CGraph documentation

Bruno Kim Medeiros Cesar

July 27, 2013

Abstract

Contents

1 sorting

2 `stat`

3 `list`

4 set

This module provides a structure able to efficiently include, remove and query integers in a hash table. Its possible to iterate over all elements via a linked list.

This data structure automatically grows to store more integers efficiently, but will not shrink if items are removed, wasting memory. Consider using set_copy to copy the structure using less memory.

4.1 Constants

The following constants can be redefined only during compilation.

Constant	Value	Description
SET_UTILIZATION_RATE	0.75	Maximum utilization rate of hash table.

4.2 Types

```
typedef struct set_t set_t;
typedef struct set_entry_t set_entry_t;
struct set_entry_t {
  int key;
  set_entry_t *next;
}
```

A set is an object of the type set_t, which is basically an array of set_entry_t. An entry contains a key and a pointer to a next element, allowing to traverse all entries as a linked list terminated by a NULL. The head can be fetched with set_head(set_t *set).

4.3 Allocation and deallocation

```
set_t *new_set(int minimum);
void  delete_set(set_t *set);
```

A set is created via new_set, where minimum is the expected number of elements to be inserted. It preallocates a table large enough to contain this number of elements, accounting for utilization rate. This may be interesting to avoid multiple memory allocations if the table needs to grow. delete_set deallocates memory requested for the data structure.

4.4 Insertion and retrieval

```
error_t set_put(set_t *set, int v);

bool set_contains(const set_t *set, int v);
int set_get(const set_t *set, int pos);
int set_index(const set_t *set, int v);
```

set_put(set,v) inserts a value v in set, with $\mathcal{O}(1)$ amortized cost. If the utilization rate goes beyond SET_UTILIZATION_RATE, a table roughly twice bigger is allocated and populated with $\mathcal{O}(n)$ operations. If there is no memory available, the function returns ERROR_NO_MEMORY; otherwise, returns ERROR_SUCCESS.

set_contains checks whether a value is in the given set, with $\mathcal{O}(1)$ amortized cost. set_get returns the value in position pos in the linked list, and set_index returns the position of a value v in the list, or -1 if there is no such value in the set. Both operations have average cost $\mathcal{O}(n/2)$.

Prerequisites: pos should be between 0 and n for set_get.

4.5 Random retrieval

```
int set_get_random(const set_t *set);
int set_get_random_r(const set_t *set, unsigned int *seedp);
```

Both functions pick an element from the table with uniform probability. The reentrant version set_get_random_r accepts a pointer to a seed that will be passed to rand_r. The non-thread-safe version is equivalent to set_get_random_r(set, NULL).

The implementation uses two strategies to pick a random element: if the table is almost empty (with $n < 4\sqrt{s}$, where s is the size of the table), it selects a number i uniformly from $[0, n)$ and picks the i-th element at the linked list. Otherwise, it selects slots at random until it finds a non-empty slot, and returns it. This approach takes s/n selections in average. The rationale for this algorithm is explained at annex 11.1.

4.6 Removing

```
bool set_remove(set_t *set, int v);
void set_clean(set_t *set);
```

set_remove removes a given element from the set. If the element is present, the function returns true and the element is removed with $\mathcal{O}(n/2)$ operations in average. Otherwise, the function returns false with $\mathcal{O}(1)$ operations.

set_clean cleans all slots, without freeing any memory.

The table is not shrinked if the utilization rate is low. If it's necessary to free memory, its recommended to copy the set with set_copy and delete the current structure.

4.7 Set operations

```
error_t set_union(set_t *dest, const set_t *other);
void set_difference(set_t *dest, const set_t *other);
void set_intersection(set_t *dest, const set_t *other);
```

set_union, set_difference and set_intersection compute, respectively, the union, intersection and difference between the two arguments, and store the result mutating the first one.

If there is no memory available for set_union, it returns ERROR_NO_MEMORY; otherwise, it returns ERROR_SUCCESS. Computing the difference and intersection does not need additional memory, so both have type void.

4.8 Querying

```
int set_size(const set_t *set);
int set_table_size(const set_t *set);
set_entry_t *set_head(const set_t *set);
```

set_size returns the number of elements inserted into the set, and set_table_size returns the size of the table used.

set_head returns the first element in the linked list of entries. If set_optimize were never called, the elements are presented in insertion order. This shouldn't be used to change the content of an entry, which would invalidate set invariants.

4.9 Structure optimization

```
void set_optimize(set_t *set);
```

As elements are inserted into a hash table, the linked list can get very tangled jumping from far away points in memory. set_optimize try to improve memory locality by rebuilding the linked list in sequential order, thus reducing the amount of cache misses when traversing it.

4.10 Copying

```
set_t *set_copy(const set_t *set);
void set_to_array(const set_t *set, int *arr);
int* set_to_dynamic_array(const set_t *set, int *n);
```

set_copy creates a new deep copy of the input set, using as much memory as strictly needed. Keys are inserted in the same order as the original set. If there isn't enough memory, the function returns NULL.

set_to_array populates an array with the keys in the set. The array must already be allocated.

set_to_dynamic_array allocates an array and populates it with the keys in the set. The n parameter is optional, holding the size of the array. If there isn't enough memory, n receives 0 and the function returns NULL.

4.11 Printing

```
void set_print(const set_t *set);
void set_fprint(FILE *stream, const set_t *set);
```

Prints a set to the indicated stream, or the standard stream in set_print.

5 `graph`

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

10 graph_game

11 Annex

11.1 Picking an element at random in a hash table

Let n be the number of elements in the table, and s the size of the table. The utilization rate is given as $r = n/s$. The probability to choose a non-empty slot uniformly at the first try is r; at the second try is $r(1-r)$; at the third try is $r(1-r)^2$ and so on. The expected number of tries k is

$$\begin{aligned} k &= 1 \cdot r + 2 \cdot r(1-r) + 3 \cdot r(1-r)^2 + \ldots \\ &= r \sum_{n=1}^{\infty} n(1-r)^{n-1} \\ &= r(1/r^2) = 1/r \\ &= s/n \end{aligned}$$

If we pick a random number between 0 and $n-1$ and walk through the linked list, we expect to step through $n/2$ elements at average. Let a be the time per random try, and b the time of a step. We should use random picking if

$$\begin{aligned} a \cdot s/n &< b \cdot n/2 \\ (2a/b)s &< n^2 \\ n &> \sqrt{(2a/b)s} \end{aligned}$$

We expect that $a > b$, and use $a/b = 8$ to derive the rule that $n > 4\sqrt{s}$ to switch between one approach and the other.