>
> Then, for each node, the file contains the list of edges coming out of the node itself. The list is represented by a word encoding its length, followed by a word for each edge, representing the destination node of the edge. Node numeration is 0-based, so the first node of the graph has index 0.

Only unlabelled graphs are implemented.

Arguments:

graph:

Pointer to an uninitialized graph object.

instream:

The stream to read from.

directed:

Logical scalar, whether to create a directed graph.

Returns:

Error code.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

Example 19-3. File examples/simple/igraph_read_graph_graphdb.c

19.3. GraphML format

19.3.1. `igraph_read_graph_graphml` — Reads a graph from a GraphML file.

```
int igraph_read_graph_graphml(igraph_t *graph, FILE *instream,
          int index);
```

GraphML is an XML-based file format for representing various types of graphs. Currently only the most basic import functionality is implemented in igraph: it can read GraphML files without nested graphs and hyperedges. Attributes of the graph are loaded only if an attribute interface is attached, ie. if you use igraph from R or Python.

Graph attribute names are taken from the `attr`.name attributes of the `key` tags in the GraphML file. Since `attr`.name is not mandatory, igraph will fall back to the `id` attribute of the `key` tag if `attr`.name is missing.

Arguments:

`graph`:

Pointer to an uninitialized graph object.

`instream`:

A stream, it should be readable.

`index`:

If the GraphML file contains more than one graph, the one specified by this index will be loaded. Indices start from zero, so supply zero here if your GraphML file contains only a single graph.

Returns:

Error code: `IGRAPH_PARSEERROR`: if there is a problem reading the file, or the file is syntactically incorrect. `IGRAPH_UNIMPLEMENTED`: the GraphML functionality was disabled at compile-time

Example 19-4. File examples/simple/graphml.c

19.3.2. `igraph_write_graph_graphml` — Writes the graph to a file in GraphML format

```
int igraph_write_graph_graphml(const igraph_t *graph, FILE *outstream);
```

GraphML is an XML-based file format for representing various types of graphs. See the GraphML Primer (http://graphml.graphdrawing.org/primer/graphml-primer.html) for detailed format description.

Arguments:

graph:

The graph to write.

outstream:

The stream object to write to, it should be writable.

Returns:

Error code: `IGRAPH_EFILE` if there is an error writing the file.

Time complexity: O(|V|+|E|) otherwise. All file operations are expected to have time complexity O(1).

Example 19-5. File examples/simple/graphml.c

19.4. GML format

19.4.1. `igraph_read_graph_gml` — Read a graph in GML format.

```
int igraph_read_graph_gml(igraph_t *graph, FILE *instream);
```

GML is a simple textual format, see http://www.infosun.fim.uni-passau.de/Graphlet/GML/ for details.

Although all syntactically correct GML can be parsed, we implement only a subset of this format, some attributes might be ignored. Here is a list of all the differences:

1. Only `node` and `edge` attributes are used, and only if they have a simple type: integer, real or string. So if an attribute is an array or a record, then it is ignored. This is also true if only some values of the attribute are complex.
2. Top level attributes except for `Version` and the first `graph` attribute are completely ignored.
3. Graph attributes except for `node` and `edge` are completely ignored.
4. There is no maximum line length.
5. There is no maximum keyword length.
6. Character entities in strings are not interpreted.
7. We allow `inf` (infinity) and `nan` (not a number) as a real number. This is case insensitive, so `nan`, `NaN` and `NAN` are equal.

Please contact us if you cannot live with these limitations of the GML parser.

Arguments:

`graph`:

Pointer to an uninitialized graph object.

`instream`:

The stream to read the GML file from.

Returns:

Error code.

Time complexity: should be proportional to the length of the file.

See also:

`igraph_read_graph_graphml()` for a more modern format, `igraph_write_graph_gml()` for writing GML files.

Example 19-6. File examples/simple/gml.c

19.4.2. `igraph_write_graph_gml` — Write the graph to a stream in GML format

```
int igraph_write_graph_gml(const igraph_t *graph, FILE *outstream,
      const igraph_vector_t *id, const char *creator);
```

GML is a quite general textual format, see http://www.infosun.fim.uni-passau.de/Graphlet/GML/ for details.

The graph, vertex and edges attributes are written to the file as well, if they are numeric of string.

As igraph is more forgiving about attribute names, it might be necessary to simplify the them before writing to the GML file. This way we'll have a syntactically correct GML file. The following simple procedure is performed on each attribute name: first the alphanumeric characters are extracted, the others are ignored. Then if the first character is not a letter then the attribute name is prefixed with "igraph". Note that this might result identical names for two attributes, igraph does not check this.

The "id" vertex attribute is treated specially. If the *id* argument is not 0 then it should be a numeric vector with the vertex ids and the "id" vertex attribute is ignored (if there is one). If *id* is 0 and there is a numeric “id” vertex attribute that is used instead. If ids are not specified in either way then the regular igraph vertex ids are used.

Note that whichever way vertex ids are specified, their uniqueness is not checked.

If the graph has edge attributes named “source” or “target” they’re silently ignored. GML uses these attributes to specify the edges, so we cannot write them to the file. Rename them before calling this function if you want to preserve them.

Arguments:

`graph`:

The graph to write to the stream.

`outstream`:

The stream to write the file to.

`id`:

Either `NULL` or a numeric vector with the vertex ids. See details above.

`creator`:

An optional string to write to the stream in the creator line. If this is 0 then the current date and time is added.

Returns:

Error code.

Time complexity: should be proportional to the number of characters written to the file.

See also:

`igraph_read_graph_gml()` for reading GML files, `igraph_read_graph_graphml()` for a more modern format.

Example 19-7. File examples/simple/gml.c

19.5. Pajek format

19.5.1. `igraph_read_graph_pajek` — Reads a file in Pajek format

```
int igraph_read_graph_pajek(igraph_t *graph, FILE *instream);
```

Arguments:

`graph`:

Pointer to an uninitialized graph object.

`file`:

An already opened file handler.

Returns:

Error code.

Only a subset of the Pajek format is implemented. This is partially because this format is not very well documented, but also because **igraph** does not support some Pajek features, like multigraphs.

Starting from version 0.6.1 igraph reads bipartite (two-mode) graphs from Pajek files and add the `type` vertex attribute for them. Warnings are given for invalid edges, i.e. edges connecting vertices of the same type.

The list of the current limitations:

1. Only `.net` files are supported, Pajek project files (`.paj`) are not. These might be supported in the future if there is need for it.
2. Time events networks are not supported.
3. Hypergraphs (ie. graphs with non-binary edges) are not supported.
4. Graphs with both directed and non-directed edges are not supported, are they cannot be represented in **igraph**.
5. Only Pajek networks are supported, permutations, hierarchies, clusters and vectors are not.

6. Graphs with multiple edge sets are not supported.

If there are attribute handlers installed, **igraph** also reads the vertex and edge attributes from the file. Most attributes are renamed to be more informative: ‘`color`’ instead of ‘`c`’, ‘`xfact`’ instead of ‘`x_fact`’, ‘`yfact`’ instead of ‘y_fact’, ‘`labeldist`’ instead of ‘`lr`’, ‘`labeldegree2`’ instead of ‘`lphi`’, ‘`framewidth`’ instead of ‘`bw`’, ‘`fontsize`’ instead of ‘`fos`’, ‘`rotation`’ instead of ‘`phi`’, ‘`radius`’ instead of ‘`r`’, ‘`diamondratio`’ instead of ‘`q`’, ‘`labeldegree`’ instead of ‘`la`’, ‘`vertexsize`’ instead of ‘`size`’, ‘`color`’ instead of ‘`ic`’, ‘`framecolor`’ instead of ‘`bc`’, ‘`labelcolor`’ instead of ‘`lc`’, these belong to vertices.

Edge attributes are also renamed, ‘`s`’ to ‘`arrowsize`’, ‘`w`’ to ‘`edgewidth`’, ‘`h1`’ to ‘`hook1`’, ‘`h2`’ to ‘`hook2`’, ‘`a1`’ to ‘`angle1`’, ‘`a2`’ to ‘`angle2`’, ‘`k1`’ to ‘`velocity1`’, ‘`k2`’ to ‘`velocity2`’, ‘`ap`’ to ‘`arrowpos`’, ‘`lp`’ to ‘`labelpos`’, ‘`lr`’ to ‘`labelangle`’, ‘`lphi`’ to ‘`labelangle2`’, ‘`la`’ to ‘`labeldegree`’, ‘`fos`’ to ‘`fontsize`’, ‘`a`’ to ‘`arrowtype`’, ‘`p`’ to ‘`linepattern`’, ‘`l`’ to ‘`label`’, ‘`lc`’ to ‘`labelcolor`’, ‘`c`’ to ‘`color`’.

In addition the following vertex attributes might be added: ‘`id`’ if there are vertex ids in the file, ‘`x`’ and ‘`y`’ or ‘`x`’ and ‘`y`’ and ‘`z`’ if there are vertex coordinates in the file.

The ‘`weight`’ edge attribute might be added if there are edge weights present.

See the pajek homepage: http://vlado.fmf.uni-lj.si/pub/networks/pajek/ for more info on Pajek and the Pajek manual: http://vlado.fmf.uni-lj.si/pub/networks/pajek/doc/pajekman.pdf for information on the Pajek file format.

Time complexity: O(|V|+|E|+|A|), |V| is the number of vertices, |E| the number of edges, |A| the number of attributes (vertex + edge) in the graph if there are attribute handlers installed.

See also:

`igraph_write_graph_pajek()` for writing Pajek files, `igraph_read_graph_graphml()` for reading GraphML files.

Example 19-8. File examples/simple/foreign.c

19.5.2. `igraph_write_graph_pajek` — Writes a graph to a file in Pajek format.

```
int igraph_write_graph_pajek(const igraph_t *graph, FILE *outstream);
```

The Pajek vertex and edge parameters (like color) are determined by the attributes of the vertices and edges, of course this requires an attribute handler to be installed. The names of the corresponding vertex and edge attributes are listed at `igraph_read_graph_pajek()`, eg. the '`color`' vertex attributes determines the color ('`c`' in Pajek) parameter.

Arguments:

`graph`:

The graph object to write.

`outstream`:

The file to write to. It should be opened and writable. Make sure that you open the file in binary format if you use MS Windows, otherwise end of line characters will be messed up. (igraph will be able to read back these messed up files, but Pajek won't.)

Returns:

Error code.

Time complexity: O(|V|+|E|+|A|), |V| is the number of vertices, |E| is the number of edges, |A| the number of attributes (vertex + edge) in the graph if there are attribute handlers installed.

See also:

`igraph_read_graph_pajek()` for reading Pajek graphs, `igraph_write_graph_graphml()` for writing a graph in GraphML format, this suites **igraph** graphs better.

Example 19-9. File examples/simple/igraph_write_graph_pajek.c

19.6. UCINET's DL file format

19.6.1. `igraph_read_graph_dl` — Read a file in the DL format of UCINET

```
int igraph_read_graph_dl(igraph_t *graph, FILE *instream,
    igraph_bool_t directed);
```

This is a simple textual file format used by UCINET. See http://www.analytictech.com/networks/dataentry.htm for examples. All the forms described here are supported by igraph. Vertex names and edge weights are also supported and they are added as attributes. (If an attribute handler is attached.)

Note the specification does not mention whether the format is case sensitive or not. For igraph DL files are case sensitive, i.e. `Larry` and `larry` are not the same.

Arguments:

`graph`:

Pointer to an uninitialized graph object.

`instream`:

The stream to read the DL file from.

`directed`:

Logical scalar, whether to create a directed file.

Returns:

Error code.

Time complexity: linear in terms of the number of edges and vertices, except for the matrix format, which is quadratic in the number of vertices.

Example 19-10. File examples/simple/igraph_read_graph_dl.c

19.7. Graphviz format

19.7.1. `igraph_write_graph_dot` — Write the graph to a stream in DOT format

```
int igraph_write_graph_dot(const igraph_t *graph, FILE* outstream);
```

DOT is the format used by the widely known GraphViz software, see http://www.graphviz.org for details. The grammar of the DOT format can be found here: http://www.graphviz.org/doc/info/lang.html

This is only a preliminary implementation, only the vertices and the edges are written but not the attributes or any visualization information.

Arguments:

`graph`:

The graph to write to the stream.

`outstream`:

The stream to write the file to.

Time complexity: should be proportional to the number of characters written to the file.

See also:

`igraph_write_graph_graphml()` for a more modern format.

Example 19-11. File examples/simple/dot.c

Chapter 20. Maximum Flows, Minimum Cuts and related measures

20.1. Maximum Flows

20.1.1. `igraph_maxflow` — Maximum network flow between a pair of vertices

```
int igraph_maxflow(const igraph_t *graph, igraph_real_t *value,
     igraph_vector_t *flow, igraph_vector_t *cut,
     igraph_vector_t *partition, igraph_vector_t *partition2,
     igraph_integer_t source, igraph_integer_t target,
     const igraph_vector_t *capacity);
```

This function implements the Goldberg-Tarjan algorithm for calculating value of the maximum flow in a directed or undirected graph. The algorithm was given in Andrew V. Goldberg, Robert E. Tarjan: A New Approach to the Maximum-Flow Problem, Journal of the ACM, 35(4), 921-940, 1988.

The input of the function is a graph, a vector of real numbers giving the capacity of the edges and two vertices of the graph, the source and the target. A flow is a function assigning positive real numbers to the edges and satisfying two requirements: (1) the flow value is less than the capacity of the edge and (2) at each vertex except the source and the target, the incoming flow (ie. the sum of the flow on the incoming edges) is the same as the outgoing flow (ie. the sum of the flow on the outgoing edges). The value of the flow is the incoming flow at the target vertex. The maximum flow is the flow with the maximum value.

Arguments:

graph:

The input graph, either directed or undirected.

value:

Pointer to a real number, the value of the maximum will be placed here, unless it is a null pointer.

flow:

If not a null pointer, then it must be a pointer to an initialized vector. The vector will be resized, and the flow on each edge will be placed in it, in the order of the edge ids. For undirected graphs this argument is bit trickier, since for these the flow direction is not predetermined by the edge direction.

For these graphs the elements of the `flow` vector can be negative, this means that the flow goes from the bigger vertex id to the smaller one. Positive values mean that the flow goes from the smaller vertex id to the bigger one.

cut:

A null pointer or a pointer to an initialized vector. If not a null pointer, then the minimum cut corresponding to the maximum flow is stored here, i.e. all edge ids that are part of the minimum cut are stored in the vector.

partition:

A null pointer or a pointer to an initialized vector. If not a null pointer, then the first partition of the minimum cut that corresponds to the maximum flow will be placed here.

partition2:

A null pointer or a pointer to an initialized vector. If not a null pointer, then the second partition of the minimum cut that corresponds to the maximum flow will be placed here.

source:

The id of the source vertex.

target:

The id of the target vertex.

capacity:

Vector containing the capacity of the edges. If NULL, then every edge is considered to have capacity 1.0.

Returns:

Error code.

Time complexity: O(|V|^3). In practice it is much faster, but i cannot prove a better lower bound for the data structure i've used. In fact, this implementation runs much faster than the `hi_pr` implementation discussed in B. V. Cherkassky and A. V. Goldberg: On implementing the push-relabel method for the maximum flow problem, (Algorithmica, 19:390--410, 1997) on all the graph classes i've tried.

See also:

`igraph_mincut_value()`, `igraph_edge_connectivity()`, `igraph_vertex_connectivity()` for properties based on the maximum flow.

Example 20-1. File examples/simple/flow.c

Example 20-2. File examples/simple/flow2.c

20.1.2. `igraph_maxflow_value` — Maximum flow in a network with the push/relabel algorithm

```
int igraph_maxflow_value(const igraph_t *graph, igraph_real_t *value,
    igraph_integer_t source, igraph_integer_t target,
    const igraph_vector_t *capacity);
```

This function implements the Goldberg-Tarjan algorithm for calculating value of the maximum flow in a directed or undirected graph. The algorithm was given in Andrew V. Goldberg, Robert E. Tarjan: A New Approach to the Maximum-Flow Problem, Journal of the ACM, 35(4), 921-940, 1988.

The input of the function is a graph, a vector of real numbers giving the capacity of the edges and two vertices of the graph, the source and the target. A flow is a function assigning positive real numbers to the edges and satisfying two requirements: (1) the flow value is less than the capacity of the edge and (2) at each vertex except the source and the target, the incoming flow (ie. the sum of the flow on the incoming edges) is the same as the outgoing flow (ie. the sum of the flow on the outgoing edges). The value of the flow is the incoming flow at the target vertex. The maximum flow is the flow with the maximum value.

According to a theorem by Ford and Fulkerson (L. R. Ford Jr. and D. R. Fulkerson. Maximal flow through a network. Canadian J. Math., 8:399-404, 1956.) the maximum flow between two vertices is the same as the minimum cut between them (also called the minimum s-t cut). So `igraph_st_mincut_value()` gives the same result in all cases as `igraph_maxflow_value`().

Note that the value of the maximum flow is the same as the minimum cut in the graph.

Arguments:

graph:

The input graph, either directed or undirected.

value:

Pointer to a real number, the result will be placed here.

source:

The id of the source vertex.

target:

The id of the target vertex.

capacity:

Vector containing the capacity of the edges. If NULL, then every edge is considered to have capacity 1.0.

Returns:

Error code.

Time complexity: O(|V|^3). In practice it is much faster, but i cannot prove a better lower bound for the data structure i've used. In fact, this implementation runs much faster than the `hi_pr` implementation discussed in B. V. Cherkassky and A. V. Goldberg: On implementing the push-relabel method for the maximum flow problem, (Algorithmica, 19:390--410, 1997) on all the graph classes i've tried.

See also:

`igraph_maxflow()` to calculate the actual flow. `igraph_mincut_value()`, `igraph_edge_connectivity()`, `igraph_vertex_connectivity()` for properties based on the maximum flow.

20.1.3. `igraph_dominator_tree` — Calculates the dominator tree of a flowgraph

```
int igraph_dominator_tree(const igraph_t *graph,
     igraph_integer_t root,
     igraph_vector_t *dom,
     igraph_t *domtree,
     igraph_vector_t *leftout,
     igraph_neimode_t mode);
```

A flowgraph is a directed graph with a distinguished start (or root) vertex r, such that for any vertex v, there is a path from r to v. A vertex v dominates another vertex w (not equal to v), if every path from r to w contains v. Vertex v is the immediate dominator or w, v=idom(w), if v dominates w and every other dominator of w dominates v. The edges {(idom(w), w)| w is not r} form a directed tree, rooted at r, called the dominator tree of the graph. Vertex v dominates vertex w if and only if v is an ancestor of w in the dominator tree.

This function implements the Lengauer-Tarjan algorithm to construct the dominator tree of a directed graph. For details please see Thomas Lengauer, Robert Endre Tarjan: A fast algorithm for finding dominators in a flowgraph, ACM Transactions on Programming Languages and Systems (TOPLAS) I/1, 121--141, 1979.

Arguments:

graph:

A directed graph. If it is not a flowgraph, and it contains some vertices not reachable from the root vertex, then these vertices will be collected in the `leftout` vector.

root:

The id of the root (or source) vertex, this will be the root of the tree.

dom:

Pointer to an initialized vector or a null pointer. If not a null pointer, then the immediate dominator of each vertex will be stored here. For vertices that are not reachable from the root, `IGRAPH_NAN` is stored here. For the root vertex itself, -1 is added.

domtree:

Pointer to an uninitialized igraph_t, or NULL. If not a null pointer, then the dominator tree is returned here. The graph contains the vertices that are unreachable from the root (if any), these will be isolates.

leftout:

Pointer to an initialized vector object, or NULL. If not NULL, then the ids of the vertices that are unreachable from the root vertex (and thus not part of the dominator tree) are stored here.

mode:

Constant, must be `IGRAPH_IN` or `IGRAPH_OUT`. If it is `IGRAPH_IN`, then all directions are considered as opposite to the original one in the input graph.

Returns:

Error code.

Time complexity: very close to O(|E|+|V|), linear in the number of edges and vertices. More precisely, it is O(|V|+|E|alpha(|E|,|V|)), where alpha(|E|,|V|) is a functional inverse of Ackermann's function.

Example 20-3. File examples/simple/dominator_tree.c

20.2. Cuts and minimum cuts

20.2.1. `igraph_st_mincut` — Minimum cut between a source and a target vertex

```
int igraph_st_mincut(const igraph_t *graph, igraph_real_t *value,
       igraph_vector_t *cut, igraph_vector_t *partition,
       igraph_vector_t *partition2,
       igraph_integer_t source, igraph_integer_t target,
       const igraph_vector_t *capacity);
```

Finds the edge set that has the smallest total capacity among all edge sets that disconnect the source and target vertices.

The calculation is performed using maximum flow techniques, by calling `igraph_maxflow()`.

Arguments:

graph:

The input graph.

value:

Pointer to a real variable, the value of the cut is stored here.

cut:

Pointer to a real vector, the edge ids that are included in the cut are stored here. This argument is ignored if it is a null pointer.

partition:

Pointer to a real vector, the vertex ids of the vertices in the first partition of the cut are stored here. This argument is ignored if it is a null pointer.

partition2:

Pointer to a real vector, the vertex ids of the vertices in the second partition of the cut are stored here. This argument is ignored if it is a null pointer.

source:

Integer, the id of the source vertex.

target:

Integer, the id of the target vertex.

capacity:

Vector containing the capacity of the edges. If a null pointer, then every edge is considered to have capacity 1.0.

Returns:

Error code.

See also:

`igraph_maxflow()`.

Time complexity: see `igraph_maxflow()`.

20.2.2. `igraph_st_mincut_value` — The minimum s-t cut in a graph

```
int igraph_st_mincut_value(const igraph_t *graph, igraph_real_t *value,
      igraph_integer_t source, igraph_integer_t target,
      const igraph_vector_t *capacity);
```

The minimum s-t cut in a weighted (=valued) graph is the total minimum edge weight needed to remove from the graph to eliminate all paths from a given vertex (`source`) to another vertex (`target`). Directed paths are considered in directed graphs, and undirected paths in undirected graphs.

The minimum s-t cut between two vertices is known to be same as the maximum flow between these two vertices. So this function calls `igraph_maxflow_value()` to do the calculation.

Arguments:

`graph`:

The input graph.

`value`:

Pointer to a real variable, the result will be stored here.

`source`:

The id of the source vertex.

`target`:

The id of the target vertex.

`capacity`:

Pointer to the capacity vector, it should contain non-negative numbers and its length should be the same the the number of edges in the graph. It can be a null pointer, then every edge has unit capacity.

Returns:

Error code.

Time complexity: O(|V|^3), see also the discussion for `igraph_maxflow_value()`, |V| is the number of vertices.

20.2.3. `igraph_all_st_cuts` — List all edge-cuts between two vertices in a directed graph

```
int igraph_all_st_cuts(const igraph_t *graph,
          igraph_vector_ptr_t *cuts,
          igraph_vector_ptr_t *partition1s,
          igraph_integer_t source,
          igraph_integer_t target);
```

This function lists all edge-cuts between a source and a target vertex. Every cut is listed exactly once. The implemented algorithm is described in JS Provan and DR Shier: A Paradigm for listing (s,t)-cuts in graphs, Algorithmica 15, 351--372, 1996.

Arguments:

`graph`:

The input graph, is must be directed.

`cuts`:

An initialized pointer vector, the cuts are stored here. It is a list of pointers to igraph_vector_t objects. Each vector will contain the ids of the edges in the cut. This argument is ignored if it is a null pointer. To free all memory allocated for `cuts`, you need call `igraph_vector_destroy()` and then `igraph_free()` on each element, before destroying the pointer vector itself.

`partition1s`:

An initialized pointer vector, the list of vertex sets, generating the actual edge cuts, are stored here. Each vector contains a set of vertex ids. If X is such a set, then all edges going from X to the complement of X form an (s,t) edge-cut in the graph. This argument is ignored if it is a null pointer. To free all memory allocated for `partition1s`, you need call `igraph_vector_destroy()` and then `igraph_free()` on each element, before destroying the pointer vector itself.

`source`:

The id of the source vertex.

`target`:

The id of the target vertex.

Returns:

Error code.

Time complexity: O(n(|V|+|E|)), where |V| is the number of vertices, |E| is the number of edges, and n is the number of cuts.

Example 20-4. File examples/simple/igraph_all_st_cuts.c

20.2.4. `igraph_all_st_mincuts` — All minimum s-t cuts of a directed graph

```
int igraph_all_st_mincuts(const igraph_t *graph, igraph_real_t *value,
     igraph_vector_ptr_t *cuts,
     igraph_vector_ptr_t *partition1s,
     igraph_integer_t source,
     igraph_integer_t target,
     const igraph_vector_t *capacity);
```

This function lists all minimum edge cuts between two vertices, in a directed graph. The implemented algorithm is described in JS Provan and DR Shier: A Paradigm for listing (s,t)-cuts in graphs, Algorithmica 15, 351--372, 1996.

Arguments:

`graph`:

The input graph, it must be directed.

`value`:

Pointer to a real number, the value of the minimum cut is stored here, unless it is a null pointer.

`cuts`:

An initialized pointer vector, the cuts are stored here. It is a list of pointers to igraph_vector_t objects. Each vector will contain the ids of the edges in the cut. This argument is ignored if it is a null pointer. To free all memory allocated for `cuts`, you need call `igraph_vector_destroy()` and then `igraph_free()` on each element, before destroying the pointer vector itself.

`partition1s`:

An initialized pointer vector, the list of vertex sets, generating the actual edge cuts, are stored here. Each vector contains a set of vertex ids. If X is such a set, then all edges going from X to the complement of X form an (s,t) edge-cut in the graph. This argument is ignored if it is a null pointer.

`source`:

The id of the source vertex.

`target`:

The id of the target vertex.

`capacity`:

Vector of edge capacities. If this is a null pointer, then all edges are assumed to have capacity one.

Returns:

Error code.

Time complexity: O(n(|V|+|E|))+O(F), where |V| is the number of vertices, |E| is the number of edges, and n is the number of cuts; O(F) is the time complexity of the maximum flow algorithm, see `igraph_maxflow()`.

Example 20-5. File examples/simple/igraph_all_st_mincuts.c

20.2.5. `igraph_mincut` — Calculates the minimum cut in a graph.

```
int igraph_mincut(const igraph_t *graph,
    igraph_real_t *value,
    igraph_vector_t *partition,
    igraph_vector_t *partition2,
    igraph_vector_t *cut,
    const igraph_vector_t *capacity);
```

This function calculates the minimum cut in a graph. The minimum cut is the minimum set of edges which needs to be removed to disconnect the graph. The minimum is calculated using the weights (*`capacity`*) of the edges, so the cut with the minimum total capacity is calculated.

For directed graphs an implementation based on calculating 2|V|-2 maximum flows is used. For undirected graphs we use the Stoer-Wagner algorithm, as described in M. Stoer and F. Wagner: A simple min-cut algorithm, Journal of the ACM, 44 585-591, 1997.

The first implementation of the actual cut calculation for undirected graphs was made by Gregory Benison, thanks Greg.

Arguments:

`graph`:

The input graph.

`value`:

Pointer to an integer, the value of the cut will be stored here.

`partition`:

Pointer to an initialized vector, the ids of the vertices in the first partition after separating the graph will be stored here. The vector will be resized as needed. This argument is ignored if it is a NULL pointer.

`partition2`:

Pointer to an initialized vector the ids of the vertices in the second partition will be stored here. The vector will be resized as needed. This argument is ignored if it is a NULL pointer.

`cut`:

Pointer to an initialized vector, the ids of the edges in the cut will be stored here. This argument is ignored if it is a NULL pointer.

`capacity`:

A numeric vector giving the capacities of the edges. If a null pointer then all edges have unit capacity.

Returns:

Error code.

See also:

`igraph_mincut_value()`, a simpler interface for calculating the value of the cut only.

Time complexity: for directed graphs it is O(|V|^4), but see the remarks at `igraph_maxflow()`. For undirected graphs it is O(|V||E|+|V|^2 log|V|). |V| and |E| are the number of vertices and edges respectively.

Example 20-6. File examples/simple/igraph_mincut.c

20.2.6. `igraph_mincut_value` — The minimum edge cut in a graph

```
int igraph_mincut_value(const igraph_t *graph, igraph_real_t *res,
   const igraph_vector_t *capacity);
```

The minimum edge cut in a graph is the total minimum weight of the edges needed to remove from the graph to make the graph *not* strongly connected. (If the original graph is not strongly connected then this is zero.) Note that in undirected graphs strong connectedness is the same as weak connectedness.

The minimum cut can be calculated with maximum flow techniques, although the current implementation does this only for directed graphs and a separate non-flow based implementation is used for undirected graphs. See Mechthild Stoer and Frank Wagner: A simple min-cut algorithm, Journal of the ACM 44 585--591, 1997. For directed graphs the maximum flow is calculated between a fixed vertex and all the other vertices in the graph and this is done in both directions. Then the minimum is taken to get the minimum cut.

Arguments:

`graph`:

The input graph.

`res`:

Pointer to a real variable, the result will be stored here.

`capacity`:

Pointer to the capacity vector, it should contain the same number of non-negative numbers as the number of edges in the graph. If a null pointer then all edges will have unit capacity.

Returns:

Error code.

See also:

`igraph_mincut()`, `igraph_maxflow_value()`, `igraph_st_mincut_value()`.

Time complexity: O(log(|V|)*|V|^2) for undirected graphs and O(|V|^4) for directed graphs, but see also the discussion at the documentation of `igraph_maxflow_value()`.

20.3. Connectivity

20.3.1. `igraph_st_edge_connectivity` — Edge connectivity of a pair of vertices

```
int igraph_st_edge_connectivity(const igraph_t *graph, igraph_integer_t *res,
    igraph_integer_t source,
    igraph_integer_t target);
```

The edge connectivity of two vertices (`source` and `target`) in a graph is the minimum number of edges that have to be deleted from the graph to eliminate all paths from `source` to `target`.

This function uses the maximum flow algorithm to calculate the edge connectivity.

Arguments:

`graph`:

The input graph, it has to be directed.

`res`:

Pointer to an integer, the result will be stored here.

`source`:

The id of the source vertex.

`target`:

The id of the target vertex.

Returns:

Error code.

Time complexity: O(|V|^3).

See also:

`igraph_maxflow_value()`, `igraph_edge_connectivity()`, `igraph_st_vertex_connectivity()`, `igraph_vertex_connectivity()`.

20.3.2. `igraph_edge_connectivity` — The minimum edge connectivity in a graph.

```
int igraph_edge_connectivity(const igraph_t *graph, igraph_integer_t *res,
        igraph_bool_t checks);
```

This is the minimum of the edge connectivity over all pairs of vertices in the graph.

The edge connectivity of a graph is the same as group adhesion as defined in Douglas R. White and Frank Harary: The cohesiveness of blocks in social networks: node connectivity and conditional density, Sociological Methodology 31:305--359, 2001.

Arguments:

`graph`:

The input graph.

`res`:

Pointer to an integer, the result will be stored here.

`checks`:

Logical constant. Whether to check that the graph is connected and also the degree of the vertices. If the graph is not (strongly) connected then the connectivity is obviously zero. Otherwise if the minimum degree is one then the edge connectivity is also one. It is a good idea to perform these checks, as they can be done quickly compared to the connectivity calculation itself. They were suggested by Peter McMahan, thanks Peter.

Returns:

Error code.

Time complexity: O(log(|V|)*|V|^2) for undirected graphs and O(|V|^4) for directed graphs, but see also the discussion at the documentation of `igraph_maxflow_value()`.

See also:

`igraph_st_edge_connectivity()`, `igraph_maxflow_value()`, `igraph_vertex_connectivity()`.

20.3.3. `igraph_st_vertex_connectivity` — The vertex connectivity of a pair of vertices

```
int igraph_st_vertex_connectivity(const igraph_t *graph,
      igraph_integer_t *res,
      igraph_integer_t source,
      igraph_integer_t target,
      igraph_vconn_nei_t neighbors);
```

The vertex connectivity of two vertices (`source` and `target`) is the minimum number of vertices that have to be deleted to eliminate all paths from `source` to `target`. Directed paths are considered in directed graphs.

The vertex connectivity of a pair is the same as the number of different (ie. node-independent) paths from source to target.

The current implementation uses maximum flow calculations to obtain the result.

Arguments:

graph:

The input graph.

res:

Pointer to an integer, the result will be stored here.

source:

The id of the source vertex.

target:

The id of the target vertex.

neighbors:

A constant giving what to do if the two vertices are connected. Possible values: `IGRAPH_VCONN_NEI_ERROR`, stop with an error message, `IGRAPH_VCONN_NEGATIVE`, return -1. `IGRAPH_VCONN_NUMBER_OF_NODES`, return the number of nodes. `IGRAPH_VCONN_IGNORE`, ignore the fact that the two vertices are connected and calculated the number of vertices needed to eliminate all paths except for the trivial (direct) paths between `source` and `vertex`. TOOD: what about neighbors?

Returns:

Error code.

Time complexity: O(|V|^3), but see the discussion at `igraph_maxflow_value()`.

See also:

`igraph_vertex_connectivity()`, `igraph_edge_connectivity()`, `igraph_maxflow_value()`.

20.3.4. `igraph_vertex_connectivity` — The vertex connectivity of a graph

```
int igraph_vertex_connectivity(const igraph_t *graph, igraph_integer_t *res,
          igraph_bool_t checks);
```

The vertex connectivity of a graph is the minimum vertex connectivity along each pairs of vertices in the graph.

The vertex connectivity of a graph is the same as group cohesion as defined in Douglas R. White and Frank Harary: The cohesiveness of blocks in social networks: node connectivity and conditional density, Sociological Methodology 31:305--359, 2001.

Arguments:

graph:

The input graph.

res:

Pointer to an integer, the result will be stored here.

checks:

Logical constant. Whether to check that the graph is connected and also the degree of the vertices. If the graph is not (strongly) connected then the connectivity is obviously zero. Otherwise if the minimum degree is one then the vertex connectivity is also one. It is a good idea to perform these checks, as they can be done quickly compared to the connectivity calculation itself. They were suggested by Peter McMahan, thanks Peter.

Returns:

Error code.

Time complexity: O(|V|^5).

See also:

`igraph_st_vertex_connectivity()`, `igraph_maxflow_value()`, and `igraph_edge_connectivity()`.

20.4. Edge- and Vertex-Disjoint Paths

20.4.1. `igraph_edge_disjoint_paths` — The maximum number of edge-disjoint paths between two vertices.

```
int igraph_edge_disjoint_paths(const igraph_t *graph, igraph_integer_t *res,
           igraph_integer_t source,
           igraph_integer_t target);
```

A set of paths between two vertices is called edge-disjoint if they do not share any edges. The maximum number of edge-disjoint paths are calculated by this function using maximum flow techniques. Directed paths are considered in directed graphs.

Note that the number of disjoint paths is the same as the edge connectivity of the two vertices using uniform edge weights.

Arguments:

`graph`:

The input graph, can be directed or undirected.

`res`:

Pointer to an integer variable, the result will be stored here.

`source`:

The id of the source vertex.

`target`:

The id of the target vertex.

Returns:

Error code.

Time complexity: O(|V|^3), but see the discussion at `igraph_maxflow_value()`.

See also:

`igraph_vertex_disjoint_paths()`, `igraph_st_edge_connectivity()`, `igraph_maxflow_value()`.

20.4.2. `igraph_vertex_disjoint_paths` — Maximum number of vertex-disjoint paths between two vertices.

```
int igraph_vertex_disjoint_paths(const igraph_t *graph, igraph_integer_t *res,
     igraph_integer_t source,
     igraph_integer_t target);
```

A set of paths between two vertices is called vertex-disjoint if they share no vertices. The calculation is performed by using maximum flow techniques.

Note that the number of vertex-disjoint paths is the same as the vertex connectivity of the two vertices in most cases (if the two vertices are not connected by an edge).

Arguments:

`graph`:

The input graph.

`res`:

Pointer to an integer variable, the result will be stored here.

`source`:

The id of the source vertex.

`target`:

The id of the target vertex.

Returns:

Error code.

Time complexity: O(|V|^3).

See also:

`igraph_edge_disjoint_paths()`, `igraph_vertex_connectivity()`, `igraph_maxflow_value()`.

20.5. Graph Adhesion and Cohesion

20.5.1. `igraph_adhesion` — Graph adhesion, this is (almost) the same as edge connectivity.

```
int igraph_adhesion(const igraph_t *graph, igraph_integer_t *res,
      igraph_bool_t checks);
```

This quantity is defined by White and Harary in The cohesiveness of blocks in social networks: node connectivity and conditional density, (Sociological Methodology 31:305--359, 2001) and basically it is the edge connectivity of the graph with uniform edge weights.

Arguments:

graph:

The input graph, either directed or undirected.

res:

Pointer to an integer, the result will be stored here.

checks:

Logical constant. Whether to check that the graph is connected and also the degree of the vertices. If the graph is not (strongly) connected then the adhesion is obviously zero. Otherwise if the minimum degree is one then the adhesion is also one. It is a good idea to perform these checks, as they can be done quickly compared to the edge connectivity calculation itself. They were suggested by Peter McMahan, thanks Peter. *

Returns:

Error code.

Time complexity: O(log(|V|)*|V|^2) for undirected graphs and O(|V|^4) for directed graphs, but see also the discussion at the documentation of `igraph_maxflow_value()`.

See also:

`igraph_cohesion()`, `igraph_maxflow_value()`, `igraph_edge_connectivity()`, `igraph_mincut_value()`.

20.5.2. `igraph_cohesion` — Graph cohesion, this is the same as vertex connectivity.

```
int igraph_cohesion(const igraph_t *graph, igraph_integer_t *res,
      igraph_bool_t checks);
```

This quantity was defined by White and Harary in "The cohesiveness of blocks in social networks: node connectivity and conditional density", (Sociological Methodology 31:305--359, 2001) and it is the same as the vertex connectivity of a graph.

Arguments:

graph:

The input graph.

res:

Pointer to an integer variable, the result will be stored here.

checks:

Logical constant. Whether to check that the graph is connected and also the degree of the vertices. If the graph is not (strongly) connected then the cohesion is obviously zero. Otherwise if the minimum degree is one then the cohesion is also one. It is a good idea to perform these checks, as they can be done quickly compared to the vertex connectivity calculation itself. They were suggested by Peter McMahan, thanks Peter.

Returns:

Error code.

Time complexity: O(|V|^4), |V| is the number of vertices. In practice it is more like O(|V|^2), see `igraph_maxflow_value()`.

See also:

`igraph_vertex_connectivity()`, `igraph_adhesion()`, `igraph_maxflow_value()`.

20.6. Cohesive Blocks

20.6.1. `igraph_cohesive_blocks` — Identifies the hierarchical cohesive block structure of a graph

```
int igraph_cohesive_blocks(const igraph_t *graph,
      igraph_vector_ptr_t *blocks,
      igraph_vector_t *cohesion,
      igraph_vector_t *parent,
      igraph_t *block_tree);
```

Cohesive blocking is a method of determining hierarchical subsets of graph vertices based on their structural cohesion (or vertex connectivity). For a given graph G, a subset of its vertices S is said to be maximally k-cohesive if there is no superset of S with vertex connectivity greater than or equal to k. Cohesive blocking is a process through which, given a k-cohesive set of vertices, maximally l-cohesive subsets are recursively identified with l>k. Thus a hiearchy of vertex subsets is found, whith the entire graph G at its root. See the following reference for details: J. Moody and D. R. White. Structural cohesion and embeddedness: A hierarchical concept of social groups. American Sociological Review, 68(1):103--127, Feb 2003.

This function implements cohesive blocking and calculates the complete cohesive block hierarchy of a graph.

Arguments:

`graph`:

The input graph. It must be undirected and simple. See `igraph_is_simple()`.

`blocks`:

If not a null pointer, then it must be an initialized vector of pointers and the cohesive blocks are stored here. Each block is encoded with a numeric vector, that contains the vertex ids of the block.

`cohesion`:

If not a null pointer, then it must be an initialized vector and the cohesion of the blocks is stored here, in the same order as the blocks in the *`blocks`* pointer vector.

`parent`:

If not a null pointer, then it must be an initialized vector and the block hierarchy is stored here. For each block, the id (i.e. the position in the *`blocks`* pointer vector) of its parent block is stored. For the top block in the hierarchy, -1 is stored.

`block_tree`:

If not a null pointer, then it must be a pointer to an uninitialized graph, and the block hierarchy is stored here as an igraph graph. The vertex ids correspond to the order of the blocks in the `blocks` vector.

Returns:

Error code.

Time complexity: TODO.

Example 20-7. File examples/simple/cohesive_blocks.c

Chapter 21. Vertex separators

21.1. `igraph_is_separator` — Decides whether the removal of a set of vertices disconnects the graph

```
int igraph_is_separator(const igraph_t *graph,
   const igraph_vs_t candidate,
   igraph_bool_t *res);
```

Arguments:

graph:

The input graph. It may be directed, but edge directions are ignored.

condidate:

The candidate separator. It must not contain all vertices.

res:

Pointer to a boolean variable, the result is stored here.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number vertices and edges.

Example 21-1. File examples/simple/igraph_is_separator.c

21.2. `igraph_is_minimal_separator` — Decides whether a set of vertices is a minimal separator

```
int igraph_is_minimal_separator(const igraph_t *graph,
    const igraph_vs_t candidate,
    igraph_bool_t *res);
```

A set of vertices is a minimal separator, if the removal of the vertices disconnects the graph, and this is not true for any subset of the set.

This implementation first checks that the given candidate is a separator, by calling `igraph_is_separator()`. If it is a separator, then it checks that each subset of size n-1, where n is the size of the candidate, is not a separator.

Arguments:

graph:

The input graph. It may be directed, but edge directions are ignored.

candidate:

Pointer to a vector of long integers, the candidate minimal separator.

res:

Pointer to a boolean variable, the result is stored here.

Returns:

Error code.

Time complexity: O(n(|V|+|E|)), |V| is the number of vertices, |E| is the number of edges, n is the number vertices in the candidate separator.

Example 21-2. File examples/simple/igraph_is_minimal_separator.c

21.3. `igraph_all_minimal_st_separators` — List all vertex sets that are minimal (s,t) separators for some s and t

```
int igraph_all_minimal_st_separators(const igraph_t *graph,
         igraph_vector_ptr_t *separators);
```

This function lists all vertex sets that are minimal (s,t) separators for some (s,t) vertex pair.

See more about the implemented algorithm in Anne Berry, Jean-Paul Bordat and Olivier Cogis: Generating All the Minimal Separators of a Graph, In: Peter Widmayer, Gabriele Neyer and Stephan Eidenbenz (editors): Graph-theoretic concepts in computer science, 1665, 167--172, 1999. Springer.

Arguments:

`graph`:

The input graph. It may be directed, but edge directions are ignored.

`separators`:

An initialized pointer vector, the separators are stored here. It is a list of pointers to igraph_vector_t objects. Each vector will contain the ids of the vertices in the separator. To free all memory allocated for `separators`, you need call `igraph_vector_destroy()` and then `igraph_free()` on each element, before destroying the pointer vector itself.

Returns:

Error code.

Time complexity: O(n|V|^3), |V| is the number of vertices, n is the number of separators.

Example 21-3. File examples/simple/igraph_minimal_separators.c

21.4. `igraph_minimum_size_separators` — Find all minimum size separating vertex sets

```
int igraph_minimum_size_separators(const igraph_t *graph,
        igraph_vector_ptr_t *separators);
```

This function lists all separator vertex sets of minimum size. A vertex set is a separator if its removal disconnects the graph.

The implementation is based on the following paper: Arkady Kanevsky: Finding all minimum-size separating vertex sets in a graph, Networks 23, 533--541, 1993.

Arguments:

`graph`:

The input graph, it may be directed, but edge directions will be ignored.

`separators`:

An initialized pointer vector, the separators are stored here. It is a list of pointers to igraph_vector_t objects. Each vector will contain the ids of the vertices in the separator. To free all memory allocated for `separators`, you need call `igraph_vector_destroy()` and then `igraph_free()` on each element, before destroying the pointer vector itself.

Returns:

Error code.

Time complexity: TODO.

Example 21-4. File examples/simple/igraph_minimum_size_separators.c

Chapter 22. Detecting Community Structure

22.1. Common functions related to community structure

22.1.1. `igraph_modularity` — Calculate the modularity of a graph with respect to some vertex types

```
int igraph_modularity(const igraph_t *graph,
        const igraph_vector_t *membership,
        igraph_real_t *modularity,
      const igraph_vector_t *weights);
```

The modularity of a graph with respect to some division (or vertex types) measures how good the division is, or how separated are the different vertex types from each other. It is defined as Q=1/(2m) * sum(Aij-ki*kj/(2m)delta(ci,cj),i,j), here 'm' is the number of edges, 'Aij' is the element of the 'A' adjacency matrix in row 'i' and column 'j', 'ki' is the degree of 'i', 'kj' is the degree of 'j', 'ci' is the type (or component) of 'i', 'cj' that of 'j', the sum goes over all 'i' and 'j' pairs of vertices, and 'delta(x,y)' is one if x=y and zero otherwise.

Modularity on weighted graphs is also meaningful. When taking edge weights into account, 'Aij' becomes the weight of the corresponding edge (or 0 if there is no edge), 'ki' is the total weight of edges incident on vertex 'i', 'kj' is the total weight of edges incident on vertex 'j' and 'm' is the total weight of all edges.

See also MEJ Newman and M Girvan: Finding and evaluating community structure in networks. Physical Review E 69 026113, 2004.

Arguments:

`graph`:

The input graph.

`membership`:

Numeric vector which gives the type of each vertex, ie. the component to which it belongs. It does not have to be consecutive, i.e. empty communities are allowed.

`modularity`:

Pointer to a real number, the result will be stored here.

weights:

Weight vector or NULL if no weights are specified.

Returns:

Error code.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

22.1.2. `igraph_community_optimal_modularity` — Calculate the community structure with the highest modularity value

```
int igraph_community_optimal_modularity(const igraph_t *graph,
     igraph_real_t *modularity,
     igraph_vector_t *membership,
     igraph_bool_t verbose);
```

This function calculates the optimal community structure for a graph, in terms of maximal modularity score.

The calculation is done by transforming the modularity maximization into an integer programming problem, and then calling the GLPK library to solve that. Please see Ulrik Brandes et al.: On Modularity Clustering, IEEE Transactions on Knowledge and Data Engineering 20(2):172-188, 2008.

Note that modularity optimization is an NP-complete problem, and all known algorithms for it have exponential time complexity. This means that you probably don't want to run this function on larger graphs. Graphs with up to fifty vertices should be fine, graphs with a couple of hundred vertices might be possible.

Arguments:

graph:

The input graph. It is always treated as undirected.

modularity:

Pointer to a real number, or a null pointer. If it is not a null pointer, then a optimal modularity value is returned here.

membership:

Pointer to a vector, or a null pointer. If not a null pointer, then the membership vector of the optimal community structure is stored here.

Returns:

Error code.

See also:

`igraph_modularity()`, `igraph_community_fastgreedy()` for an algorithm that finds a local optimum in a greedy way.

Time complexity: exponential in the number of vertices.

Example 22-1. File examples/simple/igraph_community_optimal_modularity.c

22.1.3. `igraph_community_to_membership` — Create membership vector from community structure dendrogram

```
int igraph_community_to_membership(const igraph_matrix_t *merges,
       igraph_integer_t nodes,
       igraph_integer_t steps,
       igraph_vector_t *membership,
       igraph_vector_t *csize);
```

This function creates a membership vector from a community structure dendrogram. A membership vector contains for each vertex the id of its graph component, the graph components are numbered from zero, see the same argument of `igraph_clusters()` for an example of a membership vector.

Many community detection algorithms return with a *merges* matrix, `igraph_community_walktrap()` and `igraph_community_edge_betweenness()` are two examples. The matrix contains the merge operations performed while mapping the hierarchical structure of a network. If the matrix has `n-1` rows, where `n` is the number of vertices in the graph, then it contains the hierarchical structure of the whole network and it is called a dendrogram.

This function performs *`steps`* merge operations as prescribed by the *`merges`* matrix and returns the current state of the network.

If *`merges`* is not a complete dendrogram, it is possible to take *`steps`* steps if *`steps`* is not bigger than the number lines in *`merges`*.

Arguments:

`merges`:

The two-column matrix containing the merge operations. See `igraph_community_walktrap()` for the detailed syntax.

`nodes`:

The number of leaf nodes in the dendrogram

`steps`:

Integer constant, the number of steps to take.

`membership`:

Pointer to an initialized vector, the membership results will be stored here, if not NULL. The vector will be resized as needed.

`csize`:

Pointer to an initialized vector, or NULL. If not NULL then the sizes of the components will be stored here, the vector will be resized as needed.

See also:

`igraph_community_walktrap()`, `igraph_community_edge_betweenness()`, `igraph_community_fastgreedy()` for community structure detection algorithms.

Time complexity: O(|V|), the number of vertices in the graph.

22.1.4. `igraph_reindex_membership` — Makes the IDs in a

membership vector continuous

```
int igraph_reindex_membership(igraph_vector_t *membership,
                              igraph_vector_t *new_to_old);
```

This function reindexes component IDs in a membership vector in a way that the new IDs start from zero and go up to C-1, where C is the number of unique component IDs in the original vector. This function was contributed by Tom Gregorovic.

Arguments:

membership:

Numeric vector which gives the type of each vertex, ie. the component to which it belongs. The vector will be altered in-place.

new_to_old:

Pointer to a vector which will contain the old component ID for each new one, or NULL, in which case it is not returned. The vector will be resized as needed.

Time complexity: should be O(n log n) for n elements.

22.1.5. `igraph_compare_communities` — Compares community structures using various metrics

```
int igraph_compare_communities(const igraph_vector_t *comm1,
    const igraph_vector_t *comm2, igraph_real_t* result,
    igraph_community_comparison_t method);
```

This function assesses the distance between two community structures using the variation of information (VI) metric of Meila (2003), the normalized mutual information (NMI) of Danon et al (2005), the split-join distance of van Dongen (2000), the Rand index of Rand (1971) or the adjusted Rand index of Hubert and Arabie (1985).

References:

Meila M: Comparing clusterings by the variation of information. In: Schölkopf B, Warmuth MK (eds.). Learning Theory and Kernel Machines: 16th Annual Conference on Computational Learning Theory and

7th Kernel Workshop, COLT/Kernel 2003, Washington, DC, USA. Lecture Notes in Computer Science, vol. 2777, Springer, 2003. ISBN: 978-3-540-40720-1.

Danon L, Diaz-Guilera A, Duch J, Arenas A: Comparing community structure identification. J Stat Mech P09008, 2005.

van Dongen S: Performance criteria for graph clustering and Markov cluster experiments. Technical Report INS-R0012, National Research Institute for Mathematics and Computer Science in the Netherlands, Amsterdam, May 2000.

Rand WM: Objective criteria for the evaluation of clustering methods. J Am Stat Assoc 66(336):846-850, 1971.

Hubert L and Arabie P: Comparing partitions. Journal of Classification 2:193-218, 1985.

Arguments:

comm1:

the membership vector of the first community structure

comm2:

the membership vector of the second community structure

result:

the result is stored here.

method:

the comparison method to use. `IGRAPH_COMMCMP_VI` selects the variation of information (VI) metric of Meila (2003), `IGRAPH_COMMCMP_NMI` selects the normalized mutual information measure proposed by Danon et al (2005), `IGRAPH_COMMCMP_SPLIT_JOIN` selects the split-join distance of van Dongen (2000), `IGRAPH_COMMCMP_RAND` selects the unadjusted Rand index (1971) and `IGRAPH_COMMCMP_ADJUSTED_RAND` selects the adjusted Rand index.

Returns:

Error code.

Time complexity: O(n log(n)).

22.1.6. `igraph_split_join_distance` — Calculates the split-join distance of two community structures

```
int igraph_split_join_distance(const igraph_vector_t *comm1,
    const igraph_vector_t *comm2, igraph_integer_t *distance12,
    igraph_integer_t *distance21);
```

The split-join distance between partitions A and B is the sum of the projection distance of A from B and the projection distance of B from A. The projection distance is an asymmetric measure and it is defined as follows:

First, each set in partition A is evaluated against all sets in partition B. For each set in partition A, the best matching set in partition B is found and the overlap size is calculated. (Matching is quantified by the size of the overlap between the two sets). Then, the maximal overlap sizes for each set in A are summed together and subtracted from the number of elements in A.

The split-join distance will be returned in two arguments, `distance12` will contain the projection distance of the first partition from the second, while `distance21` will be the projection distance of the second partition from the first. This makes it easier to detect whether a partition is a subpartition of the other, since in this case, the corresponding distance will be zero.

Reference:

van Dongen S: Performance criteria for graph clustering and Markov cluster experiments. Technical Report INS-R0012, National Research Institute for Mathematics and Computer Science in the Netherlands, Amsterdam, May 2000.

Arguments:

`comm1`:

the membership vector of the first community structure

`comm2`:

the membership vector of the second community structure

`distance12`:

pointer to an `igraph_integer_t`, the projection distance of the first community structure from the second one will be returned here.

distance21:

pointer to an `igraph_integer_t`, the projection distance of the second community structure from the first one will be returned here.

Returns:

Error code.

\see `igraph_compare_communities()` with the `IGRAPH_COMMCMP_SPLIT_JOIN` method if you are not interested in the individual distances but only the sum of them. Time complexity: O(n log(n)).

22.2. Community structure based on statistical mechanics

22.2.1. `igraph_community_spinglass` — Community detection based on statistical mechanics

```
int igraph_community_spinglass(const igraph_t *graph,
          const igraph_vector_t *weights,
          igraph_real_t *modularity,
          igraph_real_t *temperature,
          igraph_vector_t *membership,
          igraph_vector_t *csize,
          igraph_integer_t spins,
          igraph_bool_t parupdate,
          igraph_real_t starttemp,
          igraph_real_t stoptemp,
          igraph_real_t coolfact,
          igraph_spincomm_update_t update_rule,
          igraph_real_t gamma,
          /* the rest is for the NegSpin implementation */
          igraph_spinglass_implementation_t implementation,
/*           igraph_matrix_t *adhesion, */
/*           igraph_matrix_t *normalised_adhesion, */
/*           igraph_real_t *polarization, */
          igraph_real_t gamma_minus);
```

This function implements the community structure detection algorithm proposed by Joerg Reichardt and Stefan Bornholdt. The algorithm is described in their paper: Statistical Mechanics of Community Detection, http://arxiv.org/abs/cond-mat/0603718.

From version 0.6 igraph also supports an extension to the algorithm that allows negative edge weights. This is described in V.A. Traag and Jeroen Bruggeman: Community detection in networks with positive and negative links, http://arxiv.org/abs/0811.2329.

Arguments:

`graph`:

The input graph, it may be directed but the direction of the edge is not used in the algorithm.

`weights`:

The vector giving the edge weights, it may be `NULL`, in which case all edges are weighted equally. Edge weights should be positive, altough this is not tested.

`modularity`:

Pointer to a real number, if not `NULL` then the modularity score of the solution will be stored here. This is the gereralized modularity that simplifies to the one defined in M. E. J. Newman and M. Girvan, Phys. Rev. E 69, 026113 (2004), if the gamma parameter is one.

`temperature`:

Pointer to a real number, if not `NULL` then the temperature at the end of the algorithm will be stored here.

`membership`:

Pointer to an initialized vector or `NULL`. If not `NULL` then the result of the clustering will be stored here, for each vertex the number of its cluster is given, the first cluster is numbered zero. The vector will be resized as needed.

`csize`:

Pointer to an initialized vector or `NULL`. If not `NULL` then the sizes of the clusters will stored here in cluster number order. The vector will be resized as needed.

`spins`:

Integer giving the number of spins, ie. the maximum number of clusters. Usually it is not a program to give a high number here, the default was 25 in the original code. Even if the number of spins is high the number of clusters in the result might small.

`parupdate`:

A logical constant, whether to update all spins in parallel. The default for this argument was `FALSE` (ie. 0) in the original code.

`starttemp`:

Real number, the temperature at the start. The value of this argument was 1.0 in the original code.

stoptemp:

Real number, the algorithm stops at this temperature. The default was 0.01 in the original code.

coolfact:

Real number, the coolinf factor for the simulated annealing. The default was 0.99 in the original code.

update_rule:

The type of the update rule. Possible values: `IGRAPH_SPINCOMM_UPDATE_SIMPLE` and `IGRAPH_SPINCOMM_UPDATE_CONFIG`. Basically this parameter defined the null model based on which the actual clustering is done. If this is `IGRAPH_SPINCOMM_UPDATE_SIMPLE` then the random graph (ie. G(n,p)), if it is `IGRAPH_SPINCOMM_UPDATE` then the configuration model is used. The configuration means that the baseline for the clustering is a random graph with the same degree distribution as the input graph.

gamma:

Real number. The gamma parameter of the algorithm. This defined the weight of the missing and existing links in the quality function for the clustering. The default value in the original code was 1.0, which is equal weight to missing and existing edges. Smaller values make the existing links contibute more to the energy function which is minimized in the algorithm. Bigger values make the missing links more important. (If my understanding is correct.)

implementation:

Constant, chooses between the two implementations of the spin-glass algorithm that are included in igraph. `IGRAPH_SPINCOMM_IMP_ORIG` selects the original implementation, this is faster, `IGRAPH_SPINCOMM_INP_NEG` selects a new implementation by Vincent Traag that allows negative edge weights.

gamma_minus:

Real number. Parameter for the `IGRAPH_SPINCOMM_IMP_NEG` implementation. This specifies the balance between the importance of present and non-present negative weighted edges in a community. Smaller values of *gamma_minus* lead to communities with lesser negative intra-connectivity. If this argument is set to zero, the algorithm reduces to a graph coloring algorithm, using the number of spins as the number of colors.

Returns:

Error code.

See also:

igraph_community_spinglass_single() for calculating the community of a single vertex.

Time complexity: TODO.

Example 22-2. File examples/simple/spinglass.c

22.2.2. `igraph_community_spinglass_single` — Community of a single node based on statistical mechanics

```
int igraph_community_spinglass_single(const igraph_t *graph,
          const igraph_vector_t *weights,
          igraph_integer_t vertex,
          igraph_vector_t *community,
          igraph_real_t *cohesion,
          igraph_real_t *adhesion,
          igraph_integer_t *inner_links,
          igraph_integer_t *outer_links,
          igraph_integer_t spins,
          igraph_spincomm_update_t update_rule,
          igraph_real_t gamma);
```

This function implements the community structure detection algorithm proposed by Joerg Reichardt and Stefan Bornholdt. It is described in their paper: Statistical Mechanics of Community Detection, http://arxiv.org/abs/cond-mat/0603718.

This function calculates the community of a single vertex without calculating all the communities in the graph.

Arguments:

graph:

The input graph, it may be directed but the direction of the edges is not used in the algorithm.

weights:

Pointer to a vector with the weights of the edges. Alternatively `NULL` can be supplied to have the same weight for every edge.

vertex:

The vertex id of the vertex of which ths community is calculated.

community:

Pointer to an initialized vector, the result, the ids of the vertices in the community of the input vertex will be stored here. The vector will be resized as needed.

cohesion:

Pointer to a real variable, if not `NULL` the cohesion index of the community will be stored here.

adhesion:

Pointer to a real variable, if not `NULL` the adhesion index of the community will be stored here.

inner_links:

Pointer to an integer, if not `NULL` the number of edges within the community is stored here.

outer_links:

Pointer to an integer, if not `NULL` the number of edges between the community and the rest of the graph will be stored here.

spins:

The number of spins to use, this can be higher than the actual number of clusters in the network, in which case some clusters will contain zero vertices.

update_rule:

The type of the update rule. Possible values: `IGRAPH_SPINCOMM_UPDATE_SIMPLE` and `IGRAPH_SPINCOMM_UPDATE_CONFIG`. Basically this parameter defined the null model based on which the actual clustering is done. If this is `IGRAPH_SPINCOMM_UPDATE_SIMPLE` then the random graph (ie. G(n,p)), if it is `IGRAPH_SPINCOMM_UPDATE` then the configuration model is used. The configuration means that the baseline for the clustering is a random graph with the same degree distribution as the input graph.

gamma:

Real number. The gamma parameter of the algorithm. This defined the weight of the missing and existing links in the quality function for the clustering. The default value in the original code was 1.0, which is equal weight to missing and existing edges. Smaller values make the existing links contibute more to the energy function which is minimized in the algorithm. Bigger values make the missing links more important. (If my understanding is correct.)

Returns:

Error code.

See also:

igraph_community_spinglass() for the traditional version of the algorithm.

Time complexity: TODO.

22.3. Community structure based on eigenvectors of matrices

The function documented in these section implements the “leading eigenvector” method developed by Mark Newman and published in MEJ Newman: Finding community structure using the eigenvectors of matrices, Phys Rev E 74:036104 (2006).

The heart of the method is the definition of the modularity matrix, B, which is B=A-P, A being the adjacency matrix of the (undirected) network, and P contains the probability that certain edges are present according to the “configuration model” In other words, a Pij element of P is the probability that there is an edge between vertices i and j in a random network in which the degrees of all vertices are the same as in the input graph.

The leading eigenvector method works by calculating the eigenvector of the modularity matrix for the largest positive eigenvalue and then separating vertices into two community based on the sign of the corresponding element in the eigenvector. If all elements in the eigenvector are of the same sign that means that the network has no underlying community structure. Check Newman’s paper to understand why this is a good method for detecting community structure.

The leading eigenvector community structure detection method is implemented in `igraph_community_leading_eigenvector()`. After the initial split, the following splits are done in a way to optimize modularity regarding to the original network.

Example 22-3. File examples/simple/igraph_community_leading_eigenvector.c

22.3.1. `igraph_community_leading_eigenvector` — Leading

eigenvector community finding (proper version).

```
int igraph_community_leading_eigenvector(const igraph_t *graph,
 igraph_matrix_t *merges,
 igraph_vector_t *membership,
 igraph_integer_t steps,
 igraph_arpack_options_t *options,
 igraph_real_t *modularity,
 igraph_bool_t start,
 igraph_vector_t *eigenvalues,
 igraph_vector_ptr_t *eigenvectors,
 igraph_vector_t *history,
        igraph_community_leading_eigenvector_callback_t *callback,
        void *callback_extra);
```

Newman's leading eigenvector method for detecting community structure. This is the proper implementation of the recursive, divisive algorithm: each split is done by maximizing the modularity regarding the original network, see MEJ Newman: Finding community structure in networks using the eigenvectors of matrices, Phys Rev E 74:036104 (2006).

Arguments:

graph:

The undirected input graph.

merges:

The result of the algorithm, a matrix containing the information about the splits performed. The matrix is built in the opposite way however, it is like the result of an agglomerative algorithm. If at the end of the algorithm (after *steps* steps was done) there are "p" communities, then these are numbered from zero to "p-1". The first line of the matrix contains the first "merge" (which is in reality the last split) of two communities into community "p", the merge in the second line forms community "p+1", etc. The matrix should be initialized before calling and will be resized as needed. This argument is ignored of it is `NULL`.

membership:

The membership of the vertices after all the splits were performed will be stored here. The vector must be initialized before calling and will be resized as needed. This argument is ignored if it is `NULL`. This argument can also be used to supply a starting configuration for the community finding, in the format of a membership vector. In this case the *start* argument must be set to 1.

steps:

The maximum number of steps to perform. It might happen that some component (or the whole network) has no underlying community structure and no further steps can be done. If you want as many steps as possible then supply the number of vertices in the network here.

options:

The options for ARPACK. `n` is always overwritten. `ncv` is set to at least 4.

modularity:

If not a null pointer, then it must be a pointer to a real number and the modularity score of the final division is stored here.

start:

Boolean, whether to use the community structure given in the *membership* argument as a starting point.

eigenvalues:

Pointer to an initialized vector or a null pointer. If not a null pointer, then the eigenvalues calculated along the community structure detection are stored here. The non-positive eigenvalues, that do not result a split, are stored as well.

eigenvectors:

If not a null pointer, then the eigenvectors that are calculated in each step of the algorithm, are stored here, in a pointer vector. Each eigenvector is stored in an `igraph_vector_t` object. The user is responsible of deallocating the memory that belongs to the individual vectors, by calling first `igraph_vector_destroy()`, and then `free()` on them.

history:

Pointer to an initialized vector or a null pointer. If not a null pointer, then a trace of the algorithm is stored here, encoded numerically. The various operations:

`IGRAPH_LEVC_HIST_START_FULL`

Start the algorithm from an initial state where each connected component is a separate community.

`IGRAPH_LEVC_HIST_START_GIVEN`

Start the algorithm from a given community structure. The next value in the vector contains the initial number of communities.

`IGRAPH_LEVC_HIST_SPLIT`

Split a community into two communities. The id of the splitted community is given in the next element of the history vector. The id of the first new community is the same as the id of the splitted community. The id of the second community equals to the number of communities before the split.

`IGRAPH_LEVC_HIST_FAILED`

Tried to split a community, but it was not worth it, as it does not result in a bigger modularity value. The id of the community is given in the next element of the vector.

callback:

A null pointer or a function of type `igraph_community_leading_eigenvector_callback_t`. If given, this callback function is called after each eigenvector/eigenvalue calculation. If the callback returns a non-zero value, then the community finding algorithm stops. See the arguments passed to the callback at the documentation of `igraph_community_leading_eigenvector_callback_t`.

callback_extra:

Extra argument to pass to the callback function.

Returns:

Error code.

See also:

`igraph_community_walktrap()` and `igraph_community_spinglass()` for other community structure detection methods.

Time complexity: O(|E|+|V|^2*steps), |V| is the number of vertices, |E| the number of edges, “steps” the number of splits performed.

22.3.2. `igraph_community_leading_eigenvector_callback_t` — Callback for the leading eigenvector community finding method.

```
typedef int igraph_community_leading_eigenvector_callback_t(
 const igraph_vector_t *membership,
 long int comm,
 igraph_real_t eigenvalue,
 const igraph_vector_t *eigenvector,
 igraph_arpack_function_t *arpack_multiplier,
        void *arpack_extra,
        void *extra);
```

The leading eigenvector community finding implementation in igraph is able to call a callback function, after each eigenvalue calculation. This callback function must be of `igraph_community_leading_eigenvector_callback_t` type. The following arguments are passed to the callback:

Arguments:

`membership`:

The actual membership vector, before recording the potential change implied by the newly found eigenvalue.

`comm`:

The id of the community that the algorithm tried to split in the last iteration. The community ids are indexed from zero here!

`eigenvalue`:

The eigenvalue the algorithm has just found.

`eigenvector`:

The eigenvector corresponding to the eigenvalue the algorithm just found.

`arpack_multiplier`:

A function that was passed to `igraph_arpack_rssolve()` to solve the last eigenproblem.

`arpack_extra`:

The extra argument that was passed to the ARPACK solver.

`extra`:

Extra argument that as passed to `igraph_community_leading_eigenvector()`.

See also:

`igraph_community_leading_eigenvector()`, `igraph_arpack_function_t`, `igraph_arpack_rssolve()`.

22.3.3. `igraph_le_community_to_membership` — Vertex membership from the leading eigenvector community structure

```
int igraph_le_community_to_membership(const igraph_matrix_t *merges,
          igraph_integer_t steps,
          igraph_vector_t *membership,
```

```
igraph_vector_t *csize);
```

This function creates a membership vector from the result of `igraph_community_leading_eigenvector()`, It takes `membership` and performs `steps` merges, according to the supplied `merges` matrix.

Arguments:

merges:

The matrix defining the merges to make. This is usually from the output of the leading eigenvector community structure detection routines.

steps:

The number of steps to make according to `merges`.

membership:

Initially the starting membership vector, on output the resulting membership vector, after performing `steps` merges.

csize:

Optionally the sizes of the communities is stored here, if this is not a null pointer, but an initialized vector.

Returns:

Error code.

Time complexity: O(|V|), the number of vertices.

22.4. Walktrap: community structure based on random walks

22.4.1. `igraph_community_walktrap` — This function is the implementation of the Walktrap community

```
int igraph_community_walktrap(const igraph_t *graph,
```

```
const igraph_vector_t *weights,
int steps,
igraph_matrix_t *merges,
igraph_vector_t *modularity,
igraph_vector_t *membership);
```

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

Currently the original C++ implementation is used in igraph, see http://www.liafa.jussieu.fr/~pons/index.php?item=prog&item2=walktrap&lang=en I'm grateful to Matthieu Latapy and Pascal Pons for providing this source code.

In contrast to the original implementation, isolated vertices are allowed in the graph and they are assumed to have a single incident loop edge with weight 1.

Arguments:

`graph`:

The input graph, edge directions are ignored.

`weights`:

Numeric vector giving the weights of the edges. If it is a NULL pointer then all edges will have equal weights. The weights are expected to be positive.

`steps`:

Integer constant, the length of the random walks.

`merges`:

Pointer to a matrix, the merges performed by the algorithm will be stored here (if not NULL). Each merge is a row in a two-column matrix and contains the ids of the merged clusters. Clusters are numbered from zero and cluster numbers smaller than the number of nodes in the network belong to the individual vertices as singleton clusters. In each step a new cluster is created from two other clusters and its id will be one larger than the largest cluster id so far. This means that before the first merge we have `n` clusters (the number of vertices in the graph) numbered from zero to `n-1`. The first merge creates cluster `n`, the second cluster `n`+1, etc.

`modularity`:

Pointer to a vector. If not NULL then the modularity score of the current clustering is stored here after each merge operation.

membership:

Pointer to a vector. If not a NULL pointer, then the membership vector corresponding to the maximal modularity score is stored here. If it is not a NULL pointer, then neither *modularity* nor *merges* may be NULL.

Returns:

Error code.

See also:

`igraph_community_spinglass()`, `igraph_community_edge_betweenness()`.

Time complexity: O(|E||V|^2) in the worst case, O(|V|^2 log|V|) typically, |V| is the number of vertices, |E| is the number of edges.

Example 22-4. File examples/simple/walktrap.c

22.5. Edge betweenness based community detection

22.5.1. `igraph_community_edge_betweenness` — Community finding based on edge betweenness

```
int igraph_community_edge_betweenness(const igraph_t *graph,
          igraph_vector_t *result,
          igraph_vector_t *edge_betweenness,
          igraph_matrix_t *merges,
          igraph_vector_t *bridges,
          igraph_vector_t *modularity,
          igraph_vector_t *membership,
          igraph_bool_t directed,
          const igraph_vector_t *weights);
```

Community structure detection based on the betweenness of the edges in the network. The algorithm was invented by M. Girvan and M. Newman, see: M. Girvan and M. E. J. Newman: Community structure in social and biological networks, Proc. Nat. Acad. Sci. USA 99, 7821-7826 (2002).

The idea is that the betweenness of the edges connecting two communities is typically high, as many of the shortest paths between nodes in separate communities go through them. So we gradually remove the edge with highest betweenness from the network, and recalculate edge betweenness after every removal. This way sooner or later the network falls off to two components, then after a while one of these components falls off to two smaller components, etc. until all edges are removed. This is a divisive hierarchical approach, the result is a dendrogram.

Arguments:

`graph`:

The input graph.

`result`:

Pointer to an initialized vector, the result will be stored here, the ids of the removed edges in the order of their removal. It will be resized as needed. It may be NULL if the edge IDs are not needed by the caller.

`edge_betweenness`:

Pointer to an initialized vector or NULL. In the former case the edge betweenness of the removed edge is stored here. The vector will be resized as needed.

`merges`:

Pointer to an initialized matrix or NULL. If not NULL then merges performed by the algorithm are stored here. Even if this is a divisive algorithm, we can replay it backwards and note which two clusters were merged. Clusters are numbered from zero, see the `merges` argument of `igraph_community_walktrap()` for details. The matrix will be resized as needed.

`bridges`:

Pointer to an initialized vector of NULL. If not NULL then all edge removals which separated the network into more components are marked here.

`modularity`:

If not a null pointer, then the modularity values of the different divisions are stored here, in the order corresponding to the merge matrix. The modularity values will take weights into account if `weights` is not null.

`membership`:

If not a null pointer, then the membership vector, corresponding to the highest modularity value, is stored here.

directed:

Logical constant, whether to calculate directed betweenness (ie. directed paths) for directed graphs. It is ignored for undirected graphs.

weights:

An optional vector containing edge weights. If null, the unweighted edge betweenness scores will be calculated and used. If not null, the weighted edge betweenness scores will be calculated and used.

Returns:

Error code.

See also:

`igraph_community_eb_get_merges()`, `igraph_community_spinglass()`, `igraph_community_walktrap()`.

Time complexity: O(|V||E|^2), as the betweenness calculation requires O(|V||E|) and we do it |E|-1 times.

Example 22-5. File examples/simple/igraph_community_edge_betweenness.c

22.5.2. `igraph_community_eb_get_merges` — Calculating the merges, ie. the dendrogram for an edge betweenness community structure

```
int igraph_community_eb_get_merges(const igraph_t *graph,
       const igraph_vector_t *edges,
                                 const igraph_vector_t *weights,
       igraph_matrix_t *res,
       igraph_vector_t *bridges,
       igraph_vector_t *modularity,
       igraph_vector_t *membership);
```

This function is handy if you have a sequence of edge which are gradually removed from the network and you would like to know how the network falls apart into separate components. The edge sequence may come from the `igraph_community_edge_betweenness()` function, but this is not necessary. Note that `igraph_community_edge_betweenness` can also calculate the dendrogram, via its *`merges`* argument.

Arguments:

`graph`:

The input graph.

`edges`:

Vector containing the edges to be removed from the network, all edges are expected to appear exactly once in the vector.

`weights`:

An optional vector containing edge weights. If null, the unweighted modularity scores will be calculated. If not null, the weighted modularity scores will be calculated. Ignored if both *`modularity`* and *`membership`* are nulls.

`res`:

Pointer to an initialized matrix, if not NULL then the dendrogram will be stored here, in the same form as for the `igraph_community_walktrap()` function: the matrix has two columns and each line is a merge given by the ids of the merged components. The component ids are number from zero and component ids smaller than the number of vertices in the graph belong to individual vertices. The non-trivial components containing at least two vertices are numbered from `n`, `n` is the number of vertices in the graph. So if the first line contains `a` and `b` that means that components `a` and `b` are merged into component `n`, the second line creates component `n`+1, etc. The matrix will be resized as needed.

`bridges`:

Pointer to an initialized vector or NULL. If not null then the index of the edge removals which split the network will be stored here. The vector will be resized as needed.

`modularity`:

If not a null pointer, then the modularity values for the different divisions, corresponding to the merges matrix, will be stored here.

`membership`:

If not a null pointer, then the membership vector for the best division (in terms of modularity) will be stored here.

Returns:

Error code.

See also:

igraph_community_edge_betweenness().

Time complexity: O(|E|+|V|log|V|), |V| is the number of vertices, |E| is the number of edges.

22.6. Community structure based on the optimization of modularity

22.6.1. `igraph_community_fastgreedy` — Finding community structure by greedy optimization of modularity

```
int igraph_community_fastgreedy(const igraph_t *graph,
    const igraph_vector_t *weights,
    igraph_matrix_t *merges,
    igraph_vector_t *modularity,
    igraph_vector_t *membership);
```

This function implements the fast greedy modularity optimization algorithm for finding community structure, see A Clauset, MEJ Newman, C Moore: Finding community structure in very large networks, http://www.arxiv.org/abs/cond-mat/0408187 for the details.

Some improvements proposed in K Wakita, T Tsurumi: Finding community structure in mega-scale social networks, http://www.arxiv.org/abs/cs.CY/0702048v1 have also been implemented.

Arguments:

graph:

The input graph. It must be a graph without multiple edges. This is checked and an error message is given for graphs with multiple edges.

weights:

Potentially a numeric vector containing edge weights. Supply a null pointer here for unweighted graphs. The weights are expected to be non-negative.

merges:

Pointer to an initialized matrix or NULL, the result of the computation is stored here. The matrix has two columns and each merge corresponds to one merge, the ids of the two merged components are stored. The component ids are numbered from zero and the first `n` components are the individual vertices, `n` is the number of vertices in the graph. Component `n` is created in the first merge, component `n`+1 in the second merge, etc. The matrix will be resized as needed. If this argument is NULL then it is ignored completely.

`modularity`:

Pointer to an initialized vector or NULL pointer, in the former case the modularity scores along the stages of the computation are recorded here. The vector will be resized as needed.

`membership`:

Pointer to a vector. If not a null pointer, then the membership vector corresponding to the best split (in terms of modularity) is stored here.

Returns:

Error code.

See also:

`igraph_community_walktrap()`, `igraph_community_edge_betweenness()` for other community detection algorithms, `igraph_community_to_membership()` to convert the dendrogram to a membership vector.

Time complexity: O(|E||V|log|V|) in the worst case, O(|E|+|V|log^2|V|) typically, |V| is the number of vertices, |E| is the number of edges.

Example 22-6. File examples/simple/igraph_community_fastgreedy.c

22.6.2. `igraph_community_multilevel` — Finding community structure by multi-level optimization of modularity

```
int igraph_community_multilevel(const igraph_t *graph,
  const igraph_vector_t *weights, igraph_vector_t *membership,
  igraph_matrix_t *memberships, igraph_vector_t *modularity);
```

This function implements the multi-level modularity optimization algorithm for finding community structure, see VD Blondel, J-L Guillaume, R Lambiotte and E Lefebvre: Fast unfolding of community hierarchies in large networks, J Stat Mech P10008 (2008) for the details (preprint: http://arxiv.org/abs/arXiv:0803.0476). It is based on the modularity measure and a hierarchical approach. Initially, each vertex is assigned to a community on its own. In every step, vertices are re-assigned to communities in a local, greedy way: each vertex is moved to the community with which it achieves the highest contribution to modularity. When no vertices can be reassigned, each community is considered a vertex on its own, and the process starts again with the merged communities. The process stops when there is only a single vertex left or when the modularity cannot be increased any more in a step. This function was contributed by Tom Gregorovic.

Arguments:

`graph`:

The input graph. It must be an undirected graph.

`weights`:

Numeric vector containing edge weights. If `NULL`, every edge has equal weight. The weights are expected to be non-negative.

`membership`:

The membership vector, the result is returned here. For each vertex it gives the ID of its community. The vector must be initialized and it will be resized accordingly.

`memberships`:

Numeric matrix that will contain the membership vector after each level, if not `NULL`. It must be initialized and it will be resized accordingly.

`modularity`:

Numeric vector that will contain the modularity score after each level, if not `NULL`. It must be initialized and it will be resized accordingly.

Returns:

Error code.

Time complexity: in average near linear on sparse graphs.

Example 22-7. File examples/simple/igraph_community_multilevel.c

22.7. Label propagation

22.7.1. `igraph_community_label_propagation` — Community detection based on label propagation

```
int igraph_community_label_propagation(const igraph_t *graph,
                                       igraph_vector_t *membership,
                                       const igraph_vector_t *weights,
                                       const igraph_vector_t *initial,
                                       igraph_vector_bool_t *fixed,
           igraph_real_t *modularity);
```

This function implements the community detection method described in: Raghavan, U.N. and Albert, R. and Kumara, S.: Near linear time algorithm to detect community structures in large-scale networks. Phys Rev E 76, 036106. (2007). This version extends the original method by the ability to take edge weights into consideration and also by allowing some labels to be fixed.

Weights are taken into account as follows: when the new label of node i is determined, the algorithm iterates over all edges incident on node i and calculate the total weight of edges leading to other nodes with label 0, 1, 2, ..., k-1 (where k is the number of possible labels). The new label of node i will then be the label whose edges (among the ones incident on node i) have the highest total weight.

Arguments:

`graph`:

The input graph, should be undirected to make sense.

`membership`:

The membership vector, the result is returned here. For each vertex it gives the ID of its community (label).

`weights`:

The weight vector, it should contain a positive weight for all the edges.

`initial`:

The initial state. If NULL, every vertex will have a different label at the beginning. Otherwise it must be a vector with an entry for each vertex. Non-negative values denote different labels, negative entries denote vertices without labels.

fixed:

Boolean vector denoting which labels are fixed. Of course this makes sense only if you provided an initial state, otherwise this element will be ignored. Also note that vertices without labels cannot be fixed.

modularity:

If not a null pointer, then it must be a pointer to a real number. The modularity score of the detected community structure is stored here.

Returns:

Error code.

Time complexity: O(m+n)

Example 22-8. File examples/simple/igraph_community_label_propagation.c

22.8. The InfoMAP algorithm

22.8.1. `igraph_community_infomap` — Find community structure that minimizes the expected

```
int igraph_community_infomap(const igraph_t * graph,
                             const igraph_vector_t *e_weights,
                             const igraph_vector_t *v_weights,
                             int nb_trials,
                             igraph_vector_t *membership,
                             igraph_real_t *codelength);
```

description length of a random walker trajectory. Implementation of the InfoMap community detection algorithm.of Martin Rosvall and Carl T. Bergstrom. See : Visualization of the math and the map generator: www.mapequation.org [2] The original paper: M. Rosvall and C. T. Bergstrom, Maps of information flow reveal community structure in complex networks, PNAS 105, 1118 (2008) [http://dx.doi.org/10.1073/pnas.0706851105 , http://arxiv.org/abs/0707.0609] [3] A more detailed paper: M. Rosvall, D. Axelsson, and C. T. Bergstrom, The map equation, Eur. Phys. J. Special Topics 178, 13 (2009). [http://dx.doi.org/10.1140/epjst/e2010-01179-1 , http://arxiv.org/abs/0906.1405]

The original C++ implementation of Martin Rosvall is used, see http://www.tp.umu.se/~rosvall/downloads/infomap_undir.tgz. Intergation in igraph has be done by Emmanuel Navarro (who is grateful to * Martin Rosvall and Carl T. Bergstrom for providing this source code.)

Note that the graph must not contain isolated vertices.

If you want to specify a random seed (as in original implementation) you can use `igraph_rng_seed()`.

Arguments:

`graph`:

The input graph.

`e_weights`:

Numeric vector giving the weights of the edges. If it is a NULL pointer then all edges will have equal weights. The weights are expected to be positive.

`v_weights`:

Numeric vector giving the weights of the vertices. If it is a NULL pointer then all vertices will have equal weights. The weights are expected to be positive.

`nb_trials`:

The number of attempts to partition the network (can be any integer value equal or larger than 1).

`membership`:

Pointer to a vector. The membership vector is stored here.

`codelength`:

Pointer to a real. If not NULL the code length of the partition is stored here.

Returns:

Error code.

See also:

igraph_community_spinglass(), igraph_community_edge_betweenness(), igraph_community_walktrap().

Time complexity: TODO.

Chapter 23. Hierarchical random graphs

23.1. Introduction

A hierarchical random graph is an ensemble of undirected graphs with `n` vertices. It is defined via a binary tree with `n` leaf and `n-1` internal vertices, where the internal vertices are labeled with probabilities. The probability that two vertices are connected in the random graph is given by the probability label at their closest common ancestor.

Please read the following two articles for more about hierarchical random graphs: A. Clauset, C. Moore, and M.E.J. Newman. Hierarchical structure and the prediction of missing links in networks. Nature 453, 98 - 101 (2008); and A. Clauset, C. Moore, and M.E.J. Newman. Structural Inference of Hierarchies in Networks. In E. M. Airoldi et al. (Eds.): ICML 2006 Ws, Lecture Notes in Computer Science 4503, 1-13. Springer-Verlag, Berlin Heidelberg (2007).

igraph contains functions for fitting HRG models to a given network (`igraph_hrg_fit`), for generating networks from a given HRG ensemble (`igraph_hrg_game`, `igraph_hrg_sample`), converting an igraph graph to a HRG and back (`igraph_hrg_create`, `igraph_hrg_dendrogram`), for calculating a consensus tree from a set of sampled HRGs (`igraph_hrg_consensus`) and for predicting missing edges in a network based on its HRG models (`igraph_hrg_predict`).

The igraph HRG implementation is heavily based on the code published by Aaron Clauset, at his website, http://tuvalu.santafe.edu/~aaronc/hierarchy/

23.2. Representing HRGs

23.2.1. `igraph_hrg_t` — Data structure to store a hierarchical random graph

```
typedef struct igraph_hrg_t {
  igraph_vector_t left, right, prob, edges, vertices;
} igraph_hrg_t;
```

A hierarchical random graph (HRG) can be given as a binary tree, where the internal vertices are labeled with real numbers.

Note that you don't necessarily have to know this internal representation for using the HRG functions, just pass the HRG objects created by one igraph function, to another igraph function.

It has the following members:

Values:

`left`:

Vector that contains the left children of the internal tree vertices. The first vertex is always the root vertex, so the first element of the vector is the left child of the root vertex. Internal vertices are denoted with negative numbers, starting from -1 and going down, i.e. the root vertex is -1. Leaf vertices are denoted by non-negative number, starting from zero and up.

`right`:

Vector that contains the right children of the vertices, with the same encoding as the `left` vector.

`prob`:

The connection probabilities attached to the internal vertices, the first number belongs to the root vertex (i.e. internal vertex -1), the second to internal vertex -2, etc.

`edges`:

The number of edges in the subtree below the given internal vertex.

`vertices`:

The number of vertices in the subtree below the given internal vertex, including itself.

23.2.2. `igraph_hrg_init` — Allocate memory for a HRG.

```
int igraph_hrg_init(igraph_hrg_t *hrg, int n);
```

This function must be called before passing an `igraph_hrg_t` to an igraph function.

Arguments:

`hrg`:

Pointer to the HRG data structure to initialize.

`n`:

The number of vertices in the graph that is modeled by this HRG. It can be zero, if this is not yet known.

Returns:

Error code.

Time complexity: O(n), the number of vertices in the graph.

23.2.3. `igraph_hrg_destroy` — Deallocate memory for an HRG.

```
void igraph_hrg_destroy(igraph_hrg_t *hrg);
```

The HRG data structure can be reinitialized again with an `igraph_hrg_destroy` call.

Arguments:

`hrg`:

Pointer to the HRG data structure to deallocate.

Time complexity: operating system dependent.

23.2.4. `igraph_hrg_size` — Returns the size of the HRG, the number of leaf nodes.

```
int igraph_hrg_size(const igraph_hrg_t *hrg);
```

Arguments:

`hrg`:

Pointer to the HRG.

Returns:

The number of leaf nodes in the HRG.

Time complexity: O(1).

23.2.5. `igraph_hrg_resize` — Resize a HRG.

```
int igraph_hrg_resize(igraph_hrg_t *hrg, int newsize);
```

Arguments:

`hrg`:

Pointer to an initialized (see `igraph_hrg_init`) HRG.

`newsize`:

The new size, i.e. the number of leaf nodes.

Returns:

Error code.

Time complexity: O(n), n is the new size.

23.3. Fitting HRGs

23.3.1. `igraph_hrg_fit` — Fit a hierarchical random graph model to a network

```
int igraph_hrg_fit(const igraph_t *graph,
     igraph_hrg_t *hrg,
     igraph_bool_t start,
     int steps);
```

Arguments:

graph:

The igraph graph to fit the model to. Edge directions are ignored in directed graphs.

hrg:

Pointer to an initialized HRG, the result of the fitting is stored here. It can also be used to pass a HRG to the function, that can be used as the starting point of the Markov Chain Monte Carlo fitting, if the `start` argument is true.

start:

Logical, whether to start the fitting from the given HRG.

steps:

Integer, the number of MCMC steps to take in the fitting procedure. If this is zero, then the fitting stop is a convergence criteria is fulfilled.

Returns:

Error code.

Time complexity: TODO.

23.3.2. `igraph_hrg_consensus` — Calculate a consensus tree for a HRG.

```
int igraph_hrg_consensus(const igraph_t *graph,
    igraph_vector_t *parents,
    igraph_vector_t *weights,
    igraph_hrg_t *hrg,
    igraph_bool_t start,
    int num_samples);
```

The calculation can be started from the given HRG (`hrg`), or (if `start` is false), a HRG is first fitted to the given graph.

Arguments:

graph:

The input graph.

parents:

An initialized vector, the results are stored here. For each vertex, the id of its parent vertex is stored, or -1, if the vertex is the root vertex in the tree. The first n vertex ids (from 0) refer to the original vertices of the graph, the other ids refer to vertex groups.

weights:

Numeric vector, counts the number of times a given tree split occured in the generated network samples, for each internal vertices. The order is the same as in `parents`.

hrg:

A hierarchical random graph. It is used as a starting point for the sampling, if the `start` argument is true. It is modified along the MCMC.

start:

Logical, whether to use the supplied HRG (in `hrg`) as a starting point for the MCMC.

num_samples:

The number of samples to generate for creating the consensus tree.

Returns:

Error code.

Time complexity: TODO.

23.4. HRG sampling

23.4.1. `igraph_hrg_sample` — Sample from a hierarchical random graph model

```
int igraph_hrg_sample(const igraph_t *input_graph,
        igraph_t *sample,
        igraph_vector_ptr_t *samples,
        int no_samples,
        igraph_hrg_t *hrg,
        igraph_bool_t start);
```

Sample from a hierarchical random graph ensemble. The ensemble can be given as a graph (`input_graph`), or as a HRG object (`hrg`). If a graph is given, then first an MCMC optimization is performed to find the optimal fitting model; then the MCMC is used to sample the graph(s).

Arguments:

`input_graph`:

An igraph graph, or a null pointer. If not a null pointer, then a HRG is first fitted to the graph, possibly starting from the given HRG, if the `start` argument is true. If is is a null pointer, then the given HRG is used as a starting point, to find the optimum of the Markov chain, before the sampling.

`sample`:

Pointer to an uninitialized graph, or a null pointer. If only one sample is requested, and it is not a null pointer, then the sample is stored here.

`samples`:

An initialized vector of pointers. If more than one samples are requested, then they are stored here. Note that to free this data structure, you need to call `igraph_destroy` on each graph first, then `free`() on all pointers, and finally `igraph_vector_ptr_destroy`.

`no_samples`:

The number of samples to generate.

`hrg`:

A HRG. It is modified during the sampling.

`start`:

Logical, whether to start the MCMC from the given HRG.

Returns:

Error code.

Time complexity: TODO.

23.4.2. `igraph_hrg_game` — Generate a hierarchical random graph

```
int igraph_hrg_game(igraph_t *graph,
```

```
const igraph_hrg_t *hrg);
```

This function is a simple shortcut to `igraph_hrg_sample`. It creates a single graph, from the given HRG.

Arguments:

graph:

Pointer to an uninitialized graph, the new graph is created here.

hrg:

The hierarchical random graph model to sample from. It is modified during the MCMC process.

Returns:

Error code.

Time complexity: TODO.

23.5. Conversion to and from igraph graphs

23.5.1. `igraph_hrg_dendrogram` — Create a dendrogram from a hierarchical random graph.

```
int igraph_hrg_dendrogram(igraph_t *graph,
     const igraph_hrg_t *hrg);
```

Creates the igraph graph equivalent of an `igraph_hrg_t` data structure.

Arguments:

graph:

Pointer to an uninitialized graph, the result is stored here.

hrg:

The hierarchical random graph to convert.

Returns:

Error code.

Time complexity: O(n), the number of vertices in the graph.

23.5.2. `igraph_hrg_create` — Create a HRG from an igraph graph.

```
int igraph_hrg_create(igraph_hrg_t *hrg,
        const igraph_t *graph,
        const igraph_vector_t *prob);
```

Arguments:

hrg:

Pointer to an initialized `igraph_hrg_t`. The result is stored here.

graph:

The igraph graph to convert. It must be a directed binary tree, with n-1 internal and n leaf vertices. The root vertex must have in-degree zero.

prob:

The vector of probabilities, this is used to label the internal nodes of the hierarchical random graph. The values corresponding to the leaves are ignored.

Returns:

Error code.

Time complexity: O(n), the number of vertices in the tree.

23.6. Predicting missing edges

23.6.1. `igraph_hrg_predict` — Predict missing edges in a graph, based on HRG models

```
int igraph_hrg_predict(const igraph_t *graph,
         igraph_vector_t *edges,
         igraph_vector_t *prob,
         igraph_hrg_t *hrg,
         igraph_bool_t start,
         int num_samples,
         int num_bins);
```

Samples HRG models for a network, and estimated the probability that an edge was falsely observed as non-existent in the network.

Arguments:

`graph`:

The input graph.

`edges`:

The list of missing edges is stored here, the first two elements are the first edge, the next two the second edge, etc.

`prob`:

Vector of probabilies for the existence of missing edges, in the order corresponding to `edges`.

`hrg`:

A HRG, it is used as a starting point if `start` is true. It is also modified during the MCMC sampling.

`start`:

Logical, whether to start the MCMC from the given HRG.

`num_samples`:

The number of samples to generate.

`num_bins`:

Controls the resolution of the edge probabilities. Higher numbers result higher resolution.

Returns:

Error code.

Time complexity: TODO.

Chapter 24. Spectral Coarse Graining

24.1. Introduction

The SCG functions provide a framework, called Spectral Coarse Graining (SCG), for reducing large graphs while preserving their *spectral-related features*, that is features closely related with the eigenvalues and eigenvectors of a graph matrix (which for now can be the adjacency, the stochastic, or the Laplacian matrix).

Common examples of such features comprise the first-passage-time of random walkers on Markovian graphs, thermodynamic properties of lattice models in statistical physics (e.g. Ising model), and the epidemic threshold of epidemic network models (SIR and SIS models).

SCG differs from traditional clustering schemes by producing a *coarse-grained graph* (not just a partition of the vertices), representative of the original one. As shown in [1], Principal Component Analysis can be viewed as a particular SCG, called *exact SCG*, where the matrix to be coarse-grained is the covariance matrix of some data set.

SCG should be of interest to practitioners of various fields dealing with problems where matrix eigenpairs play an important role, as for instance is the case of dynamical processes on networks.

24.1.1. SCG in brief

The main idea of SCG is to operate on a matrix a shrinkage operation specifically designed to preserve some of the matrix eigenpairs while not altering other important matrix features (such as its structure). Mathematically, this idea was expressed as follows. Consider a (complex) n x n matrix M and form the product

M'=LMR*,

where n' < n and L, R are from C[n'xn]} and are such that LR*=I[n'] (R* denotes the conjugate transpose of R). Under these assumptions, it can be shown that P=R*L is an n'-rank projector and that, if (lambda, v) is a (right) eigenpair of M (i.e. Mv=lambda v} and P is orthogonal, there exists an eigenvalue lambda' of M' such that

|lambda-lambda'| <= const ||e[P](v)|| [1+O(||e[P](v)||^2)],

where ||e[P](v)||=||v-Pv||. Hence, if P (or equivalently L, R) is chosen so as to make ||e[P](v)|| as small as possible, one can preserve to any desired level the original eigenvalue lambda in the coarse-grained matrix M'; under extra assumptions on M, this result can be generalized to eigenvectors [1]. This leads to the following generic definition of a SCG problem.

Given M (C[nxn]) and (lambda, v), a (right) eigenpair of M to be preserved by the coarse graining, the problem is to find a projector P' solving min(||e[P](v)||, p in Omega),

where Omega is a set of projectors in C[nxn] described by some ad hoc constraints c[1], ..., c[r] (e.g. c[1]: P in R[nxn], c[2]: P=t(P), c[3]: P[i,j] >= 0}, etc).

Choosing pertinent constraints to solve the SCG problem is of great importance in applications. For instance, in the absence of constraints the SCG problem is solved trivially by P'=vv* (v is assumed normalized). We have designed a particular constraint, called *homogeneous mixing*, which ensures that vertices belonging to the same group are merged consistently from a physical point of view (see [1] for details). Under this constraint the SCG problem reduces to finding the partition of 1, ..., n (labeling the original vertices) minimizing

$||e[P](v)||^2$ = sum($[v(i)-(Pv)(i)]^2$; alpha=1,...,n', i in alpha),

where alpha denotes a group (i.e. a block) in a partition of {1, ..., n}, and |alpha| is the number of elements in alpha.

If M is symmetric or stochastic, for instance, then it may be desirable (or mandatory) to choose L, R so that M' is symmetric or stochastic as well. This *structural constraint* has led to the construction of particular semi-projectors for symmetric [1], stochastic [3] and Laplacian [2] matrices, that are made available.

In short, the coarse graining of matrices and graphs involves:

1. Retrieving a matrix or a graph matrix M from the problem.
2. Computing the eigenpairs of M to be preserved in the coarse-grained graph or matrix.
3. Setting some problem-specific constraints (e.g. dimension of the coarse-grained object).
4. Solving the constrained SCG problem, that is finding P'.
5. Computing from P' two semi-projectors L' and R' (e.g. following the method proposed in [1]).
6. Working out the product M'=L'MR'* and, if needed, defining from M' a coarse-grained graph.

24.1.2. Functions for performing SCG

The main functions are `igraph_scg_adjacency()`, `igraph_scg_laplacian()` and `igraph_scg_stochastic()`. These functions handle all the steps involved in the Spectral Coarse Graining (SCG) of some particular matrices and graphs as described above and in reference [1]. In more details, they compute some prescribed eigenpairs of a matrix or a graph matrix, (for now adjacency, Laplacian and stochastic matrices are available), work out an optimal partition to preserve the eigenpairs, and finally output a coarse-grained matrix or graph along with other useful information.

These steps can also be carried out independently: (1) Use `igraph_get_adjacency()`, `igraph_get_sparsemat()`, `igraph_laplacian()`, `igraph_get_stochastic()` or `igraph_get_stochastic_sparsemat()` to compute a matrix M. (2) Work out some prescribed eigenpairs of M e.g. by means of `igraph_arpack_rssolve()` or `igraph_arpack_rnsolve()`. (3) Invoke one the four algorithms of the function `igraph_scg_grouping()` to get a partition that will preserve the eigenpairs in the coarse-grained matrix. (4) Compute the semi-projectors L and R using `igraph_scg_semiprojectors()` and from there the coarse-grained matrix M'=LMR*. If necessary, construct a coarse-grained graph from M' (e.g. as in [1]).

24.1.3. References

[1] D. Morton de Lachapelle, D. Gfeller, and P. De Los Rios, Shrinking Matrices while Preserving their Eigenpairs with Application to the Spectral Coarse Graining of Graphs. Submitted to *SIAM Journal on Matrix Analysis and Applications*, 2008. http://people.epfl.ch/david.morton

[2] D. Gfeller, and P. De Los Rios, Spectral Coarse Graining and Synchronization in Oscillator Networks. *Physical Review Letters*, **100**(17), 2008. http://arxiv.org/abs/0708.2055

[3] D. Gfeller, and P. De Los Rios, Spectral Coarse Graining of Complex Networks, *Physical Review Letters*, **99**(3), 2007. http://arxiv.org/abs/0706.0812

24.2. SCG functions

24.2.1. `igraph_scg_adjacency` — Spectral coarse graining, symmetric case.

```
int igraph_scg_adjacency(const igraph_t *graph,
    const igraph_matrix_t *matrix,
    const igraph_sparsemat_t *sparsemat,
    const igraph_vector_t *ev,
    igraph_integer_t nt,
    const igraph_vector_t *nt_vec,
    igraph_scg_algorithm_t algo,
    igraph_vector_t *values,
    igraph_matrix_t *vectors,
    igraph_vector_t *groups,
    igraph_bool_t use_arpack,
    igraph_integer_t maxiter,
    igraph_t *scg_graph,
    igraph_matrix_t *scg_matrix,
    igraph_sparsemat_t *scg_sparsemat,
    igraph_matrix_t *L,
    igraph_matrix_t *R,
    igraph_sparsemat_t *Lsparse,
    igraph_sparsemat_t *Rsparse);
```

This function handles all the steps involved in the Spectral Coarse Graining (SCG) of some matrices and graphs as described in the reference below.

Arguments:

graph:

The input graph. Exactly one of *graph*, *matrix* and *sparsemat* must be given, the other two must be NULL pointers.

matrix:

The input matrix. Exactly one of *graph*, *matrix* and *sparsemat* must be given, the other two must be NULL pointers.

sparsemat:

The input sparse matrix. Exactly one of *graph*, *matrix* and *sparsemat* must be given, the other two must be NULL pointers.

ev:

A vector of positive integers giving the indexes of the eigenpairs to be preserved. 1 designates the eigenvalue with largest algebraic value, 2 the one with second largest algebraic value, etc.

nt:

Positive integer. When *algo* is IGRAPH_SCG_OPTIMUM, it gives the number of groups to partition each eigenvector separately. When *algo* is IGRAPH_SCG_INTERV or IGRAPH_SCG_INTERV_KM, it gives the number of intervals to partition each eigenvector. This is ignored when *algo* is IGRAPH_SCG_EXACT.

nt_vec:

A numeric vector of length one or the length must match the number of eigenvectors given in *V*, or a NULL pointer. If not NULL, then this argument gives the number of groups or intervals, and *nt* is ignored. Different number of groups or intervals can be specified for each eigenvector.

algo:

The algorithm to solve the SCG problem. Possible values: IGRAPH_SCG_OPTIMUM, IGRAPH_SCG_INTERV_KM, IGRAPH_SCG_INTERV and IGRAPH_SCG_EXACT. Please see the details about them above.

values:

If this is not NULL and the eigenvectors are re-calculated, then the eigenvalues are stored here.

vectors:

If this is not NULL, and not a zero-length matrix, then it is interpreted as the eigenvectors to use for the coarse-graining. Otherwise the eigenvectors are re-calculated, and they are stored here. (If this is not NULL.)

groups:

If this is not NULL, and not a zero-length vector, then it is interpreted as the vector of group labels. (Group labels are integers from zero and are sequential.) Otherwise group labels are re-calculated and stored here, if this argument is not a null pointer.

use_arpack:

Whether to use ARPACK for solving the eigenproblem. Currently ARPACK is not implemented.

maxiter:

A positive integer giving the number of iterations of the k-means algorithm when *algo* is IGRAPH_SCG_INTERV_KM. It is ignored in other cases. A reasonable (initial) value for this argument is 100.

scg_graph:

If not a NULL pointer, then the coarse-grained graph is returned here.

scg_matrix:

If not a NULL pointer, then it must be an initialied matrix, and the coarse-grained matrix is returned here.

scg_sparsemat:

If not a NULL pointer, then the coarse grained matrix is returned here, in sparse matrix form.

L:

If not a NULL pointer, then it must be an initialized matrix and the left semi-projector is returned here.

R:

If not a NULL pointer, then it must be an initialized matrix and the right semi-projector is returned here.

Lsparse:

If not a NULL pointer, then the left semi-projector is returned here.

Rsparse:

If not a NULL pointer, then the right semi-projector is returned here.

Returns:

Error code.

Time complexity: TODO.

See also:

`igraph_scg_grouping()`, `igraph_scg_semiprojectors()`, `igraph_scg_stochastic()` and `igraph_scg_laplacian()`.

Example 24-1. File examples/simple/scg.c

24.2.2. `igraph_scg_stochastic` — Spectral coarse graining, stochastic case.

```
int igraph_scg_stochastic(const igraph_t *graph,
     const igraph_matrix_t *matrix,
     const igraph_sparsemat_t *sparsemat,
     const igraph_vector_t *ev,
     igraph_integer_t nt,
     const igraph_vector_t *nt_vec,
     igraph_scg_algorithm_t algo,
     igraph_scg_norm_t norm,
     igraph_vector_complex_t *values,
     igraph_matrix_complex_t *vectors,
     igraph_vector_t *groups,
     igraph_vector_t *p,
     igraph_bool_t use_arpack,
     igraph_integer_t maxiter,
     igraph_t *scg_graph,
     igraph_matrix_t *scg_matrix,
     igraph_sparsemat_t *scg_sparsemat,
     igraph_matrix_t *L,
     igraph_matrix_t *R,
     igraph_sparsemat_t *Lsparse,
     igraph_sparsemat_t *Rsparse);
```

This function handles all the steps involved in the Spectral Coarse Graining (SCG) of some matrices and graphs as described in the reference below.

Arguments:

graph:

The input graph. Exactly one of *graph*, *matrix* and *sparsemat* must be given, the other two must be `NULL` pointers.

matrix:

The input matrix. Exactly one of *graph*, *matrix* and *sparsemat* must be given, the other two must be NULL pointers.

sparsemat:

The input sparse matrix. Exactly one of *graph*, *matrix* and *sparsemat* must be given, the other two must be NULL pointers.

ev:

A vector of positive integers giving the indexes of the eigenpairs to be preserved. 1 designates the eigenvalue with largest magnitude, 2 the one with second largest magnitude, etc.

nt:

Positive integer. When *algo* is IGRAPH_SCG_OPTIMUM, it gives the number of groups to partition each eigenvector separately. When *algo* is IGRAPH_SCG_INTERV or IGRAPH_SCG_INTERV_KM, it gives the number of intervals to partition each eigenvector. This is ignored when *algo* is IGRAPH_SCG_EXACT.

nt_vec:

A numeric vector of length one or the length must match the number of eigenvectors given in *V*, or a NULL pointer. If not NULL, then this argument gives the number of groups or intervals, and *nt* is ignored. Different number of groups or intervals can be specified for each eigenvector.

algo:

The algorithm to solve the SCG problem. Possible values: IGRAPH_SCG_OPTIMUM, IGRAPH_SCG_INTERV_KM, IGRAPH_SCG_INTERV and IGRAPH_SCG_EXACT. Please see the details about them above.

norm:

Either IGRAPH_SCG_NORM_ROW or IGRAPH_SCG_NORM_COL. Specifies whether the rows or the columns of the stochastic matrix sum up to one.

values:

If this is not NULL and the eigenvectors are re-calculated, then the eigenvalues are stored here.

vectors:

If this is not NULL, and not a zero-length matrix, then it is interpreted as the eigenvectors to use for the coarse-graining. Otherwise the eigenvectors are re-calculated, and they are stored here. (If this is not NULL.)

groups:

If this is not NULL, and not a zero-length vector, then it is interpreted as the vector of group labels. (Group labels are integers from zero and are sequential.) Otherwise group labels are re-calculated and stored here, if this argument is not a null pointer.

p:

If this is not `NULL`, and not zero length, then it is interpreted as the stationary probability distribution of the Markov chain corresponding to the input matrix/graph. Its length must match the number of vertices in the input graph (or number of rows in the input matrix). If not given, then the stationary distribution is calculated and stored here. (Unless this argument is a `NULL` pointer, in which case it is not stored.)

use_arpack:

Whether to use ARPACK for solving the eigenproblem. Currently ARPACK is not implemented.

maxiter:

A positive integer giving the number of iterations of the k-means algorithm when *algo* is `IGRAPH_SCG_INTERV_KM`. It is ignored in other cases. A reasonable (initial) value for this argument is 100.

scg_graph:

If not a `NULL` pointer, then the coarse-grained graph is returned here.

scg_matrix:

If not a `NULL` pointer, then it must be an initialied matrix, and the coarse-grained matrix is returned here.

scg_sparsemat:

If not a `NULL` pointer, then the coarse grained matrix is returned here, in sparse matrix form.

L:

If not a `NULL` pointer, then it must be an initialized matrix and the left semi-projector is returned here.

R:

If not a `NULL` pointer, then it must be an initialized matrix and the right semi-projector is returned here.

Lsparse:

If not a `NULL` pointer, then the left semi-projector is returned here.

Rsparse:

If not a `NULL` pointer, then the right semi-projector is returned here.

Returns:

Error code.

Time complexity: TODO.

See also:

`igraph_scg_grouping()`, `igraph_scg_semiprojectors()`, `igraph_scg_adjacency()` and `igraph_scg_laplacian()`.

Example 24-2. File examples/simple/scg2.c

24.2.3. `igraph_scg_laplacian` — Spectral coarse graining, laplacian matrix.

```
int igraph_scg_laplacian(const igraph_t *graph,
    const igraph_matrix_t *matrix,
    const igraph_sparsemat_t *sparsemat,
    const igraph_vector_t *ev,
    igraph_integer_t nt,
    const igraph_vector_t *nt_vec,
    igraph_scg_algorithm_t algo,
    igraph_scg_norm_t norm,
    igraph_scg_direction_t direction,
    igraph_vector_complex_t *values,
    igraph_matrix_complex_t *vectors,
    igraph_vector_t *groups,
    igraph_bool_t use_arpack,
    igraph_integer_t maxiter,
    igraph_t *scg_graph,
    igraph_matrix_t *scg_matrix,
    igraph_sparsemat_t *scg_sparsemat,
    igraph_matrix_t *L,
    igraph_matrix_t *R,
    igraph_sparsemat_t *Lsparse,
    igraph_sparsemat_t *Rsparse);
```

This function handles all the steps involved in the Spectral Coarse Graining (SCG) of some matrices and graphs as described in the reference below.

Arguments:

graph:

The input graph. Exactly one of *graph*, *matrix* and *sparsemat* must be given, the other two must be `NULL` pointers.

matrix:

The input matrix. Exactly one of *graph*, *matrix* and *sparsemat* must be given, the other two must be `NULL` pointers.

sparsemat:

The input sparse matrix. Exactly one of *graph*, *matrix* and *sparsemat* must be given, the other two must be `NULL` pointers.

ev:

A vector of positive integers giving the indexes of the eigenpairs to be preserved. 1 designates the eigenvalue with largest magnitude, 2 the one with second largest magnitude, etc.

nt:

Positive integer. When *algo* is `IGRAPH_SCG_OPTIMUM`, it gives the number of groups to partition each eigenvector separately. When *algo* is `IGRAPH_SCG_INTERV` or `IGRAPH_SCG_INTERV_KM`, it gives the number of intervals to partition each eigenvector. This is ignored when *algo* is `IGRAPH_SCG_EXACT`.

nt_vec:

A numeric vector of length one or the length must match the number of eigenvectors given in *V*, or a `NULL` pointer. If not `NULL`, then this argument gives the number of groups or intervals, and *nt* is ignored. Different number of groups or intervals can be specified for each eigenvector.

algo:

The algorithm to solve the SCG problem. Possible values: `IGRAPH_SCG_OPTIMUM`, `IGRAPH_SCG_INTERV_KM`, `IGRAPH_SCG_INTERV` and `IGRAPH_SCG_EXACT`. Please see the details about them above.

norm:

Either `IGRAPH_SCG_NORM_ROW` or `IGRAPH_SCG_NORM_COL`. Specifies whether the rows or the columns of the Laplacian matrix sum up to zero.

direction:

Whether to work with left or right eigenvectors. Possible values: `IGRAPH_SCG_DIRECTION_DEFAULT`, `IGRAPH_SCG_DIRECTION_LEFT`, `IGRAPH_SCG_DIRECTION_RIGHT`. This argument is currently ignored and right eigenvectors are always used.

values:

If this is not `NULL` and the eigenvectors are re-calculated, then the eigenvalues are stored here.

vectors:

If this is not `NULL`, and not a zero-length matrix, then it is interpreted as the eigenvectors to use for the coarse-graining. Otherwise the eigenvectors are re-calculated, and they are stored here. (If this is not `NULL`.)

groups:

If this is not `NULL`, and not a zero-length vector, then it is interpreted as the vector of group labels. (Group labels are integers from zero and are sequential.) Otherwise group labels are re-calculated and stored here, if this argument is not a null pointer.

use_arpack:

Whether to use ARPACK for solving the eigenproblem. Currently ARPACK is not implemented.

maxiter:

A positive integer giving the number of iterations of the k-means algorithm when *algo* is `IGRAPH_SCG_INTERV_KM`. It is ignored in other cases. A reasonable (initial) value for this argument is 100.

scg_graph:

If not a `NULL` pointer, then the coarse-grained graph is returned here.

scg_matrix:

If not a `NULL` pointer, then it must be an initialied matrix, and the coarse-grained matrix is returned here.

scg_sparsemat:

If not a `NULL` pointer, then the coarse grained matrix is returned here, in sparse matrix form.

L:

If not a `NULL` pointer, then it must be an initialized matrix and the left semi-projector is returned here.

R:

If not a `NULL` pointer, then it must be an initialized matrix and the right semi-projector is returned here.

Lsparse:

If not a `NULL` pointer, then the left semi-projector is returned here.

Rsparse:

If not a `NULL` pointer, then the right semi-projector is returned here.

Returns:

Error code.

Time complexity: TODO.

See also:

`igraph_scg_grouping()`, `igraph_scg_semiprojectors()`, `igraph_scg_stochastic()` and `igraph_scg_adjacency()`.

Example 24-3. File examples/simple/scg3.c

24.2.4. `igraph_scg_grouping` — SCG problem solver

```
int igraph_scg_grouping(const igraph_matrix_t *V,
   igraph_vector_t *groups,
   igraph_integer_t nt,
   const igraph_vector_t *nt_vec,
   igraph_scg_matrix_t mtype,
   igraph_scg_algorithm_t algo,
   const igraph_vector_t *p,
   igraph_integer_t maxiter);
```

This function solves the Spectral Coarse Graining (SCG) problem; either exactly, or approximately but faster.

The algorithm `IGRAPH_SCG_OPTIMUM` solves exactly the SCG problem for each eigenvector in *`V`*. The running time of this algorithm is O(max(nt) m^2) for the symmetric and laplacian matrix problems It is O(m^3) for the stochastic problem. Here m is the number of rows in *`V`*. In all three cases, the memory usage is O(m^2).

The algorithms `IGRAPH_SCG_INTERV` and `IGRAPH_SCG_INTERV_KM` solve approximately the SCG problem by performing a (for now) constant binning of the components of the eigenvectors, that is *`nt`* `VECTOR(nt_vec)[i]`) constant-size bins are used to partition `V[,i]` . When *`algo`* is `IGRAPH_SCG_INTERV_KM`, the (Lloyd) k-means algorithm is run on each partition obtained by `IGRAPH_SCG_INTERV` to improve accuracy.

Once a minimizing partition (either exact or approximate) has been found for each eigenvector, the final grouping is worked out as follows: two vertices are grouped together in the final partition if they are grouped together in each minimizing partition. In general the size of the final partition is not known in advance when the number of columns in `V` is larger than one.

Finally, the algorithm `IGRAPH_SCG_EXACT` groups the vertices with equal components in each eigenvector. The last three algorithms essentially have linear running time and memory load.

Arguments:

V:

The matrix of eigenvectors to be preserved by coarse graining, each column is an eigenvector.

groups:

Pointer to an initialized vector, the result of the SCG is stored here.

nt:

Positive integer. When *algo* is `IGRAPH_SCG_OPTIMUM`, it gives the number of groups to partition each eigenvector separately. When *algo* is `IGRAPH_SCG_INTERV` or `IGRAPH_SCG_INTERV_KM`, it gives the number of intervals to partition each eigenvector. This is ignored when *algo* is `IGRAPH_SCG_EXACT`.

nt_vec:

A numeric vector of length one or the length must match the number of eigenvectors given in *V*, or a `NULL` pointer. If not `NULL`, then this argument gives the number of groups or intervals, and *nt* is ignored. Different number of groups or intervals can be specified for each eigenvector.

mtype:

The type of semi-projectors used in the SCG. Possible values are `IGRAPH_SCG_SYMMETRIC`, `IGRAPH_SCG_STOCHASTIC` and `IGRAPH_SCG_LAPLACIAN`.

algo:

The algorithm to solve the SCG problem. Possible values: `IGRAPH_SCG_OPTIMUM`, `IGRAPH_SCG_INTERV_KM`, `IGRAPH_SCG_INTERV` and `IGRAPH_SCG_EXACT`. Please see the details about them above.

p:

A probability vector, or `NULL`. This argument must be given if *mtype* is `IGRAPH_SCG_STOCHASTIC`, but it is ignored otherwise. For the stochastic case it gives the stationary probability distribution of a Markov chain, the one specified by the graph/matrix under study.

maxiter:

A positive integer giving the number of iterations of the k-means algorithm when *algo* is `IGRAPH_SCG_INTERV_KM`. It is ignored in other cases. A reasonable (initial) value for this argument is 100.

Returns:

Error code.

Time complexity: see description above.

See also:

`igraph_scg_adjacency()`, `igraph_scg_laplacian()`, `igraph_scg_stochastic()`.

Example 24-4. File examples/simple/igraph_scg_grouping.c

Example 24-5. File examples/simple/igraph_scg_grouping2.c

Example 24-6. File examples/simple/igraph_scg_grouping3.c

Example 24-7. File examples/simple/igraph_scg_grouping4.c

24.2.5. `igraph_scg_semiprojectors` — Compute SCG semi-projectors for a given partition

```
int igraph_scg_semiprojectors(const igraph_vector_t *groups,
         igraph_scg_matrix_t mtype,
         igraph_matrix_t *L,
         igraph_matrix_t *R,
         igraph_sparsemat_t *Lsparse,
```

```
        igraph_sparsemat_t *Rsparse,
        const igraph_vector_t *p,
        igraph_scg_norm_t norm);
```

The three types of semi-projectors are defined as follows. Let gamma(j) label the group of vertex j in a partition of all the vertices.

The symmetric semi-projectors are defined as

L[alpha,j] = R[alpha,j] = 1/sqrt(|alpha|) delta[alpha,gamma(j)],

the (row) Laplacian semi-projectors as

L[alpha,j] = 1/|alpha| delta[alpha,gamma(j)]

and

R[alpha,j] = delta[alpha,gamma(j)],

and the (row) stochastic semi-projectors as

L[alpha,j] = p[1][j] / sum(p[1][k]; k in gamma(j)) delta[alpha,gamma(j)]

and

R[alpha,j] = delta[alpha,gamma(j)],

where p[1] is the (left) eigenvector associated with the one-eigenvalue of the stochastic matrix. L and R are defined in a symmetric way when `norm` is `IGRAPH_SCG_NORM_COL`. All these semi-projectors verify various properties described in the reference.

Arguments:

groups:

A vector of integers, giving the group label of every vertex in the partition. Group labels should start at zero and should be sequential.

mtype:

The type of semi-projectors. For now `IGRAPH_SCG_SYMMETRIC`, `IGRAPH_SCG_STOCHASTIC` and `IGRAP_SCG_LAPLACIAN` are supported.

L:

If not a `NULL` pointer, then it must be a pointer to an initialized matrix. The left semi-projector is stored here.

R:

If not a `NULL` pointer, then it must be a pointer to an initialized matrix. The right semi-projector is stored here.

Lsparse:

If not a `NULL` pointer, then it must be a pointer to an uninitialized sparse matrix. The left semi-projector is stored here.

Rsparse:

If not a `NULL` pointer, then it must be a pointer to an uninitialized sparse matrix. The right semi-projector is stored here.

p:

`NULL`, or a probability vector of the same length as *groups*. *p* is the stationary probability distribution of a Markov chain when *mtype* is `IGRAPH_SCG_STOCHASTIC`. This argument is ignored in all other cases.

norm:

Either `IGRAPH_SCG_NORM_ROW` or `IGRAPH_SCG_NORM_COL`. Specifies whether the rows or the columns of the Laplacian matrix sum up to zero, or whether the rows or the columns of the stochastic matrix sum up to one.

Returns:

Error code.

Time complexity: TODO.

See also:

`igraph_scg_adjacency()`, `igraph_scg_stochastic()` and `igraph_scg_laplacian()`, `igraph_scg_grouping()`.

Example 24-8. File examples/simple/igraph_scg_semiprojectors.c

Example 24-9. File examples/simple/igraph_scg_semiprojectors2.c

Example 24-10. File examples/simple/igraph_scg_semiprojectors3.c

24.2.6. `igraph_scg_norm_eps` — Calculate SCG residuals

```
int igraph_scg_norm_eps(const igraph_matrix_t *V,
   const igraph_vector_t *groups,
   igraph_vector_t *eps,
   igraph_scg_matrix_t mtype,
   const igraph_vector_t *p,
   igraph_scg_norm_t norm);
```

Computes |v[i]-Pv[i]|, where v[i] is the i-th eigenvector in *V* and P is the projector corresponding to the *mtype* argument.

Arguments:

V:

The matrix of eigenvectors to be preserved by coarse graining, each column is an eigenvector.

groups:

A vector of integers, giving the group label of every vertex in the partition. Group labels should start at zero and should be sequential.

eps:

Pointer to a real value, the result is stored here.

mtype:

The type of semi-projectors. For now `IGRAPH_SCG_SYMMETRIC`, `IGRAPH_SCG_STOCHASTIC` and `IGRAP_SCG_LAPLACIAN` are supported.

p:

`NULL`, or a probability vector of the same length as *groups*. *p* is the stationary probability distribution of a Markov chain when *mtype* is `IGRAPH_SCG_STOCHASTIC`. This argument is ignored in all other cases.

norm:

Either `IGRAPH_SCG_NORM_ROW` or `IGRAPH_SCG_NORM_COL`. Specifies whether the rows or the columns of the Laplacian matrix sum up to zero, or whether the rows or the columns of the stochastic matrix sum up to one.

Returns:

Error code.

Time complexity: TODO.

See also:

`igraph_scg_adjacency()`, `igraph_scg_stochastic()` and `igraph_scg_laplacian()`, `igraph_scg_grouping()`, `igraph_scg_semiprojectors()`.

Chapter 25. Graph Operators

25.1. Union and intersection

25.1.1. `igraph_disjoint_union` — Creates the union of two disjoint graphs

```
int igraph_disjoint_union(igraph_t *res, const igraph_t *left,
     const igraph_t *right);
```

First the vertices of the second graph will be relabeled with new vertex ids to have two disjoint sets of vertex ids, then the union of the two graphs will be formed. If the two graphs have |V1| and |V2| vertices and |E1| and |E2| edges respectively then the new graph will have |V1|+|V2| vertices and |E1|+|E2| edges.

Both graphs need to have the same directedness, ie. either both directed or both undirected.

The current version of this function cannot handle graph, vertex and edge attributes, they will be lost.

Arguments:

`res`:

Pointer to an uninitialized graph object, the result will stored here.

`left`:

The first graph.

`right`:

The second graph.

Returns:

Error code.

See also:

`igraph_disjoint_union_many()` for creating the disjoint union of more than two graphs, `igraph_union()` for non-disjoint union.

Time complexity: O(|V1|+|V2|+|E1|+|E2|).

Example 25-1. File examples/simple/igraph_disjoint_union.c

25.1.2. `igraph_disjoint_union_many` — The disjint union of many graphs.

```
int igraph_disjoint_union_many(igraph_t *res,
          const igraph_vector_ptr_t *graphs);
```

First the vertices in the graphs will be relabeled with new vertex ids to have pairwise disjoint vertex id sets and then the union of the graphs is formed. The number of vertices and edges in the result is the total number of vertices and edges in the graphs.

Both graphs need to have the same directedness, ie. either both directed or both undirected.

The current version of this function cannot handle graph, vertex and edge attributes, they will be lost.

Arguments:

`res`:

Pointer to an uninitialized graph object, the result of the operation will be stored here.

`graphs`:

Pointer vector, contains pointers to initialized graph objects.

Returns:

Error code.

See also:

`igraph_disjoint_union()` for an easier syntax if you have only two graphs, `igraph_union_many()` for non-disjoint union.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the result.

25.1.3. `igraph_union` — Calculates the union of two graphs.

```
int igraph_union(igraph_t *res,
   const igraph_t *left, const igraph_t *right);
```

The number of vertices in the result is that of the larger graph from the two arguments. The result graph contains edges which are present in at least one of the operand graphs.

Arguments:

`res`:

Pointer to an uninitialized graph object, the result will be stored here.

`left`:

The first graph.

`right`:

The second graph.

Returns:

Error code.

See also:

`igraph_union_many()` for the union of many graphs, `igraph_intersection()` and `igraph_difference()` for other operators.

Time complexity: O(|V|+|E|), |V| is the number of vertices, |E| the number of edges in the result graph.

Example 25-2. File examples/simple/igraph_union.c

25.1.4. `igraph_union_many` — Creates the union of many graphs.

```
int igraph_union_many(igraph_t *res, const igraph_vector_ptr_t *graphs);
```

The result graph will contain as many vertices as the largest graph among the arguments does, and an edge will be included in it if it is part of at least one operand graph.

The directedness of the operand graphs must be the same.

Arguments:

`res`:

Pointer to an uninitialized graph object, this will contain the result.

`graphs`:

Pointer vector, contains pointers to the operands of the union operator, graph objects of course.

Returns:

Error code.

See also:

`igraph_union()` for the union of two graphs, `igraph_intersection_many()`, `igraph_intersection()` and `igraph_difference` for other operators.

Time complexity: O(|V|+|E|), |V| is the number of vertices in largest graph and |E| is the number of edges in the result graph.

Example 25-3. File examples/simple/igraph_union.c

25.1.5. `igraph_intersection` — Collect the common edges from two graphs.

```
int igraph_intersection(igraph_t *res,
   const igraph_t *left, const igraph_t *right);
```

The result graph contains only edges present both in the first and the second graph. The number of vertices in the result graph is the same as the larger from the two arguments.

Arguments:

res:

Pointer to an uninitialized graph object. This will contain the result of the operation.

left:

The first operand, a graph object.

right:

The second operand, a graph object.

Returns:

Error code.

See also:

`igraph_intersection_many()` to calculate the intersection of many graphs at once, `igraph_union()`, `igraph_difference()` for other operators.

Time complexity: O(|V|+|E|), |V| is the number of nodes, |E| is the number of edges in the smaller graph of the two. (The one containing less vertices is considered smaller.)

Example 25-4. File examples/simple/igraph_intersection.c

25.1.6. `igraph_intersection_many` — The intersection of more than two graphs.

```
int igraph_intersection_many(igraph_t *res,
        const igraph_vector_ptr_t *graphs);
```

This function calculates the intersection of the graphs stored in the `graphs` argument. Only those edges will be included in the result graph which are part of every graph in `graphs`.

The number of vertices in the result graph will be the maximum number of vertices in the argument graphs.

Arguments:

`res`:

Pointer to an uninitialized graph object, the result of the operation will be stored here.

`graphs`:

Pointer vector, contains pointers to graphs objects, the operands of the intersection operator.

Returns:

Error code.

See also:

`igraph_intersection()` for the intersection of two graphs, `igraph_union_many()`, `igraph_union()` and `igraph_difference()` for other operators.

Time complexity: O(|V|+|E|), |V| is the number of vertices, |E| is the number of edges in the smallest graph (ie. the graph having the less vertices).

25.2. Other set-like operators

25.2.1. `igraph_difference` — Calculate the difference of two graphs

```
int igraph_difference(igraph_t *res,
        const igraph_t *orig, const igraph_t *sub);
```

The number of vertices in the result is the number of vertices in the original graph, ie. the left, first operand. In the results graph only edges will be included from `orig` which are not present in `sub`.

Arguments:

`res`:

Pointer to an uninitialized graph object, the result will be stored here.

`orig`:

The left operand of the operator, a graph object.

`sub`:

The right operand of the operator, a graph object.

Returns:

Error code.

See also:

`igraph_intersection()` and `igraph_union()` for other operators.

Time complexity: O(|V|+|E|), |V| is the number vertices in the smaller graph, |E| is the number of edges in the result graph.

Example 25-5. File examples/simple/igraph_difference.c

25.2.2. `igraph_complementer` — Create the complementer of a graph

```
int igraph_complementer(igraph_t *res, const igraph_t *graph,
   igraph_bool_t loops);
```

The complementer graph means that all edges which are not part of the original graph will be included in the result.

Arguments:

`res`:

Pointer to an uninitialized graph object.

`graph`:

The original graph.

`loops`:

Whether to add loop edges to the complementer graph.

Returns:

Error code.

See also:

`igraph_union()`, `igraph_intersection()` and `igraph_difference()`.

Time complexity: O(|V|+|E1|+|E2|), |V| is the number of vertices in the graph, |E1| is the number of edges in the original and |E2| in the complementer graph.

Example 25-6. File examples/simple/igraph_complementer.c

25.2.3. `igraph_compose` — Calculates the composition of two graphs

```
int igraph_compose(igraph_t *res, const igraph_t *g1, const igraph_t *g2);
```

The composition of graphs contains the same number of vertices as the bigger graph of the two operands. It contains an (i,j) edge if and only if there is a k vertex, such that the first graphs contains an (i,k) edge and the second graph a (k,j) edge.

This is of course exactly the composition of two binary relations.

Two two graphs must have the same directedness, otherwise the function returns with an error message. Note that for undirected graphs the two relations are by definition symmetric.

Arguments:

`res`:

Pointer to an uninitialized graph object, the result will be stored here.

`g1`:

The firs operand, a graph object.

g2:

The second operand, another graph object.

Returns:

Error code.

Time complexity: O(|V|*d1*d2), |V| is the number of vertices in the first graph, d1 and d2 the average degree in the first and second graphs.

Example 25-7. File examples/simple/igraph_compose.c

Chapter 26. Using BLAS, LAPACK and ARPACK for igraph matrices and graphs

26.1. BLAS interface in igraph

BLAS is a highly optimized library for basic linear algebra operations such as vector-vector, matrix-vector and matrix-matrix product. Please see http://www.netlib.org/blas/ for details and a reference implementation in Fortran. igraph contains some wrapper functions that can be used to call BLAS routines in a somewhat more user-friendly way. Not all BLAS routines are included in igraph, and even those which are included might not have wrappers; the extension of the set of wrapped functions will probably be driven by igraph's internal requirements. The wrapper functions usually substitute double-precision floating point arrays used by BLAS with igraph_vector_t and igraph_matrix_t instances and also remove those parameters (such as the number of rows/columns) that can be inferred from the passed arguments directly.

26.1.1. `igraph_blas_dgemv` — Matrix-vector multiplication using BLAS, vector version.

```
void igraph_blas_dgemv(igraph_bool_t transpose, igraph_real_t alpha,
        const igraph_matrix_t* a, const igraph_vector_t* x,
        igraph_real_t beta, igraph_vector_t* y);
```

This function is a somewhat more user-friendly interface to the `dgemv` function in BLAS. `dgemv` performs the operation y = alpha*A*x + beta*y, where x and y are vectors and A is an appropriately sized matrix (symmetric or unsymmetric).

Arguments:

`transpose`:

whether to transpose the matrix *`A`*

`alpha`:

the constant *`alpha`*

`a`:

the matrix *`A`*

x:

the vector *x*

beta:

the constant *beta*

y:

the vector *y* (which will be modified in-place)

Time complexity: O(nk) if the matrix is of size n x k

See also:

`igraph_blas_dgemv_array` if you have arrays instead of vectors.

Example 26-1. File examples/simple/blas.c

26.1.2. `igraph_blas_dgemv_array` — Matrix-vector multiplication using BLAS, array version.

```
void igraph_blas_dgemv_array(igraph_bool_t transpose, igraph_real_t alpha,
        const igraph_matrix_t* a, const igraph_real_t* x,
        igraph_real_t beta, igraph_real_t* y);
```

This function is a somewhat more user-friendly interface to the `dgemv` function in BLAS. `dgemv` performs the operation y = alpha*A*x + beta*y, where x and y are vectors and A is an appropriately sized matrix (symmetric or unsymmetric).

Arguments:

transpose:

whether to transpose the matrix *A*

alpha:

the constant *alpha*

a:

the matrix *A*

x:

the vector *x* as a regular C array

beta:

the constant *beta*

y:

the vector *y* as a regular C array (which will be modified in-place)

Time complexity: O(nk) if the matrix is of size n x k

See also:

`igraph_blas_dgemv` if you have vectors instead of arrays.

26.2. LAPACK interface in igraph

LAPACK is written in Fortran90 and provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as reordering of the Schur factorizations and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision.

igraph provides an interface to a very limited set of LAPACK functions, using the regular igraph data structures.

See more about LAPACK at http://www.netlib.org/lapack/

26.2.1. Matrix factorization, solving linear systems

26.2.1.1. `igraph_lapack_dgetrf` — LU factorization of a general M-by-N matrix

```
int igraph_lapack_dgetrf(igraph_matrix_t *a, igraph_vector_int_t *ipiv,
    int *info);
```

The factorization has the form A = P * L * U where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

Arguments:

a:

The input/output matrix. On entry, the M-by-N matrix to be factored. On exit, the factors L and U from the factorization A = P * L * U; the unit diagonal elements of L are not stored.

ipiv:

An integer vector, the pivot indices are stored here, unless it is a null pointer. Row i of the matrix was interchanged with row ipiv[i].

info:

LAPACK error code. Zero on successful exit. If positive and i, then U(i,i) is exactly zero. The factorization has been completed, but the factor U is exactly singular, and division by zero will occur if it is used to solve a system of equations. If LAPACK returns an error, i.e. a negative info value, then an igraph error is generated as well.

Returns:

Error code.

Time complexity: TODO.

26.2.1.2. `igraph_lapack_dgetrs` — Solve general system of linear equations using LU factorization

```
int igraph_lapack_dgetrs(igraph_bool_t transpose, const igraph_matrix_t *a,
```

```
    igraph_vector_int_t *ipiv, igraph_matrix_t *b);
```

This function calls LAPACK to solve a system of linear equations A * X = B or A' * X = B with a general N-by-N matrix A using the LU factorization computed by `igraph_lapack_dgetrf`.

Arguments:

transpose:

Logical scalar, whether to transpose the input matrix.

a:

A matrix containing the L and U factors from the factorization A = P*L*U.

ipiv:

An integer vector, the pivot indices from `igraph_lapack_dgetrf` must be given here.

b:

The right hand side matrix must be given here.

Returns:

Error code.

Time complexity: TODO.

26.2.1.3. `igraph_lapack_dgesv` — Solve system of linear equations with LU factorization

```
int igraph_lapack_dgesv(igraph_matrix_t *a, igraph_vector_int_t *ipiv,
   igraph_matrix_t *b, int *info);
```

This function computes the solution to a real system of linear equations A * X = B, where A is an N-by-N matrix and X and B are N-by-NRHS matrices.

The LU decomposition with partial pivoting and row interchanges is used to factor A as A = P * L * U, where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. The factored form of A is then used to solve the system of equations A * X = B.

Arguments:

a:

Matrix. On entry the N-by-N coefficient matrix, on exit, the factors L and U from the factorization A=P*L*U; the unit diagonal elements of L are not stored.

ipiv:

An integer vector or a null pointer. If not a null pointer, then the pivot indices that define the permutation matrix P, are stored here. Row i of the matrix was interchanged with row IPIV(i).

b:

Matrix, on entry the right hand side matrix should be stored here. On exit, if there was no error, and the info argument is zero, then it contains the solution matrix X.

info:

The LAPACK info code. If it is positive, then U(info,info) is exactly zero. In this case the factorization has been completed, but the factor U is exactly singular, so the solution could not be computed.

Returns:

Error code.

Time complexity: TODO.

Example 26-2. File examples/simple/igraph_lapack_dgesv.c

26.2.2. Eigenvalues and eigenvectors of matrices

26.2.2.1. `igraph_lapack_dsyevr` — Selected eigenvalues and optionally eigenvectors of a symmetric matrix

```
int igraph_lapack_dsyevr(const igraph_matrix_t *A,
    igraph_lapack_dsyev_which_t which,
    igraph_real_t vl, igraph_real_t vu, int vestimate,
```

```
    int il, int iu, igraph_real_t abstol,
    igraph_vector_t *values, igraph_matrix_t *vectors,
    igraph_vector_int_t *support);
```

Calls the DSYEVR LAPACK function to compute selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

See more in the LAPACK documentation.

Arguments:

A:

Matrix, on entry it contains the symmetric input matrix. Only the leading N-by-N upper triangular part is used for the computation.

which:

Constant that gives which eigenvalues (and possibly the corresponding eigenvectors) to calculate. Possible values are `IGRAPH_LAPACK_DSYEV_ALL`, all eigenvalues; `IGRAPH_LAPACK_DSYEV_INTERVAL`, all eigenvalues in the half-open interval (vl,vu]; `IGRAPH_LAPACK_DSYEV_SELECT`, the il-th through iu-th eigenvalues.

vl:

If *which* is `IGRAPH_LAPACK_DSYEV_INTERVAL`, then this is the lower bound of the interval to be searched for eigenvalues. See also the *vestimate* argument.

vu:

If *which* is `IGRAPH_LAPACK_DSYEV_INTERVAL`, then this is the upper bound of the interval to be searched for eigenvalues. See also the *vestimate* argument.

vestimate:

An upper bound for the number of eigenvalues in the (vl,vu] interval, if *which* is `IGRAPH_LAPACK_DSYEV_INTERVAL`. Memory is allocated only for the given number of eigenvalues (and eigenvectors), so this upper bound must be correct.

il:

The index of the smallest eigenvalue to return, if *which* is `IGRAPH_LAPACK_DSYEV_SELECT`.

iu:

The index of the largets eigenvalue to return, if *which* is `IGRAPH_LAPACK_DSYEV_SELECT`.

abstol:

The absolute error tolerance for the eigevalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to abstol + EPS * max(|a|,|b|), where EPS is the machine precision.

values:

An initialized vector, the eigenvalues are stored here, unless it is a null pointer. It will be resized as needed.

vectors:

An initialized matrix, the eigenvectors are stored in its columns, unless it is a null pointer. It will be resized as needed.

support:

An integer vector. If not a null pointer, then it will be resized to (2*max(1,M)) (M is a the total number of eigenvalues found). Then the support of the eigenvectors in *vectors* is stored here, i.e., the indices indicating the nonzero elements in *vectors*. The i-th eigenvector is nonzero only in elements support(2*i-1) through support(2*i).

Returns:

Error code.

Time complexity: TODO.

Example 26-3. File examples/simple/igraph_lapack_dsyevr.c

26.2.2.2. `igraph_lapack_dgeev` — Eigenvalues and optionally eigenvectors of a non-symmetric matrix

```
int igraph_lapack_dgeev(const igraph_matrix_t *A,
   igraph_vector_t *valuesreal,
   igraph_vector_t *valuesimag,
   igraph_matrix_t *vectorsleft,
   igraph_matrix_t *vectorsright,
   int *info);
```

This function calls LAPACK to compute, for an N-by-N real nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors.

The right eigenvector v(j) of A satisfies A * v(j) = lambda(j) * v(j) where lambda(j) is its eigenvalue. The left eigenvector u(j) of A satisfies u(j)**H * A = lambda(j) * u(j)**H where u(j)**H denotes the conjugate transpose of u(j).

The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Arguments:

`A`:

matrix. On entry it contains the N-by-N input matrix.

`valuesreal`:

Pointer to an initialized vector, or a null pointer. If not a null pointer, then the real parts of the eigenvalues are stored here. The vector will be resized as needed.

`valuesimag`:

Pointer to an initialized vector, or a null pointer. If not a null pointer, then the imaginary parts of the eigenvalues are stored here. The vector will be resized as needed.

`vectorsleft`:

Pointer to an initialized matrix, or a null pointer. If not a null pointer, then the left eigenvectors are stored in the columns of the matrix. The matrix will be resized as needed.

`vectorsright`:

Pointer to an initialized matrix, or a null pointer. If not a null pointer, then the right eigenvectors are stored in the columns of the matrix. The matrix will be resized as needed.

`info`:

This argument is used for two purposes. As an input argument it gives whether an igraph error should be generated if the QR algorithm fails to compute all eigenvalues. If *`info`* is non-zero, then an error is generated, otherwise only a warning is given. On exit it contains the LAPACK error code. Zero means successful exit. A negative values means that some of the arguments had an illegal value, this always triggers an igraph error. An i positive value means that the QR algorithm failed to compute all the eigenvalues, and no eigenvectors have been computed; element i+1:N of *`valuesreal`* and *`valuesimag`* contain eigenvalues which have converged. This case only generates an igraph error, if *`info`* was non-zero on entry.

Returns:

Error code.

Time complexity: TODO.

Example 26-4. File examples/simple/igraph_lapack_dgeev.c

26.2.2.3. `igraph_lapack_dgeevx` — Eigenvalues/vectors of nonsymmetric matrices, expert mode

```
int igraph_lapack_dgeevx(igraph_lapack_dgeevx_balance_t balance,
    const igraph_matrix_t *A,
    igraph_vector_t *valuesreal,
    igraph_vector_t *valuesimag,
    igraph_matrix_t *vectorsleft,
    igraph_matrix_t *vectorsright,
    int *ilo, int *ihi, igraph_vector_t *scale,
    igraph_real_t *abnrm,
    igraph_vector_t *rconde,
    igraph_vector_t *rcondv,
    int *info);
```

This function calculates the eigenvalues and optionally the left and/or right eigenvectors of a nonsymmetric N-by-N real matrix.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (*ilo*, \pihi, *scale*, and *abnrm*), reciprocal condition numbers for the eigenvalues (*rconde*), and reciprocal condition numbers for the right eigenvectors (*rcondv*).

The right eigenvector v(j) of A satisfies A * v(j) = lambda(j) * v(j) where lambda(j) is its eigenvalue. The left eigenvector u(j) of A satisfies u(j)**H * A = lambda(j) * u(j)**H where u(j)**H denotes the conjugate transpose of u(j).

The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Balancing a matrix means permuting the rows and columns to make it more nearly upper triangular, and applying a diagonal similarity transformation D * A * D**(-1), where D is a diagonal matrix, to make its rows and columns closer in norm and the condition numbers of its eigenvalues and eigenvectors smaller. The computed reciprocal condition numbers correspond to the balanced matrix. Permuting rows and columns will not change the condition numbers (in exact arithmetic) but diagonal scaling will. For further explanation of balancing, see section 4.10.2 of the LAPACK Users' Guide.

Arguments:

`balance`:

Scalar that indicated, whether the input matrix should be balanced. Possible values:

`IGRAPH_LAPACK_DGEEVX_BALANCE_NONE`

no not diagonally scale or permute.

`IGRAPH_LAPACK_DGEEVX_BALANCE_PERM`

perform permutations to make the matrix more nearly upper triangular. Do not diagonally scale.

`IGRAPH_LAPACK_DGEEVX_BALANCE_SCALE`

diagonally scale the matrix, i.e. replace A by D*A*D**(-1), where D is a diagonal matrix, chosen to make the rows and columns of A more equal in norm. Do not permute.

`IGRAPH_LAPACK_DGEEVX_BALANCE_BOTH`

both diagonally scale and permute A.

`A`:

The input matrix, must be square.

`valuesreal`:

An initialized vector, or a NULL pointer. If not a NULL pointer, then the real parts of the eigenvalues are stored here. The vector will be resized, as needed.

`valuesimag`:

An initialized vector, or a NULL pointer. If not a NULL pointer, then the imaginary parts of the eigenvalues are stored here. The vector will be resized, as needed.

`vectorsleft`:

An initialized matrix or a NULL pointer. If not a null pointer, then the left eigenvectors are stored here. The order corresponds to the eigenvalues and the eigenvectors are stored in a compressed form. If the j-th eigenvalue is real then column j contains the corresponding eigenvector. If the j-th and (j+1)-th eigenvalues form a complex conjugate pair, then the j-th and (j+1)-th columns contain their corresponding eigenvectors.

vectorsright:

An initialized matrix or a NULL pointer. If not a null pointer, then the right eigenvectors are stored here. The format is the same, as for the *vectorsleft* argument.

ilo:

ihi:

ilo and *ihi* are integer values determined when A was balanced. The balanced A(i,j) = 0 if I>J and J=1,...,ilo-1 or I=ihi+1,...,N.

scale:

Pointer to an initialized vector or a NULL pointer. If not a NULL pointer, then details of the permutations and scaling factors applied when balancing

A,:

are stored here. If P(j) is the index of the row and column interchanged with row and column j, and D(j) is the scaling factor applied to row and column j, then

```
scale(J) = P(J), for J = 1,...,ilo-1
```

```
scale(J) = D(J), for J = ilo,...,ihi
```

```
scale(J) = P(J) for J = ihi+1,...,N.
```

The order in which the interchanges are made is N to *ihi*+1, then 1 to *ilo*-1.

abnrm:

Pointer to a real variable, the one-norm of the balanced matrix is stored here. (The one-norm is the maximum of the sum of absolute values of elements in any column.)

rconde:

An initialized vector or a NULL pointer. If not a null pointer, then the reciprocal condition numbers of the eigenvalues are stored here.

rcondv:

An initialized vector or a NULL pointer. If not a null pointer, then the reciprocal condition numbers of the right eigenvectors are stored here.

info:

This argument is used for two purposes. As an input argument it gives whether an igraph error should be generated if the QR algorithm fails to compute all eigenvalues. If *info* is non-zero, then an error is generated, otherwise only a warning is given. On exit it contains the LAPACK error code. Zero means successful exit. A negative values means that some of the arguments had an illegal value, this always triggers an igraph error. An i positive value means that the QR algorithm failed to compute all the eigenvalues, and no eigenvectors have been computed; element i+1:N of *valuesreal* and *valuesimag* contain eigenvalues which have converged. This case only generated an igraph error, if *info* was non-zero on entry.

Returns:

Error code.

Time complexity: TODO

Example 26-5. File examples/simple/igraph_lapack_dgeevx.c

26.3. ARPACK interface in igraph

ARPACK is a library for solving large scale eigenvalue problems. The package is designed to compute a few eigenvalues and corresponding eigenvectors of a general `n` by `n` matrix `A`. It is most appropriate for large sparse or structured matrices `A` where structured means that a matrix-vector product `w <- Av` requires order `n` rather than the usual order `n^2` floating point operations. Please see http://www.caam.rice.edu/software/ARPACK/ for details.

The eigenvalue calculation in ARPACK (in the simplest case) involves the calculation of the `Av` product where `A` is the matrix we work with and `v` is an arbitrary vector. A user-defined function of type `igraph_arpack_function_t` is expected to perform this product. If the product can be done efficiently, e.g. if the matrix is sparse, then ARPACK is usually able to calculate the eigenvalues very quickly.

In igraph, eigenvalue/eigenvector calculations usually involve the following steps:

1. Initialization of an `igraph_arpack_options_t` data structure using `igraph_arpack_options_init`.
2. Setting some options in the initialized `igraph_arpack_options_t` object.
3. Defining a function of type `igraph_arpack_function_t`. The input of this function is a vector, and the output should be the output matrix multiplied by the input vector.

4. Calling `igraph_arpack_rssolve()` (is the matrix is symmetric), or `igraph_arpack_rnsolve()`.

The `igraph_arpack_options_t` object can be used multiple times.

If we have many eigenvalue problems to solve, then it might worth to create an `igraph_arpack_storage_t` object, and initialize it via `igraph_arpack_storage_init()`. This structure contains all memory needed for ARPACK (with the given upper limit regerding to the size of the eigenvalue problem). Then many problems can be solved using the same `igraph_arpack_storage_t` object, without always reallocating the required memory. The `igraph_arpack_storage_t` object needs to be destroyed by calling `igraph_arpack_storage_destroy()` on it, when it is not needed any more.

igraph does not contain all ARPACK routines, only the ones dealing with symmetric and non-symmetric eigenvalue problems using double precision real numbers.

26.3.1. Data structures

26.3.1.1. `igraph_arpack_options_t` — Options for ARPACK

```
typedef struct igraph_arpack_options_t {
  /* INPUT */
  char bmat[1];   /* I-standard problem, G-generalized */
  int n;    /* Dimension of the eigenproblem */
  char which[2];  /* LA, SA, LM, SM, BE */
  int nev;                 /* Number of eigenvalues to be computed */
  igraph_real_t tol;  /* Stopping criterion */
  int ncv;   /* Number of columns in V */
  int ldv;   /* Leading dimension of V */
  int ishift;  /* 0-reverse comm., 1-exact with tridiagonal */
  int mxiter;              /* Maximum number of update iterations to take */
  int nb;   /* Block size on the recurrence, only 1 works */
  int mode;  /* The kind of problem to be solved (1-5)
       1: A*x=l*x, A symmetric
       2: A*x=l*M*x, A symm. M pos. def.
       3: K*x = l*M*x, K symm., M pos. semidef.
       4: K*x = l*KG*x, K s. pos. semidef. KG s. indef.
       5: A*x = l*M*x, A symm., M symm. pos. semidef. */
  int start;  /* 0: random, 1: use the supplied vector */
  int lworkl;  /* Size of temporary storage, default is fine */
  igraph_real_t sigma;          /* The shift for modes 3,4,5 */
  igraph_real_t sigmai;  /* The imaginary part of shift for rnsolve */
  /* OUTPUT */
  int info;  /* What happened, see docs */
  int ierr;  /* What happened  in the dseupd call */
  int noiter;  /* The number of iterations taken */
  int nconv;
```

```
  int numop;  /* Number of OP*x operations */
  int numopb;  /* Number of B*x operations if BMAT='G' */
  int numreo;  /* Number of steps of re-orthogonalizations */
  /* INTERNAL */
  int iparam[11];
  int ipntr[14];
} igraph_arpack_options_t;
```

This data structure contains the options of thee ARPACK eigenvalue solver routines. It must be initialized by calling `igraph_arpack_options_init()` on it. Then it can be used for multiple ARPACK calls, as the ARPACK solvers do not modify it. Input options:

Values:

`bmat`:

Character. Whether to solve a standard ('I') ot a generalized problem ('B').

`n`:

Dimension of the eigenproblem.

`which`:

Specifies which eigenvalues/vectors to compute. Possible values for symmetric matrices:

`LA`

Compute `nev` largest (algebraic) eigenvalues.

`SA`

Compute `nev` smallest (algebraic) eigenvalues.

`LM`

Compute `nev` largest (in magnitude) eigenvalues.

`SM`

Compute `nev` smallest (in magnitude) eigenvalues.

`BE`

Compute `nev` eigenvalues, half from each end of the spectrum. When `nev` is odd, compute one more from the high en than from the low end.

Possible values for non-symmetric matrices:

`LM`

Compute `nev` largest (in magnitude) eigenvalues.

`SM`

Compute `nev` smallest (in magnitude) eigenvalues.

`LR`

Compute `nev` eigenvalues of largest real part.

`SR`

Compute `nev` eigenvalues of smallest real part.

`LI`

Compute `nev` eigenvalues of largest imaginary part.

`SI`

Compute `nev` eigenvalues of smallest imaginary part.

`nev`:

The number of eigenvalues to be computed.

`tol`:

Stopping criterion: the relative accuracy of the Ritz value is considered acceptable if its error is less than `tol` times its estimated value. If this is set to zero then machine precision is used.

`ncv`:

Number of Lanczos vectors to be generated. Setting this to zero means that `igraph_arpack_rssolve` and `igraph_arpack_rnsolve` will determine a suitable value for `ncv` automatically.

`ldv`:

Numberic scalar. It should be set to zero in the current igraph implementation.

`ishift`:

Either zero or one. If zero then the shifts are provided by the user via reverse communication. If one then exact shifts with respect to the reduced tridiagonal matrix `T`. Please always set this to one.

`mxiter`:

Maximum number of Arnoldi update iterations allowed.

`nb`:

Blocksize to be used in the recurrence. Please always leave this on the default value, one.

`mode`:

The type of the eigenproblem to be solved. Possible values if the input matrix is symmetric:

1. A*x=lambda*x, A is symmetric.
2. A*x=lambda*M*x, A is symmetric, M is symmetric positive definite.

3. K*x=lambda*M*x, K is symmetric, M is symmetric positive semi-definite.
4. K*x=lambda*KG*x, K is symmetric positive semi-definite, KG is symmetric indefinite.
5. A*x=lambda*M*x, A is symmetric, M is symmetric positive semi-definite. (Cayley transformed mode.)

Please note that only `mode` ==1 was tested and other values might not work properly. Possible values if the input matrix is not symmetric:

1. A*x=lambda*x.
2. A*x=lambda*M*x, M is symmetric positive definite.
3. A*x=lambda*M*x, M is symmetric semi-definite.
4. A*x=lambda*M*x, M is symmetric semi-definite.

Please note that only `mode` == 1 was tested and other values might not work properly.

`start`:

Whether to use the supplied starting vector (1), or use a random starting vector (0). The starting vector must be supplied in the first column of the `vectors` argument of the `igraph_arpack_rssolve()` of `igraph_arpack_rnsolve()` call.

Output options:

Values:

`info`:

Error flag of ARPACK. Possible values:

`0`

Normal exit.

`1`

Maximum number of iterations taken.

`3`

No shifts could be applied during a cycle of the Implicitly restarted Arnoldi iteration. One possibility is to increase the size of `ncv` relative to `nev`.

ARPACK can return other error flags as well, but these are converted to igraph errors, see `igraph_error_type_t`.

`ierr`:

Error flag of the second ARPACK call (one eigenvalue computation usually involves two calls to ARPACK). This is always zero, as other error codes are converted to igraph errors.

`noiter`:

Number of Arnoldi iterations taken.

`nconv`:

Number of converged Ritz values. This represents the number of Ritz values that satisfy the convergence critetion.

`numop`:

Total number of matrix-vector multiplications.

`numopb`:

Not used currently.

`numreo`:

Total number of steps of re-orthogonalization.

Internal options:

Values:

`lworkl`:

Do not modify this option.

`sigma`:

The shift for the shift-invert mode.

`sigmai`:

The imaginary part of the shift, for the non-symmetric or complex shift-invert mode.

`iparam`:

Do not modify this option.

`ipntr`:

Do not modify this option.

26.3.1.2. `igraph_arpack_storage_t` — Storage for ARPACK

```
typedef struct igraph_arpack_storage_t {
  int maxn, maxncv, maxldv;
  igraph_real_t *v;
```

```
  igraph_real_t *workl;
  igraph_real_t *workd;
  igraph_real_t *d;
  igraph_real_t *resid;
  igraph_real_t *ax;
  int *select;
  igraph_real_t *di;  /* These two only for non-symmetric problems */
  igraph_real_t *workev;
} igraph_arpack_storage_t;
```

Public members, do not modify them directly, these are considered to be read-only.

Values:

`maxn`:

Maximum rank of matrix.

`maxncv`:

Maximum NCV.

`maxldv`:

Maximum LDV.

These members are considered to be private:

Values:

`workl`:

Working memory.

`workd`:

Working memory.

`d`:

Memory for eigenvalues.

`resid`:

Memory for residuals.

`ax`:

Working memory.

`select`:

Working memory.

`di`:

Memory for eigenvalues, non-symmetric case only.

`workev`:

Working memory, non-symmetric case only.

26.3.1.3. `igraph_arpack_function_t` — Type of the ARPACK callback function

```
typedef int igraph_arpack_function_t(igraph_real_t *to, const igraph_real_t *from,
         int n, void *extra);
```

Arguments:

`to`:

Pointer to an `igraph_real_t`, the result of the matrix-vector product is expected to be stored here.

`from`:

Pointer to an `igraph_real_t`, the input matrix should be multiplied by the vector stored here.

`n`:

The length of the vector (which is the same as the order of the input matrix).

`extra`:

Extra argument to the matrix-vector calculation function. This is coming from the `igraph_arpack_rssolve()` or `igraph_arpack_rnsolve()` function.

Returns:

Error code, if not zero, then the ARPACK solver considers this as an error, stops and calls the igraph error handler.

26.3.1.4. `igraph_arpack_options_init` — Initialize ARPACK options

```
void igraph_arpack_options_init(igraph_arpack_options_t *o);
```

Initializes ARPACK options, set them to default values. You can always pass the initialized `igraph_arpack_options_t` object to built-in igraph functions without any modification. The built-in igraph functions modify the options to perform their calculation, e.g. `igraph_pagerank()` always searches for the eigenvalue with the largest magnitude, regardless of the supplied value.

If you want to implement your own function involving eigenvalue calculation using ARPACK, however, you will likely need to set up the fields for yourself.

Arguments:

o:

The `igraph_arpack_options_t` object to initialize.

Time complexity: O(1).

26.3.1.5. `igraph_arpack_storage_init` — Initialize ARPACK storage

```
int igraph_arpack_storage_init(igraph_arpack_storage_t *s, long int maxn,
          long int maxncv, long int maxldv,
          igraph_bool_t symm);
```

You only need this function if you want to run multiple eigenvalue calculations using ARPACK, and want to spare the memory allocation/deallocation between each two runs. Otherwise it is safe to supply a null pointer as the `storage` argument of both `igraph_arpack_rssolve()` and `igraph_arpack_rnsolve()` to make memory allocated and deallocated automatically.

Don't forget to call the `igraph_arpack_storage_destroy()` function on the storage object if you don't need it any more.

Arguments:

s:

The `igraph_arpack_storage_t` object to initialize.

maxn:

The maximum order of the matrices.

maxncv:

The maximum NCV parameter intended to use.

maxldv:

The maximum LDV parameter intended to use.

symm:

Whether symmetric or non-symmetric problems will be solved using this `igraph_arpack_storage_t`. (You cannot use the same storage both with symmetric and non-symmetric solvers.)

Returns:

Error code.

Time complexity: O(maxncv*(maxldv+maxn)).

26.3.1.6. `igraph_arpack_storage_destroy` — Deallocate ARPACK storage

```
void igraph_arpack_storage_destroy(igraph_arpack_storage_t *s);
```

Arguments:

s:

The `igraph_arpack_storage_t` object for which the memory will be deallocated.

Time complexity: operating system dependent.

26.3.2. ARPACK solvers

26.3.2.1. `igraph_arpack_rssolve` — ARPACK solver for symmetric matrices

```
int igraph_arpack_rssolve(igraph_arpack_function_t *fun, void *extra,
     igraph_arpack_options_t *options,
     igraph_arpack_storage_t *storage,
     igraph_vector_t *values, igraph_matrix_t *vectors);
```

This is the ARPACK solver for symmetric matrices. Please use `igraph_arpack_rnsolve()` for non-symmetric matrices.

Arguments:

`fun`:

Pointer to an `igraph_arpack_function_t` object, the function that performs the matrix-vector multiplication.

`extra`:

An extra argument to be passed to `fun`.

`options`:

An `igraph_arpack_options_t` object.

`storage`:

An `igraph_arpack_storage_t` object, or a null pointer. In the latter case memory allocation and deallocation is performed automatically. Either this or the *`vectors`* argument must be non-null if the ARPACK iteration is started from a given starting vector. If both are given *`vectors`* take precedence.

`values`:

If not a null pointer, then it should be a pointer to an initialized vector. The eigenvalues will be stored here. The vector will be resized as needed.

`vectors`:

If not a null pointer, then it must be a pointer to an initialized matrix. The eigenvectors will be stored in the columns of the matrix. The matrix will be resized as needed. Either this or the *`vectors`* argument must be non-null if the ARPACK iteration is started from a given starting vector. If both are given *`vectors`* take precedence.

Returns:

Error code.

Time complexity: depends on the matrix-vector multiplication. Usually a small number of iterations is enough, so if the matrix is sparse and the matrix-vector multiplication can be done in O(n) time (the number of vertices), then the eigenvalues are found in O(n) time as well.

26.3.2.2. `igraph_arpack_rnsolve` — ARPACK solver for non-symmetric matrices

```
int igraph_arpack_rnsolve(igraph_arpack_function_t *fun, void *extra,
     igraph_arpack_options_t *options,
     igraph_arpack_storage_t *storage,
     igraph_matrix_t *values, igraph_matrix_t *vectors);
```

Please always consider calling `igraph_arpack_rssolve()` if your matrix is symmetric, it is much faster. `igraph_arpack_rnsolve()` for non-symmetric matrices.

Note that ARPACK is not called for 2x2 matrices as an exact algebraic solution exists in these cases.

Arguments:

fun:

Pointer to an `igraph_arpack_function_t` object, the function that performs the matrix-vector multiplication.

extra:

An extra argument to be passed to `fun`.

options:

An `igraph_arpack_options_t` object.

storage:

An `igraph_arpack_storage_t` object, or a null pointer. In the latter case memory allocation and deallocation is performed automatically.

values:

If not a null pointer, then it should be a pointer to an initialized matrix. The (possibly complex) eigenvalues will be stored here. The matrix will have two columns, the first column contains the real, the second the imaginary parts of the eigenvalues. The matrix will be resized as needed.

vectors:

If not a null pointer, then it must be a pointer to an initialized matrix. The eigenvectors will be stored in the columns of the matrix. The matrix will be resized as needed.

Returns:

Error code.

Time complexity: depends on the matrix-vector multiplication. Usually a small number of iterations is enough, so if the matrix is sparse and the matrix-vector multiplication can be done in O(n) time (the number of vertices), then the eigenvalues are found in O(n) time as well.

26.3.2.3. `igraph_arpack_unpack_complex` — Make the result of the non-symmetric ARPACK solver more readable

```
int igraph_arpack_unpack_complex(igraph_matrix_t *vectors, igraph_matrix_t *values,
     long int nev);
```

This function works on the output of `igraph_arpack_rnsolve` and brushes it up a bit: it only keeps *nev* eigenvalues/vectors and every eigenvector is stored in two columns of the *vectors* matrix.

The output of the non-symmetric ARPACK solver is somewhat hard to parse, as real eigenvectors occupy only one column in the matrix, and the complex conjugate eigenvectors are not stored at all (usually). The other problem is that the solver might return more eigenvalues than requested. The common use of this function is to call it directly after `igraph_arpack_rnsolve` with its *vectors* and *values* argument and `options`->nev as *nev*.

Arguments:

vectors:

The eigenvector matrix, as returned by `igraph_arpack_rnsolve`. It will be resized, typically it will be larger.

values:

The eigenvalue matrix, as returned by `igraph_arpack_rnsolve`. It will be resized, typically extra, unneeded rows (=eigenvalues) will be removed.

`nev`:

The number of eigenvalues/vectors to keep. Can be less or equal than the number originally requested from ARPACK.

Returns:

Error code.

Time complexity: linear in the number of elements in the `vectors` matrix.

Chapter 27. Bipartite, i.e. two-mode graphs

27.1. Bipartite networks in igraph

A bipartite network contains two kinds of vertices and connections are only possible between two vertices of different kind. There are many natural examples, e.g. movies and actors as vertices and a movie is connected to all participating actors, etc.

igraph does not have direct support for bipartite networks, at least not at the C language level. In other words the igraph_t structure does not contain information about the vertex types. The C functions for bipartite networks usually have an additional input argument to graph, called `types`, a boolean vector giving the vertex types.

Most functions creating bipartite networks are able to create this extra vector, you just need to supply an initialized boolean vector to them.

27.2. Create two-mode networks

27.2.1. `igraph_create_bipartite` — Create a bipartite graph

```
int igraph_create_bipartite(igraph_t *graph, const igraph_vector_bool_t *types,
       const igraph_vector_t *edges,
       igraph_bool_t directed);
```

This is a simple wrapper function to create a bipartite graph. It does a little more than `igraph_create()`, e.g. it checks that the graph is indeed bipartite with respect to the given *`types`* vector. If there is an edge connecting two vertices of the same kind, then an error is reported.

Arguments:

`graph`:

Pointer to an uninitialized graph object, the result is created here.

`types`:

Boolean vector giving the vertex types. The length of the vector defines the number of vertices in the graph.

edges:

Vector giving the edges of the graph. The highest vertex id in this vector must be smaller than the length of the *types* vector.

directed:

Boolean scalar, whether to create a directed graph.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

Example 27-1. File examples/simple/igraph_bipartite_create.c

27.2.2. `igraph_full_bipartite` — Create a full bipartite network

```
int igraph_full_bipartite(igraph_t *graph,
     igraph_vector_bool_t *types,
     igraph_integer_t n1, igraph_integer_t n2,
     igraph_bool_t directed,
     igraph_neimode_t mode);
```

A bipartite network contains two kinds of vertices and connections are only possible between two vertices of different kind. There are many natural examples, e.g. movies and actors as vertices and a movie is connected to all participating actors, etc.

igraph does not have direct support for bipartite networks, at least not at the C language level. In other words the igraph_t structure does not contain information about the vertex types. The C functions for bipartite networks usually have an additional input argument to graph, called `types`, a boolean vector giving the vertex types.

Most functions creating bipartite networks are able to create this extra vector, you just need to supply an initialized boolean vector to them.

Arguments:

`graph`:

Pointer to an igraph_t object, the graph will be created here.

`types`:

Pointer to a boolean vector. If not a null pointer, then the vertex types will be stored here.

`n1`:

Integer, the number of vertices of the first kind.

`n2`:

Integer, the number of vertices of the second kind.

`directed`:

Boolean, whether to create a directed graph.

`mode`:

A constant that gives the type of connections for directed graphs. If `IGRAPH_OUT`, then edges point from vertices of the first kind to vertices of the second kind; if `IGRAPH_IN`, then the opposite direction is realized; if `IGRAPH_ALL`, then mutual edges will be created.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

See also:

`igraph_full()` for non-bipartite full graphs.

27.3. Incidence matrices

27.3.1. `igraph_incidence` — Create a bipartite graph from an incidence matrix

```
int igraph_incidence(igraph_t *graph, igraph_vector_bool_t *types,
       const igraph_matrix_t *incidence,
       igraph_bool_t directed,
       igraph_neimode_t mode, igraph_bool_t multiple);
```

A bipartite (or two-mode) graph contains two types of vertices and edges always connect vertices of different types. An incidence matrix is an nxm matrix, n and m are the number of vertices of the two types, respectively. Nonzero elements in the matrix denote edges between the two corresponding vertices.

Note that this function can operate in two modes, depending on the *`multiple`* argument. If it is FALSE (i.e. 0), then a single edge is created for every non-zero element in the incidence matrix. If *`multiple`* is TRUE (i.e. 1), then the matrix elements are rounded up to the closest non-negative integer to get the number of edges to create between a pair of vertices.

This function does not create multiple edges if *`multiple`* is FALSE, but might create some if it is TRUE.

Arguments:

`graph`:

Pointer to an uninitialized graph object.

`types`:

Pointer to an initialized boolean vector, or a null pointer. If not a null pointer, then the vertex types are stored here. It is resized as needed.

`incidence`:

The incidence matrix.

`directed`:

Gives whether to create an undirected or a directed graph.

`mode`:

Specifies the direction of the edges in a directed graph. If `IGRAPH_OUT`, then edges point from vertices of the first kind (corresponding to rows) to vertices of the second kind (corresponding to columns); if `IGRAPH_IN`, then the opposite direction is realized; if `IGRAPH_ALL`, then mutual edges will be created.

multiple:

How to interpret the incidence matrix elements. See details below.

Returns:

Error code.

Time complexity: O(n*m), the size of the incidence matrix.

27.3.2. `igraph_get_incidence` — Convert a bipartite graph into an incidence matrix

```
int igraph_get_incidence(const igraph_t *graph,
    const igraph_vector_bool_t *types,
    igraph_matrix_t *res,
    igraph_vector_t *row_ids,
    igraph_vector_t *col_ids);
```

Arguments:

graph:

The input graph, edge directions are ignored.

types:

Boolean vector containing the vertex types.

res:

Pointer to an initialized matrix, the result is stored here. An element of the matrix gives the number of edges (irrespectively of their direction) between the two corresponding vertices.

row_ids:

Pointer to an initialized vector or a null pointer. If not a null pointer, then the vertex ids (in the graph) corresponding to the rows of the result matrix are stored here.

col_ids:

Pointer to an initialized vector or a null pointer. If not a null pointer, then the vertex ids corresponding to the columns of the result matrix are stored here.

Returns:

Error code.

Time complexity: O(n*m), n and m are number of vertices of the two different kind.

See also:

`igraph_incidence()` for the opposite operation.

27.4. Project a two-mode graphs

27.4.1. `igraph_bipartite_projection_size` — Calculate the number of vertices and edges in the bipartite projections

```
int igraph_bipartite_projection_size(const igraph_t *graph,
         const igraph_vector_bool_t *types,
         igraph_integer_t *vcount1,
         igraph_integer_t *ecount1,
         igraph_integer_t *vcount2,
         igraph_integer_t *ecount2);
```

This function calculates the number of vertices and edges in the two projections of a bipartite network. This is useful if you have a big bipartite network and you want to estimate the amount of memory you would need to calculate the projections themselves.

Arguments:

`graph`:

The input graph.

`types`:

Boolean vector giving the vertex types of the graph.

`vcount1`:

Pointer to an `igraph_integer_t`, the number of vertices in the first projection is stored here.

ecount1:

Pointer to an `igraph_integer_t`, the number of edges in the first projection is stored here.

vcount2:

Pointer to an `igraph_integer_t`, the number of vertices in the second projection is stored here.

ecount2:

Pointer to an `igraph_integer_t`, the number of edges in the second projection is stored here.

Returns:

Error code.

See also:

`igraph_bipartite_projection()` to calculate the actual projection.

Time complexity: O(|V|*d^2+|E|), |V| is the number of vertices, |E| is the number of edges, d is the average (total) degree of the graphs.

Example 27-2. File examples/simple/igraph_bipartite_projection.c

27.4.2. `igraph_bipartite_projection` — Create one or both projections of a bipartite (two-mode) network

```
int igraph_bipartite_projection(const igraph_t *graph,
    const igraph_vector_bool_t *types,
    igraph_t *proj1,
    igraph_t *proj2,
    igraph_vector_t *multiplicity1,
    igraph_vector_t *multiplicity2,
    igraph_integer_t probe1);
```

Creates one or both projections of a bipartite graph.

Arguments:

graph:

The bipartite input graph. Directedness of the edges is ignored.

types:

Boolean vector giving the vertex types of the graph.

proj1:

Pointer to an uninitialized graph object, the first projection will be created here. It a null pointer, then it is ignored, see also the *probe1* argument.

multiplicity1:

Pointer to a vector, or a null pointer. If not the latter, then the multiplicity of the edges is stored here. E.g. if there is an A-C-B and also an A-D-B triple in the bipartite graph (but no more X, such that A-X-B is also in the graph), then the multiplicity of the A-B edge in the projection will be 2.

multiplicity2:

The same as `multiplicity1`, but for the other projection.

proj2:

Pointer to an uninitialized graph object, the second projection is created here, if it is not a null pointer. See also the *probe1* argument.

Returns:

Error code.

See also:

`igraph_bipartite_projection_size()` to calculate the number of vertices and edges in the projections, without creating the projection graphs themselves.

Time complexity: O(|V|*d^2+|E|), |V| is the number of vertices, |E| is the number of edges, d is the average (total) degree of the graphs.

Example 27-3. File examples/simple/igraph_bipartite_projection.c

27.5. Other operations on bipartite graphs

27.5.1. `igraph_is_bipartite` — Check whether a graph is bipartite

```
int igraph_is_bipartite(const igraph_t *graph,
   igraph_bool_t *res,
   igraph_vector_bool_t *type);
```

This function simply checks whether a graph \emph{could} be bipartite. It tries to find a mapping that gives a possible division of the vertices into two classes, such that no two vertices of the same class are connected by an edge.

The existence of such a mapping is equivalent of having no circuits of odd length in the graph. A graph with loop edges cannot bipartite.

Note that the mapping is not necessarily unique, e.g. if the graph has at least two components, then the vertices in the separate components can be mapped independently.

Arguments:

graph:

The input graph.

res:

Pointer to a boolean, the result is stored here.

type:

Pointer to an initialized boolean vector, or a null pointer. If not a null pointer and a mapping was found, then it is stored here. If not a null pointer, but no mapping was found, the contents of this vector is invalid.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

Chapter 28. Advanced igraph programming

28.1. Using igraph in multi-threaded programs

The igraph library is considered thread-safe on platforms that support thread-local storage. This currently includes Linux and MS Windows operating systems, but not Mac OSX. The best way to check whether an igraph build is thread-safe is checking the `IGRAPH_THREAD_SAFE` macro.

28.1.1. `IGRAPH_THREAD_SAFE` — Macro that is defined to be 1 if the current build of the

```
#define IGRAPH_THREAD_SAFE
```

igraph library is thread-safe, and 0 if it is not.

28.1.2. Thread-safe ARPACK library

Note that igraph is only thread-safe if it was built with the internal ARPACK library, i.e. the one that comes with igraph. The standard ARPACK library is not thread-safe.

28.2. Progress handlers

28.2.1. About progress handlers

It is often useful to report the progress of some long calculation, to allow the user to follow the computation and guess the total running time. A couple of igraph functions support this at the time of writing, hopefully more will support it in the future.

To see the progress of a computation, the user has to install a progress handler, as there is none installed by default. If an igraph function supports progress reporting, then it calls the installed progress handler periodically, and passes a percentage value to it, the percentage of computation already performed. To install a progress handler, you need to call `igraph_set_progress_handler()`. Currently there is a single pre-defined progress handler, called `igraph_progress_handler_stderr()`.

28.2.2. Setting up progress handlers

28.2.2.1. `igraph_progress_handler_t` — Type of progress handler functions

```
typedef int igraph_progress_handler_t(const char *message, igraph_real_t percent,
          void *data);
```

This is the type of the igraph progress handler functions. There is currently one such predefined function, `igraph_progress_handler_stderr()`, but the user can write and set up more sophisticated ones.

Arguments:

message:

A string describing the function or algorithm that is reporting the progress. Current igraph functions always use the name *message* argument if reporting from the same function.

percent:

Numeric, the percentage that was completed by the algorithm or function.

data:

User-defined data. Current igraph functions that report progress pass a null pointer here. Users can write their own progress handlers and functions with progress reporting, and then pass some meaningfull context here.

Returns:

If the return value of the progress handler is not IGRAPH_SUCCESS (=0), then `igraph_progress()` returns the error code `IGRAPH_INTERRUPTED`. The `IGRAPH_PROGRESS()` macro frees all memory and finishes the igraph function with error code `IGRAPH_INTERRUPTED` in this case.

28.2.2.2. `igraph_set_progress_handler` — Install a progress handler, or remove the current handler

```
igraph_progress_handler_t *
igraph_set_progress_handler(igraph_progress_handler_t new_handler);
```

There is a single simple predefined progress handler: `igraph_progress_handler_stderr()`.

Arguments:

`new_handler`:

Pointer to a function of type `igraph_progress_handler_t`, the progress handler function to install. To uninstall the current progress handler, this argument can be a null pointer.

Returns:

Pointer to the previously installed progress handler function.

Time complexity: O(1).

28.2.2.3. `igraph_progress_handler_stderr` — A simple predefined progress handler

```
int igraph_progress_handler_stderr(const char *message, igraph_real_t percent,
      void* data);
```

This simple progress handler first prints *`message`*, and then the percentage complete value in a short message to standard error.

Arguments:

`message`:

A string describing the function or algorithm that is reporting the progress. Current igraph functions always use the name *`message`* argument if reporting from the same function.

`percent`:

Numeric, the percentage that was completed by the algorithm or function.

`data`:

User-defined data. Current igraph functions that report progress pass a null pointer here. Users can write their own progress handlers and functions with progress reporting, and then pass some meaningfull context here.

Returns:

This function always returns with `IGRAPH_SUCCESS`.

Time complexity: O(1).

28.2.3. Invoking the progress handler

28.2.3.1. `IGRAPH_PROGRESS` — Report progress.

```
#define IGRAPH_PROGRESS(message,
```

The standard way to report progress from an igraph function

Arguments:

`message`:

A string, a textual message that references the calculation under progress.

`percent`:

Numeric scalar, the percentage that is complete.

`data`:

User-defined data, this can be used in user-defined progress handler functions, from user-written igraph functions.

Returns:

If the progress handler returns with `IGRAPH_INTERRUPTED`, then this macro frees up the igraph allocated memory for temporary data and returns to the caller with `IGRAPH_INTERRUPTED`.

28.2.3.2. `igraph_progress` — Report progress

```
int igraph_progress(const char *message, igraph_real_t percent, void *data);
```

Note that the usual way to report progress is the `IGRAPH_PROGRESS` macro, as that takes care of the return value of the progress handler.

Arguments:

`message`:

A string describing the function or algorithm that is reporting the progress. Current igraph functions always use the name *`message`* argument if reporting from the same function.

`percent`:

Numeric, the percentage that was completed by the algorithm or function.

`data`:

User-defined data. Current igraph functions that report progress pass a null pointer here. Users can write their own progress handlers and functions with progress reporting, and then pass some meaningfull context here.

Returns:

If there is a progress handler installed and it does not return `IGRAPH_SUCCESS`, then `IGRAPH_INTERRUPTED` is returned.

Time complexity: O(1).

28.2.3.3. `igraph_progressf` — Report progress, printf-like version

```
int igraph_progressf(const char *message, igraph_real_t percent, void *data,
        ...);
```

This is a more flexible version of `igraph_progress()`, with a printf-like template string. First the template string is filled with the additional arguments and then `igraph_progress()` is called.

Note that there is an upper limit for the length of the *`message`* string, currently 1000 characters.

Arguments:

`message`:

A string describing the function or algorithm that is reporting the progress. For this function this is a template string, using the same syntax as the standard `libc` `printf` function.

`percent`:

Numeric, the percentage that was completed by the algorithm or function.

`data`:

User-defined data. Current igraph functions that report progress pass a null pointer here. Users can write their own progress handlers and functions with progress reporting, and then pass some meaningfull context here.

`...`:

Additional argument that were specified in the *message* argument.

Returns:

If there is a progress handler installed and it does not return `IGRAPH_SUCCESS`, then `IGRAPH_INTERRUPTED` is returned. \return

28.2.4. Writing progress handlers

To write a new progress handler, one needs to create a function of type `igraph_progress_handler_t`. The new progress handler can then be installed with the `igraph_set_progress_handler()` function.

One can assume that the first progress handler call from a calculation will be call with zero as the *percentage* argument, and the last call from a function will have 100 as the *percentage* argument. Note, however, that if an error happens in the middle of a computation, then the 100 percent call might be omitted.

28.2.5. Writing igraph functions with progress reporting

If you want to write a function that uses igraph and supports progress reporting, you need to include `igraph_progress()` calls in your function, usually via the `IGRAPH_PROGRESS()` macro.

It is good practice to always include a call to `igraph_progress()` with a zero *percentage* argument, before the computation; and another call with 100 *percentage* value after the computation is completed.

It is also good practice *not* to call `igraph_progress()` too often, as this would slow down the computation. It might not be worth to support progress reporting in functions with linear or log-linear time complexity, as these are fast, even with a large amount of data. For functions with quadratic or higher time complexity make sure that the time complexity of the progress reporting is constant or at least linear. In practice this means having at most O(n) progress checks and at most 100 \reg igraph_progress() calls.

28.2.6. Multi-threaded programs

In multi-threaded programs, each thread has its own progress handler, if thread-local storage is supported and igraph is thread-safe. See the `IGRAPH_THREAD_SAFE` macro for checking whether an igraph build is thread-safe.

28.3. Status handlers

28.3.1. Status reporting

In addition to the possibility of reporting the progress of an igraph computation via `igraph_progress()`, it is also possible to report simple status messages from within igraph functions, without having to judge how much of the computation was performed already. For this one needs to install a status handler function.

Status handler functions must be of type `igraph_status_handler_t` and they can be install by a call to `igraph_set_status_handler()`. Currently there is a simple predefined status handler function, called `igraph_status_handler_stderr()`, but the user can define new ones.

Igraph functions report their status via a call to the `IGRAPH_STATUS()` or the `IGRAPH_STATUSF()` macro.

28.3.2. Setting up status handlers

28.3.2.1. `igraph_status_handler_t` — The type of the igraph status handler functions

```
typedef int igraph_status_handler_t(const char *message, void *data);
```

Arguments:

message:

The status message.

data:

Additional context, with user-defined semantics. Existing igraph functions pass a null pointer here.

28.3.2.2. `igraph_set_status_handler` — Install of uninstall a status handler function.

```
igraph_status_handler_t *
igraph_set_status_handler(igraph_status_handler_t new_handler);
```

To uninstall the currently installed status handler, call this function with a null pointer.

Arguments:

new_handler:

The status handler function to install.

Returns:

The previously installed status handler function.

Time complexity: O(1).

28.3.2.3. `igraph_status_handler_stderr` — A simple predefined status handler function.

```
int igraph_status_handler_stderr(const char *message, void *data);
```

A simple status handler function, that writes the status message to the standard errror.

Arguments:

message:

The status message.

data:

Additional context, with user-defined semantics. Existing igraph functions pass a null pointer here.

Returns:

Error code.

Time complexity: O(1).

28.3.3. Invoking the status handler

28.3.3.1. IGRAPH_STATUS — Report the status of an igraph function.

```
#define IGRAPH_STATUS(message,
```

Typically this function is called only a handful of times from an igraph function. E.g. if an algorithm has three major steps, then it is logical to call it three times, to signal the three major steps.

Arguments:

message:

The status message.

data:

Additional context, with user-defined semantics. Existing igraph functions pass a null pointer here.

Returns:

If the status handler returns with a value other than `IGRAPH_SUCCESS`, then the function that called this macro returns as well, with error code `IGRAPH_INTERRUPTED`.

28.3.3.2. `IGRAPH_STATUSF` — Report the status from an igraph function

```
#define IGRAPH_STATUSF(args)
```

This is the more flexible version of `IGRAPH_STATUS()`, having a printf-like syntax. As this macro takes variable number of arguments, they must be all supplied as a single argument, enclosed in parentheses. Then `igraph_statusf()` is called with the given arguments.

Arguments:

`args`:

The arguments to pass to `igraph_statusf()`.

Returns:

If the status handler returns with a value other than `IGRAPH_SUCCESS`, then the function that called this macro returns as well, with error code `IGRAPH_INTERRUPTED`.

28.3.3.3. `igraph_status` — Report status from an igraph function.

```
int igraph_status(const char *message, void *data);
```

It calls the installed status handler function, if there is one. Otherwise it does nothing. Note that the standard way to report the status from an igraph function is the `IGRAPH_STATUS` or `IGRAPH_STATUSF` macro, as these take care of the termination of the calling function if the status handler returns with `IGRAPH_INTERRUPTED`.

Arguments:

`message`:

The status message.

`data`:

Additional context, with user-defined semantics. Existing igraph functions pass a null pointer here.

Returns:

Error code. If a status handler function was called and it did not return with `IGRAPH_SUCCESS`, then `IGRAPH_INTERRUPTED` is returned by `igraph_status`().

Time complexity: O(1).

28.3.3.4. `igraph_statusf` — Report status, more flexible printf-like version.

```
int igraph_statusf(const char *message, void *data, ...);
```

This is the more flexible version of `igraph_status()`, that has a syntax similar to the `printf` standard C library function. It substitutes the values of the additional arguments into the *`message`* template string and calls `igraph_status()`.

Arguments:

`message`:

Status message template string, the syntax is the same as for the `printf` function.

`data`:

Additional context, with user-defined semantics. Existing igraph functions pass a null pointer here.

`...`:

The additional arguments to fill the template given in the *`message`* argument.

Returns:

Error code. If a status handler function was called and it did not return with `IGRAPH_SUCCESS`, then `IGRAPH_INTERRUPTED` is returned by `igraph_status`().

Chapter 29. Not Graph Related Functions

29.1. Igraph version number

29.1.1. `igraph_version` — Return the version of the igraph C library

```
int igraph_version(const char **version_string,
     int *major,
     int *minor,
     int *subminor);
```

Arguments:

version_string:

Pointer to a string pointer. If not null, it is set to the igraph version string, e.g. "0.6" or "0.5.3". This string should not be modified or deallocated.

major:

If not a null pointer, then it is set to the major igraph version. E.g. for version "0.5.3" this is 0.

minor:

If not a null pointer, then it is set to the minor igraph version. E.g. for version "0.5.3" this is 5.

subminor:

If not a null pointer, then it is set to the subminor igraph version. E.g. for version "0.5.3" this is 3.

Returns:

Error code.

Time complexity: O(1).

Example 29-1. File examples/simple/igraph_version.c

29.2. Running Mean of a Time Series

29.2.1. `igraph_running_mean` — Calculates the running mean of a vector.

```
int igraph_running_mean(const igraph_vector_t *data, igraph_vector_t *res,
   igraph_integer_t binwidth);
```

The running mean is defined by the mean of the previous *binwidth* values.

Arguments:

data:

The vector containing the data.

res:

The vector containing the result. This should be initialized before calling this function and will be resized.

binwidth:

Integer giving the width of the bin for the running mean calculation.

Returns:

Error code.

Time complexity: O(n), n is the length of the data vector.

29.3. Random Sampling from Very Long Sequences

29.3.1. `igraph_random_sample` — Generates an increasing random sequence of integers.

```
int igraph_random_sample(igraph_vector_t *res, igraph_real_t l, igraph_real_t h,
    igraph_integer_t length);
```

This function generates an increasing sequence of random integer numbers from a given interval. The algorithm is taken literally from (Vitter 1987). This method can be used for generating numbers from a *very* large interval. It is primarily created for randomly selecting some edges from the sometimes huge set of possible edges in a large graph.

Note that the type of the lower and the upper limit is `igraph_real_t`, not `igraph_integer_t`. This does not mean that you can pass fractional numbers there; these values must still be integral, but we need the longer range of `igraph_real_t` in several places in the library (for instance, when generating Erdos-Renyi graphs).

Arguments:

`res`:

Pointer to an initialized vector. This will hold the result. It will be resized to the proper size.

`l`:

The lower limit of the generation interval (inclusive). This must be less than or equal to the upper limit, and it must be integral. Passing a fractional number here results in undefined behaviour.

`h`:

The upper limit of the generation interval (inclusive). This must be greater than or equal to the lower limit, and it must be integral. Passing a fractional number here results in undefined behaviour.

`length`:

The number of random integers to generate.

Returns:

The error code `IGRAPH_EINVAL` is returned in each of the following cases: (1) The given lower limit is greater than the given upper limit, i.e. `l` > `h`. (2) Assuming that `l` < `h` and N is the sample size, the above error code is returned if N > |`h - l`|, i.e. the sample size exceeds the size of the candidate pool.

Time complexity: according to (Vitter 1987), the expected running time is O(length).

Reference:

`(Vitter 1987)`

J. S. Vitter. An efficient algorithm for sequential random sampling. *ACM Transactions on Mathematical Software,* 13(1):58--67, 1987.

Example 29-2. File examples/simple/igraph_random_sample.c

29.4. Convex hull of a set of points on a plane

29.4.1. `igraph_convex_hull` — Determines the convex hull of a given set of points in the 2D plane

```
int igraph_convex_hull(const igraph_matrix_t *data, igraph_vector_t *resverts,
         igraph_matrix_t *rescoords);
```

The convex hull is determined by the Graham scan algorithm. See the following reference for details:

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0262032937. Pages 949-955 of section 33.3: Finding the convex hull.

Arguments:

data:

vector containing the coordinates. The length of the vector must be even, since it contains X-Y coordinate pairs.

resverts:

the vector containing the result, e.g. the vector of vertex indices used as the corners of the convex hull. Supply `NULL` here if you are only interested in the coordinates of the convex hull corners.

rescoords:

the matrix containing the coordinates of the selected corner vertices. Supply `NULL` here if you are only interested in the vertex indices.

Returns:

Error code: `IGRAPH_ENOMEM`: not enough memory

Time complexity: O(n log(n)) where n is the number of vertices

Example 29-3. File examples/simple/igraph_convex_hull.c

29.5. Fitting power-law distributions to empirical data

29.5.1. `igraph_plfit_result_t` — Result of fitting a power-law distribution to a vector

```
typedef struct igraph_plfit_result_t {
  igraph_bool_t continuous;
  double alpha;
  double xmin;
  double L;
  double D;
  double p;
} igraph_plfit_result_t;
```

This data structure contains the result of `igraph_power_law_fit()`, which tries to fit a power-law distribution to a vector of numbers. The structure contains the following members:

Values:

`continuous`:

Whether the fitted power-law distribution was continuous or discrete.

`alpha`:

The exponent of the fitted power-law distribution.

`xmin`:

The minimum value from which the power-law distribution was fitted. In other words, only the values larger than `xmin` were used from the input vector.

`L`:

The log-likelihood of the fitted parameters; in other words, the probability of observing the input vector given the parameters.

`D`:

The test statistic of a Kolmogorov-Smirnov test that compares the fitted distribution with the input vector. Smaller scores denote better fit.

`p`:

The p-value of the Kolmogorov-Smirnov test. Small p-values (less than 0.05) indicate that the test rejected the hypothesis that the original data could have been drawn from the fitted power-law distribution.

29.5.2. `igraph_power_law_fit` — Fits a power-law distribution to a vector of numbers

```
int igraph_power_law_fit(const igraph_vector_t* data, igraph_plfit_result_t* result,
 igraph_real_t xmin, igraph_bool_t force_continuous);
```

This function fits a power-law distribution to a vector containing samples from a distribution (that is assumed to follow a power-law of course). In a power-law distribution, it is generally assumed that P(X=x) is proportional to x^{-alpha}, where x is a positive number and alpha is greater than 1. In many real-world cases, the power-law behaviour kicks in only above a threshold value *xmin*. The goal of this functions is to determine *alpha* if *xmin* is given, or to determine *xmin* and the corresponding value of *alpha*.

The function uses the maximum likelihood principle to determine *alpha* for a given *xmin;* in other words, the function will return the *alpha* value for which the probability of drawing the given sample is the highest. When *xmin* is not given in advance, the algorithm will attempt to find the optimal *xmin* value for which the p-value of a Kolmogorov-Smirnov test between the fitted distribution and the original sample is the largest. The function uses the method of Clauset, Shalizi and Newman to calculate the parameters of the fitted distribution. See the following reference for details:

Aaron Clauset, Cosma R .Shalizi and Mark E.J. Newman: Power-law distributions in empirical data. SIAM Review 51(4):661-703, 2009.

Arguments:

`data`:

vector containing the samples for which a power-law distribution is to be fitted. Note that you have to provide the *samples,* not the probability density function or the cumulative distribution function. For example, if you wish to fit a power-law to the degrees of a graph, you can use the output of `igraph_degree` directly as an input argument to `igraph_power_law_fit`

`result`:

the result of the fitting algorithm. See `igraph_plfit_result_t` for more details.

`xmin`:

the minimum value in the sample vector where the power-law behaviour is expected to kick in. Samples smaller than `xmin` will be ignored by the algoritm. Pass zero here if you want to include all the samples. If `xmin` is negative, the algorithm will attempt to determine its best value automatically.

`force_continuous`:

assume that the samples in the `data` argument come from a continuous distribution even if the sample vector contains integer values only (by chance). If this argument is false, igraph will assume a continuous distribution if at least one sample is non-integer and assume a discrete distribution otherwise.

Returns:

Error code: `IGRAPH_ENOMEM`: not enough memory `IGRAPH_EINVAL`: one of the arguments is invalid `IGRAPH_EOVERFLOW`: overflow during the fitting process `IGRAPH_EUNDERFLOW`: underflow during the fitting process `IGRAPH_FAILURE`: the underlying algorithm signaled a failure without returning a more specific error code

Time complexity: in the continuous case, O(n log(n)) if `xmin` is given. In the discrete case, the time complexity is dominated by the complexity of the underlying L-BFGS algorithm that is used to optimize alpha. If `xmin` is not given, the time complexity is multiplied by the number of unique samples in the input vector (although it should be faster in practice).

Example 29-4. File examples/simple/igraph_power_law_fit.c

Chapter 30. Licenses for igraph and this manual

30.1. THE GNU GENERAL PUBLIC LICENSE

30.1.1. Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

30.1.2. GNU GENERAL PUBLIC LICENSE

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third

parties under the terms of this License.

c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO

OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

30.1.3. How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program’s name and a brief idea of what it does.>
Copyright (C) <year>  <name of author>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program
‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

30.2. The GNU Free Documentation License

Version 1.2, November 2002

30.2.1. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

30.2.2. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

30.2.3. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

30.2.4. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

30.2.5. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of

Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

30.2.6. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

30.2.7. 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

30.2.8. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

30.2.9. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

30.2.10. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

30.2.11. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

30.2.12. G.1.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (c)  YEAR  YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU
Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index