CGraph documentation

Bruno Kim Medeiros Cesar

July 10, 2013

Abstract

Contents

1 `sorting` 4

2 `stat` 5

3 `list` 6

4 `set` 7

5 `graph` 8

5.1 Allocation and deallocation 8

5.1.1 Creation 8

5.1.2 Deallocation 9

5.2 Conversion 9

5.2.1 Threshold 9

5.2.2 Simmetry 9

5.2.3 Remove self loops 10

5.2.4 Coalesce 10

5.2.5 Split edges 10

5.3 Input/Output 10

5.4 Insertion 10

5.5 Retrieval 10

5.6 Removal 10

5.7 Query 10

5.8 Adjacencies 10

5.9 Copying 10

6 `graph_metric` 11

6.1 Constants 11

6.1.1 `GRAPH_METRIC_TOLERANCE` 11

6.1.2 `GRAPH_METRIC_MAX_ITERATIONS` 11

6.2 Component identification and extraction 11

6.2.1 `graph_undirected_components` 11

6.2.2 `graph_directed_components` 11

6.2.3 `graph_num_components` 12

6.2.4 graph_components . . . 12
6.2.5 graph_components . . . 12
6.3 Degree metrics . . . 12
6.3.1 graph_degree . . . 12
6.3.2 graph_directed_degree . . . 13
6.4 Clustering metrics . . . 13
6.4.1 graph_clustering . . . 13
6.4.2 graph_num_triplets . . . 13
6.4.3 graph_transitivity . . . 13
6.5 Geodesic distance metrics . . . 14
6.5.1 Definitions . . . 14
6.5.2 graph_geodesic_distance . . . 14
6.5.3 graph_geodesic_vertex . . . 14
6.5.4 graph_geodesic_all . . . 14
6.5.5 graph_geodesic_distribution . . . 14
6.6 Centrality measures . . . 14
6.6.1 graph_betweenness . . . 14
6.6.2 graph_eigenvector . . . 14
6.6.3 graph_pagerank . . . 14
6.6.4 graph_kcore . . . 14
6.7 Correlation measures . . . 14
6.7.1 graph_degree_matrix . . . 14
6.7.2 graph_neighbor_degree_vertex . . . 14
6.7.3 graph_neighbor_degree_all . . . 14
6.7.4 graph_knn . . . 14
6.7.5 graph_assortativity . . . 14

7 graph_layout **15**
7.1 Types . . . 15
7.1.1 coord_t . . . 15
7.1.2 box_t . . . 15
7.1.3 color_t . . . 15
7.1.4 circle_style_t . . . 15
7.1.5 path_style_t . . . 15
7.2 Layout . . . 16
7.2.1 graph_layout_random . . . 16
7.2.2 graph_layout_random_wout_overlap . . . 16
7.2.3 graph_layout_circle . . . 16
7.2.4 graph_layout_circle_edges . . . 16
7.2.5 graph_layout_degree . . . 17
7.3 Printing . . . 17
7.3.1 graph_print_svg . . . 17
7.3.2 graph_print_svg_one_style . . . 17
7.3.3 graph_print_svg_some_styles . . . 18

8 graph_model **19**
8.1 Graph creation . . . 19
8.1.1 new_clique . . . 19
8.1.2 new_erdos_renyi . . . 19
8.1.3 new_watts_strogatz . . . 19

8.1.4 new_barabasi_albert 20

9 graph_propagation **21**

9.1 Types 21

9.1.1 message_t 21

9.1.2 propagation_step_t 21

9.1.3 state_transition_f 21

9.1.4 is_propagation_end 22

9.2 Functions 22

9.2.1 graph_count_state 22

9.2.2 graph_propagation 22

9.2.3 delete_propagation_steps 22

9.2.4 graph_animate_coefficient 22

9.2.5 graph_propagation_freq 23

9.3 Models 23

9.3.1 SI 23

9.3.2 SIS 23

9.3.3 SIR 23

9.3.4 SEIR 23

9.3.5 Daley-Kendall 23

1 sorting

2 `stat`

3 `list`

4 `set`

5 graph

The `graph` module has the basic algorithms and data structures to manipulate graphs. All modules starting with `graph_*` depends on this module.

The basic data structure is `graph_t`, which uses adjacencies lists to store edges. All storage is done in-memory, which means the graph size is limited to what will fit with your RAM requirements. Also, as everything is indexed by native `int`s, storage may increase whether it is stored in a 32-bit or 64-bit system. Also, in 32-bit systems the maximum number of edges and vertices is $2^{31} \approx 2.2 \times 10^9$.

Several types of graphs are supported, although more complex types require more memory. The basic traits which can be combined are listed below. Simple graphs (undirected, unweighted and unlooped, true graph) need approximately $4N + 16M$ bytes in 32-bit and $8N + 32M$ bytes in 64-bit systems.

Directed Directed graphs consider edges to be ordered sets, i.e., where order matters. An edge between vertices V_1 and V_2 is different from an edge between vertices V_2 and V_1. Directed graphs need approximately $4N + 24M$ bytes in 32-bit and $8N + 48M$ bytes in 64-bit systems. Creation flag is `GRAPH_DIRECTION`.

Weighted Weighted graphs attach weights to edges, stored in `float`s (standardized to be 32-bits). Weighted graphs need additionaly $4M$ bytes of storage. Creation flag is `GRAPH_WEIGHT`.

Pseudo Pseudo graphs allow multiple edges between the same vertices, which can be directed or weighted. Pseudo graphs don't need additional storage. Creation flag is `GRAPH_PSEUDO`.

Looped Looped graphs allow self-loops, or edges from a vertex to itself. Looped graphs need additional $4N$ bytes in 32-bit and $8N$ bytes in 64-bit systems. Creation flag is `GRAPH_LOOP`.

Multi Multigraphs allow that edges store multiple vertices. This is useful to represent bipartite graphs, where edges may represent one set of elements. The storage of this kind of graph fluctuates according to how many vertices each edge stores. Creation flag is `GRAPH_MULTI`.

Graphs can be converted between types, from a more complex to a simpler. This will be discussed in subsection 5.2.

5.1 Allocation and deallocation

5.1.1 Creation

```
graph_t *new_simple_graph();
graph_t *new_graph(unsigned int flags);
```

Graphs are created using one of the functions above. `new_simple_graph` creates a simple graph, which is equivalent to `new_graph(0)`. To create more complex graphs its needed to pass flags bitwise OR'ed together. For example, to create a directed, looped, weighted graph you should call `new_graph(GRAPH_DIRECTION | GRAPH_LOOP | GRAPH_WEIGHT)`.

If there isn't enough memory, or if illegal flags are passed, both function return `NULL`.

5.1.2 Deallocation

```
void *delete_graph(graph_t *graph);
```

5.2 Conversion

These functions convert from a type of graph to a simpler one. For implementation safety, all functions return a new graph, thus possibly creating duplicates in memory.

5.2.1 Threshold

```
graph_t *graph_threshold
 (const graph_t *original, double threshold,
  bool keep_weights);
```

Removes every edge with weight smaller than the specified threshold. If `keep_weights` is false, the resulting graph is unweighted. Otherwise, the new graph keeps all weights equal to or bigger than the threshold from the input graph.

To keep all edges removing weights, use threshold $= -\infty$, i.e.,

graph_threshold(original_graph, −1.0/0.0, false).

If original is not weighted, or if memory was exhausted, the function returns `NULL`.

5.2.2 Simmetry

```
graph_t *graph_dual(const graph_t *original);
graph_t *graph_simmetry
  (const graph_t *original, bool keep_directed);
graph_t *graph_direct
  (graph_t *original, bool split_weights);
```

The dual graph of a directed graph is the graph with all its edges reversed. The symmetric graph is the union of a graph with its dual, thus converting a directed graph into an undirected one.

graph_dual receives a directed graph and returns its directed dual.

graph_simmetry receives a directed graph and returns its dual. If keep_directed is true, the resulting graph is directed, and dual edges keep their weights; otherwise, the resulting graph is undirected, and the weights of dual edges are summed together.

graph_direct receives an undirected graph and returns its directed equivalent, where an edge E_{ij} is split into $E_{i\to j}$ and $E_{j\to i}$. If split_weights is true, its weight is split evenly between the new edges; otherwise, both edges receive the same weight.

If original isn't of the specified type, or if memory was exhausted, the function returns `NULL`.

5.2.3 Remove self loops

```
graph_t *graph_remove_self_loops(const graph_t *original);
```

Remove self loops from the original unlooped graph, returning an unlooped graph.

If original isn't unlooped, or if memory was exhausted, the function returns `NULL`.

5.2.4 Coalesce

```
graph_t *graph_coalesce(const graph_t *original);
```

Coalesce multiple edges in a pseudo-graph in a single one, returning a weighted true graph (i.e., graph_is_pseudo() returns false). If original is unweighted, the weight of the edge E_{ij} is the number of edges between V_i and V_j in original. If original is weighted, the weight of the edge E_{ij} is the sum of weights between V_i and V_j in original.

If original isn't a pseudo-graph, or if memory was exhausted, the function returns `NULL`.

5.2.5 Split edges

```
graph_t *graph_split_edges
  (const graph_t *original, bool split_weights);
```

Split multiedges in a multigraph into separate single edges, returning a regular pseudo-graph (i.e., graph_is_multi() returns false). Each edge $E_i = (V_{i1}, \ldots, V_{il}$ is split into simple edges such that every vertex V_{ik} has an edge to vertices $V_{i,k+1}, \ldots, V_{il}$. This means that splitting does not include a self-loop if the graph is looped, and that if it is directed a vertex does not have edges to its predecessor in a multiedge.

If split_weights is true, a multiedge weight is splitted evenly among all its resulting edges; otherwise all resulting edges receive the original weight.

If original is not a multigraph, or if memory was exhausted, the function returns `NULL`.

5.3 Input/Output

5.4 Insertion

5.5 Retrieval

5.6 Removal

5.7 Query

5.8 Adjacencies

5.9 Copying

6 `graph_metric`

6.1 Constants

These constants are hard-coded to protect some numeric processes of hanging. They can be redefined during compilation, passing a flag such as
`-DGRAPH_METRIC_TOLERANCE=1E-3`.

6.1.1 `GRAPH_METRIC_TOLERANCE`

Error tolerance for numeric methods.

6.1.2 `GRAPH_METRIC_MAX_ITERATIONS`

Maximum number of iterations for numeric methods.

6.2 Component identification and extraction

6.2.1 `graph_undirected_components`

Label vertices' components treating edges as undirected.

Preconditions `label` must have dimension n.

Postconditions `label[i]` is the component ID of vertex v_i.

Return Number of components

For directed graphs, considers adjacencies as incidences. Labels start from 0 and are sequential with step 1. Component IDs are not ordered according to size.

6.2.2 `graph_directed_components`

Label vertices' components treating edges as directed. NOT IMPLEMENTED YET.

Preconditions `label` must have dimension n.

Postconditions `label[i]` is the component ID of vertex v_i.

Return Number of components

For undirected graphs, simply call `graph_undirected_components`. For directed graphs, two vertices v_i and v_j are in the same component if and only if

$$d(v_i, v_j) \neq \infty$$
$$d(v_j, v_i) \neq \infty$$

where $d(u, v)$ is the geodesic distance between them. In other words, they are in the same component if they are mutually reachable.

Labels start from 0 and are sequential with step 1. Component IDs are not ordered according to size.

6.2.3 `graph_num_components`

Extract number of components from label vector.

Preconditions
  $n > 0$
  `label` must have dimension n.
  `label` must contain sequential IDs starting from 0.

Return Number of components

6.2.4 `graph_components`

Map components to vertices from label vector.

Preconditions
  $n > 0$
  `label` must have dimension n.
  `label` must contain sequential IDs starting from 0.
  `comp` must have size `num_comp` and all sets should be already initialized.
  `graph_num_components(g) == num_comp`

Postconditions
  If v_i is in component c_j, then
  `label[i] == j` and
  `set_contains(comp[j], i)` is true.

Return Number of components

6.2.5 `graph_components`

Creates a new graph from `g`'s largest component.

The guarantee of vertices' order ID is the same as `graph_subset`. If two or more components have the same maximum size, one will be chosen in an undefined way.

Return A new graph isomorphic to `g`'s largest component.

Memory deallocation

```
graph_t *largest = graph_components(g);
delete_graph(largest);
```

6.3 Degree metrics

6.3.1 `graph_degree`

List all vertices' degrees.

Preconditions `degree` must have dimension n.

Postconditions `degree[i]` is the degree of vertex v_i.

The degree of a directed graph's vertex is defined as the sum of incoming and outgoing edges.

6.3.2 `graph_directed_degree`

List all vertices' incoming and outgoing degrees.

Preconditions
: `g` must be directed. `in_degree` must have dimension n. `out_degree` must have dimension n.

Postconditions
: `in_degree[i]` is the number of incoming edges to vertex v_i. `out_degree[i]` is the number of outgoing edges from vertex v_i.

6.4 Clustering metrics

6.4.1 `graph_clustering`

List all vertices' local clustering.

Preconditions
: `g` must be undirected.
`clustering` must have dimension n.

Postconditions `clustering[i]` is the local clustering coefficient of vertex v_i.

The local clustering coefficient is only defined for undirected graphs, and gives the ratio of edges between a vertex' neighbors and all possible edges.

Formally,

$$C_i = \frac{e_i}{\binom{k_i}{2}} = \frac{2e_i}{k_i(k_i - 1)}$$

where

C_i is the local clustering coefficient of vertex v_i.

e_i is the number of edges between v_i's neighbors.

k_i is the degree of v_i.

If a vertex v_i has 0 or 1 adjacents, $C_i = 0$ by definition.

6.4.2 `graph_num_triplets`

Counts number of triplets and triangles (6 * number of closed triplets).

6.4.3 `graph_transitivity`

Compute the ratio between number of triangles and number of triplets.

6.5 Geodesic distance metrics

6.5.1 Definitions

6.5.2 graph_geodesic_distance

6.5.3 graph_geodesic_vertex

6.5.4 graph_geodesic_all

6.5.5 graph_geodesic_distribution

6.6 Centrality measures

6.6.1 graph_betweenness

6.6.2 graph_eigenvector

6.6.3 graph_pagerank

6.6.4 graph_kcore

6.7 Correlation measures

6.7.1 graph_degree_matrix

6.7.2 graph_neighbor_degree_vertex

6.7.3 graph_neighbor_degree_all

6.7.4 graph_knn

6.7.5 graph_assortativity

7 graph_layout

7.1 Types

7.1.1 coord_t

Euclidean coordinates in 2D.

7.1.2 box_t

Box (rectangle) definition in 2D, given by its SW and NE vertices in a positively oriented world frame, such as the screen. Images may have a negatively oriented frame, with y pointing down. It is necessary that `box.sw.y < box.ne.y` and `box.sw.x < box.ne.x`.

7.1.3 color_t

Array with 4 colors between 0 and 255, inclusive: red (R), green (G), blue (B) and alpha (A). $A = 0$ means totally transparent, and $A = 255$ means totally opaque.

7.1.4 circle_style_t

SVG circle style.

`radius` Circle radius in pixels.

`width` Stroke width in pixels. This is added to the radius for total size.

`fill` Color of the fill.

`stroke` Color of the stroke.

7.1.5 path_style_t

SVG path style.

`type` Path type.

`from, to` Path origin and destination.

`control` Control point

`width` Stroke width in pixels.

`color` Stroke color.

For `style.type == GRAPH_STRAIGHT`, draws a straight line from origin to destination.

For `style.type == GRAPH_PARABOLA`, draws a parabola from origin to destination using the control point.

For `style.type == GRAPH_CIRCULAR`, draws the arc of a circle from origin to destination using the control point as the circle center.

7.2 Layout

7.2.1 `graph_layout_random`

Place points uniformly inside specified box.

Preconditions

`box` must be a valid box.
`p` must have dimension n.

Postconditions `p[i]` is a random coordinate inside `box`.

7.2.2 `graph_layout_random_wout_overlap`

Place points with specified radius uniformly avoiding overlap with probability t.

Preconditions

`radius` must be positive.
t must be a valid probability $(0 \geq t \geq 1)$.
`p` must have dimension n.

Postconditions `p[i]` is a random coordinate.

The algorithm determines a box with size l such that, if n points with radius r are thrown within it, will not have any collision with probability t. The formula is derived in Math Exchange.

$$l = \frac{nr}{2}\sqrt{\frac{2\pi}{-\log(1-t)}}$$

7.2.3 `graph_layout_circle`

Place points with specified radius in a circle without overlap.

Preconditions

`radius` must be positive.
`p` must have dimension n.

Postconditions `p[i]` is a coordinate in a circle.

Return value Circle bounding box size.

Points are positioned sequentially in a circle, starting from the rightmost and following in counterclockwise order.

7.2.4 `graph_layout_circle_edges`

Fill edge style for a circular layout.

Preconditions

`size` must be the circle bounding box size.
`width` must be positive.
`color` must be a valid color.
`es` must have dimension m.
`edge_style` must have dimension 2.

Postconditions
: `es[i]` is one of the styles `CIRCULAR` or `PARABOLA`.
`edge_style[0]` is the `CIRCULAR` style.
`edge_style[1]` is the `PARABOLA` style.

This function maps `es` to a circular or parabolic style, where an edge is circular if its endpoints are adjacent in a circle, and parabolic otherwise.

7.2.5 `graph_layout_degree`

Place points in concentric shells, with highest degrees near the center.

Preconditions
: `radius` must be positive.
`p` must have dimension n.

Postconditions `p[i]` is a coordinate.

Each shell is attached to a degree value; the inner shell contains elements of the highest degree, and the outer shell contains elements with the lowest degree. In each shell, elements are placed equally apart.

7.3 Printing

Printing functions accept optional `width` and `height` parameters in pixels. They won't be considered if they are negative or zero.

7.3.1 `graph_print_svg`

Prints graph as SVG to file, using vertex coordinates given in p and with a style for each point and edge.

Preconditions
: `p` must have dimension n. `point_style` must have dimension n. `edge_style` must have dimension m.

Postconditions `filename` is a valid SVG file.

Edges are ordered according to vertices' order. In undirected graphs, an edge E_{ij} is considered only if $i < j$. In directed graphs, mutual edges will superimpose if `edge_style.type == GRAPH_STRAIGHT`.

7.3.2 `graph_print_svg_one_style`

Prints graph as SVG to file, using vertex coordinates given in p and with a single style for all points and edges.

Preconditions
: `p` must have dimension n.

Postconditions `filename` is a valid SVG file.

The edge style type is ignored, using only `GRAPH_STRAIGHT`.

7.3.3 `graph_print_svg_some_styles`

Prints graph as SVG to file, using vertex coordinates given in `p` and with a number of styles given. The mapping vertex→style is given in `ps`, and the mapping edge→style is given in `es`.

Preconditions

`p` must have dimension n.
`ps` must have dimension n.
`es` must have dimension m.
`point_style` must have dimension `num_point_style`.
`edge_style` must have dimension `num_edge_style`.

Postconditions `filename` is a valid SVG file.

This function tries to avoid extensive memory utilization one just some styles are desired. If vertex v_i should have style S_j, then `ps[i] = j`. Ditto for edges.

Edge order is based on vertices order. In undirected edges, edge E_{ij} is considered only if $i < j$.

8 graph_model

8.1 Graph creation

These functions creates new graphs, whose memory should be managed by the caller.

The reentrant versions `new_erdos_renyi_r`, `new_watts_strogatz_r` and `new_barabasi_albert_r` accept a state argument that will be used to call `rand_r` for pseudo-random number generation. Two calls with the same state argument yield the same graph and same final state, allowing reproducibility.

8.1.1 `new_clique`

Creates a complete network with n vertices.

Preconditions $n > 0$

Return value An undirected, unweighted complete graph, or `NULL` in case of memory exhaustion.

It should be noticed that the data structure is inefficient to represent large dense graphs, so it is recommended to check for memory exhaustion upon return.

8.1.2 `new_erdos_renyi`

Creates a random network with n vertices and average degree k.

Preconditions
$n > 0$
$0 < k < n$

Return value An undirected, unweighted random graph.

There is no guarantee that the network will be connected. The size and characteristic of the largest component follow different regimes depending on k:

Regime	Size	Loop
$k < 1$	$\log n$	No loop
$k = 1$	$n^{2/3}$	No loop
$k > 1$	αn	Some loops
$k > \log n$	n	Many loops

8.1.3 `new_watts_strogatz`

Creates a small-world network with n vertices and average degree k, with rewiring probability β.

Preconditions
$n > 0$
k is even
$0 < k < n$
β is a valid probability ($0 <= \beta <= 1$)

Return value An undirected, unweighted small-world graph.

8.1.4 `new_barabasi_albert`

Creates a scale-free network with n vertices and average degree k.

Preconditions

$n > 0$
$0 < k < n$

Return value An undirected, unweighted scale-free graph.

9 graph_propagation

Information dissemination simulation in networks are implemented in CGraph in a more abstract way, as there is lots in common between different propagation models.

Propagation models consists in a state diagram that represent the transition sequence for each individual, where one of them is the *infectious state*. At each time step, an infectious individual sends a message to one of its adjacents, chosen from an uniform distribution. Care should be taken to determine the next state if an individual receives more than one message per time step.

Models are implemented using two callbacks that are called in each time step:

`state_transition_f` determine the next state vector (ie, in which state each individual is in);

and `is_propagation_end` determines if the propagation has ended.

Some models may never reach an end, so there's an additional condition that each simulation will run for at most $K \log_2 n$ iterations, where K is defined in `GRAPH_PROPAGATION_K`. It can be redefined during compilation with

`-DGRAPH_PROPAGATION_K=10`

9.1 Types

9.1.1 message_t

Message type storing the origin `orig` and destination `dest` of a message.

9.1.2 propagation_step_t

Structure storing information on a propagation time step: its state vector and the messages exchanged.

`n` Number of individuals in this time step.

`state` State vector, where `state[i]` is the state of individual i.

`num_message` Number of messages exchanged, that must be equal to the number of individuals in the infectious state.

`message` Message array, storing the origin and destination of messages.

9.1.3 state_transition_f

Callback for state transition, implemented by the propagation model.

Preconditions

`next` must have dimension n. `curr` must be information about the current step, including exchanged messages.
`n` is the number of elements, that in a dynamic network may be different than the one in the current time step.
`params` is a pointer to model specific parameters.
`seedp` is a pointer to a PRNG state variable, or `NULL`.

Postcondition `next[i]` is the next state of the element i.

9.1.4 `is_propagation_end`

Callback for simulation termination, implemented by the propagation model.

`state` is the state vector, and `num_step` is the current iteration number. `params` is a pointer to model specific parameters.

9.2 Functions

9.2.1 `graph_count_state`

Counts number of individuals in s that are in the given state.

9.2.2 `graph_propagation`

Simulates a propagation in graph with a given initial state vector using the given propagation model.

Preconditions

`init_state` is a valid state vector with dimension n.
`model` is a valid propagation model.
`params` is a pointer to the model specific parameter structure.

Postcondition

`num_step` is the number of steps in simulation.

Return value

Array of `propagation_step_t`.

Memory deallocation

```
int num_step;
propagation_step_t *step = graph_propagation(..., &num_step, ...);
delete_propagation_steps(step, num_step);
```

There is a reentrant version `graph_propagation_r`, that expects a pointer to the PRNG state variable, allowing reproducible simulations.

9.2.3 `delete_propagation_steps`

Deallocate a `propagation_step_t` array that was allocated with `graph_propagation`.

9.2.4 `graph_animate_coefficient`

Creates animation frames of a propagation in the given graph.

Preconditions

`folder` is an existing folder.
`p` is a coordinate array with dimension n.
`num_state` is the number of states in the propagation model used. `step` is a propagation step array with dimension `num_step`.

Postcondition

The given folder has `num_step` SVG files with name format `frame%05d`, numbered incrementally from 0.

9.2.5 `graph_propagation_freq`

Compute the number of individuals in each state at each propagation step.

Preconditions

`step` is an array with dimension `num_step`.
`freq` is an allocated matrix with dimensions `num_step` × `num_state`
`num_state` is the number of states in the propagation model used.

Postcondition

`freq[i][s]` is the number of individuals in state s at iteration i.

9.3 Models

9.3.1 SI

9.3.2 SIS

9.3.3 SIR

9.3.4 SEIR

9.3.5 Daley-Kendall