Topologic

1.0.6

Generated by Doxygen 1.9.1

Chapter 1

Topologic

A library to simulate DFAs and Probabilistic DFAs using context switching. This library provides a way to build and define the behavior of a graph. The client is able to define the function each vertex and edge of the graph make and how the graph transitions from one (the first valid edge) or to many states, as well as which states the graph starts in.

The library provides a way for the client to define what information is saved localy to each vertex and edge, as well as which information the vertex and edges originating at that edge share. In addition the client can pass information to each initial state that will be passed and modifyable by vertices at each state transition.

This library can be used to create AI, ML, and simulations (such as of stock options, electron-electron interaction, etc.).

1.1 Installation

1.1.1 Arch Linux

>yay -S topologic-git

-To use in program use

>#include <topologic/topologic.h>

-Link with

>gcc ... -ltopologic

1.1.2 Manualy

Build C Library >make

-creates libtopologic.a

Build C++ Library >make cpp

-creates libtopologic.a

Build Python 3 Library >make python

-creates topylogic/topylogic.so

Build Python 2 Library >make python2

-creates topylogic/topylogic.so

Clean >make clean

Debug >make CFLAGS=-DDEBUG

1.2 TODO

-C# Wrap

-Rust Wrap

-Python (2/3) Wrap

1.3 Errata

1.3.1 Edge/vertex modification

Modifying/deleting vertices and edges inside f() not using submit request can lead to undefined behavior or dead locks. This is because in CONTEXT set to SWITCH or NONE with many starting vertices, can lead to a structure being NULL'd with its lock destroyed while another thread is holding that lock or two threads trying to modify each other and thus resulting in a dead lock. It is up to the client's discretion to modify directly when in SWITCH or NONE with many starting vertices. Although in SINGLE, this should be fine. To mitigate this problem, the client should use submit request which will handle the requests sequentially with destroying structures last.

1.3.2 Graph modification

Trying to delete the graph while running will result in undefined behav- ior. Destroying the graph does not lock any thread and thus will cause race conditions. The graph should be deleted only once all threads reach a sink.

1.3.3 Parameter Passing

Parameters passed to edges or vertices functions will be free'd immediately after use and therefore will cause an error should they try and be accessed. To mitigate the values may be stored in the edge's or vertex's global or shared variables. Any non standard data type, such as struct, is dependent on the user to free its content as the library will only free the pointer to the struct and the pointer to the array of variables.

1.3.4 Illegal Arguments

Passing wrong values or wrong number of variables to any function will result in failure. A client should be aware of which vertices connect and what edges it has and the proper handling required between such connections. Should the client choose to dynamically add/remove vertices or edges or even modify while running should be aware of the changes that may occur in the graph and the resulting change in dependence on proper variable handling.

1.3.5 Graph Modification (cont.)

It is possible to modify the graph while it is running. To do so the client should submit request to add a change or pause the graph. Making any changes externally may result in undefined behavior if done improperly. Modifying the graph directly while running could result in failure. Should the number of init vertex args not much the number of vertices in the start set the program will result in an error.

1.3.6 Data Structure Errata

Stacks and AVLTree can take non malloc’d data and function normally within scope of those non malloc’d data. However, in another scope the memory will be unaddressable and thus should be malloc’d memory instead. The stack and AVL Tree wll not free the void $*$ data since the void $*$ data structure is unknown to them and thus the client should free the memory.

1.3.7 SWITCH_UNSAFE

Should SWITCH_USNAFE be used as the context, then the graph will run like how it runs in SWITCH, but the variables shared between the edge and the second vertex b will be passed to the edges function. Trying to read the shared variables may lead to a race condition in which vertex b or one/some of it's edges are modifying those varibales. The client is expected to handle such race conditions, and failing to do so can lead to a possibly unsafe execution. Thus this mode is UNSAFE. It is safe to use the shared variables with vertex b in context modes NONE and SINGLE since in NONE vertex b cannot also be active, and in SINGLE there is no threading.

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 AVLNode Struct Reference

```
#include <AVL.h>
```

Data Fields

- void ∗ data
- int id
- int height
- struct AVLNode ∗ left
- struct AVLNode ∗ right

4.1.1 Field Documentation

4.1.1.1 data

```
void* AVLNode::data
```

Referenced by create_node(), destroy_avl_nodes(), find_node(), inorder_nodes(), insert(), postorder_nodes(), preorder_nodes(), remove_ID(), remove_node(), and stackify_nodes().

4.1.1.2 height

```
int AVLNode::height
```

Referenced by balance(), create_node(), destroy_avl_nodes(), insert_node(), left_rotate(), max_height(), remove↩
_node(), right_rotate(), and stackify_nodes().

4.1.1.3 id

```
int AVLNode::id
```

Referenced by create_node(), find_node(), inorder_nodes(), insert_node(), postorder_nodes(), preorder_nodes(), and remove_node().

4.1.1.4 left

```
struct AVLNode* AVLNode::left
```

Referenced by balance(), create_node(), destroy_avl_nodes(), find_node(), inorder_nodes(), insert_node(), left↩
_rotate(), max_height(), minNode(), postorder_nodes(), preorder_nodes(), remove_node(), right_rotate(), and stackify_nodes().

4.1.1.5 right

```
struct AVLNode* AVLNode::right
```

Referenced by balance(), create_node(), destroy_avl_nodes(), find_node(), inorder_nodes(), insert_node(), left_↩
rotate(), max_height(), postorder_nodes(), preorder_nodes(), remove_node(), right_rotate(), and stackify_nodes().

The documentation for this struct was generated from the following file:

- include/AVL.h

4.2 AVLTree Struct Reference

```
#include <AVL.h>
```

Data Fields

- struct AVLNode ∗ root
- int size

4.2.1 Field Documentation

4.2.1.1 root

```
struct AVLNode* AVLTree::root
```

Referenced by destroy_avl(), find(), init_avl(), inorder(), insert(), postorder(), preorder(), remove_ID(), and stackify().

4.2.1.2 size

```
int AVLTree::size
```

Referenced by destroy_avl(), init_avl(), inorder(), insert(), postorder(), preorder(), print_graph(), print_state(), remove_ID(), and stackify().

The documentation for this struct was generated from the following file:

- include/AVL.h

4.3 destroy_edge_id_request Struct Reference

```
#include <edge.h>
```

Data Fields

- struct vertex ∗ a
- int id

4.3.1 Field Documentation

4.3.1.1 a

```
struct vertex* destroy_edge_id_request::a
```

Referenced by procces_request().

4.3.1.2 id

```
int destroy_edge_id_request::id
```

Referenced by procces_request().

The documentation for this struct was generated from the following file:

- include/edge.h

4.4 destroy_edge_request Struct Reference

```
#include <edge.h>
```

Data Fields

- struct vertex ∗ a
- struct vertex ∗ b

4.4.1 Field Documentation

4.4.1.1 a

```
struct vertex* destroy_edge_request::a
```

Referenced by procces_request().

4.4.1.2 b

```
struct vertex* destroy_edge_request::b
```

Referenced by procces_request().

The documentation for this struct was generated from the following file:

- include/edge.h

4.5 destroy_vertex_id_request Struct Reference

```
#include <vertex.h>
```

Data Fields

- struct graph ∗ graph
- int id

4.5.1 Field Documentation

4.5.1.1 graph

```
struct graph* destroy_vertex_id_request::graph
```

Referenced by procces_request().

4.5.1.2 id

```
int destroy_vertex_id_request::id
```

Referenced by procces_request().

The documentation for this struct was generated from the following file:

- include/vertex.h

4.6 destroy_vertex_request Struct Reference

```
#include <vertex.h>
```

Data Fields

- struct graph * graph
- struct vertex * vertex

4.6.1 Field Documentation

4.6.1.1 graph

```
struct graph* destroy_vertex_request::graph
```

Referenced by procces_request().

4.6.1.2 vertex

```
struct vertex* destroy_vertex_request::vertex
```

Referenced by procces_request().

The documentation for this struct was generated from the following file:

- include/vertex.h

4.7 edge Struct Reference

```
#include <edge.h>
```

Data Fields

- int id
- int(∗ f)(void ∗, void ∗, const void ∗const a_vars, const void ∗const b_vars)
- void ∗ glbl
- const void ∗const ∗ a_vars
- const void ∗const ∗ b_vars
- struct vertex ∗ a
- struct vertex ∗ b
- enum edge_type edge_type
- struct edge ∗ bi_edge
- pthread_mutex_t bi_edge_lock

4.7.1 Detailed Description

Edge

4.7.2 Field Documentation

4.7.2.1 a

```
struct vertex* edge::a
```

Referenced by create_bi_edge(), create_edge(), modify_bi_edge(), modify_edge(), print_edges(), remove_bi_↩ edge(), remove_edge(), remove_edge_id(), and remove_vertex().

4.7.2.2 a_vars

```
const void* const* edge::a_vars
```

Referenced by create_edge(), fire(), remove_edge(), remove_edge_id(), and run_single().

4.7.2.3 b

```
struct vertex* edge::b
```

Referenced by create_bi_edge(), create_edge(), fire(), modify_bi_edge(), modify_edge(), print_edges(), remove↩ _bi_edge(), remove_edge(), remove_edge_id(), remove_vertex(), and run_single().

4.7.2.4 b_vars

```
const void* const* edge::b_vars
```

Referenced by create_edge(), fire(), remove_edge(), remove_edge_id(), and run_single().

4.7.2.5 bi_edge

```
struct edge* edge::bi_edge
```

Referenced by create_bi_edge(), create_edge(), print_edges(), remove_edge(), remove_edge_id(), and remove↩
_vertex().

4.7.2.6 bi_edge_lock

```
pthread_mutex_t edge::bi_edge_lock
```

Referenced by create_bi_edge(), fire(), remove_edge(), remove_edge_id(), and remove_vertex().

4.7.2.7 edge_type

```
enum edge_type edge::edge_type
```

Referenced by create_bi_edge(), create_edge(), fire(), print_edges(), remove_edge(), remove_edge_id(), and remove_vertex().

4.7.2.8 f

```
int(* edge::f) (void *, void *, const void *const a_vars, const void *const b_vars)
```

Referenced by create_bi_edge(), create_edge(), fire(), modify_bi_edge(), modify_edge(), print_edges(), remove↩
_edge(), remove_edge_id(), remove_vertex(), and run_single().

4.7.2.9 glbl

```
void* edge::glbl
```

Referenced by create_bi_edge(), create_edge(), fire(), modify_bi_edge(), modify_edge(), print_edges(), print_↩
state(), remove_edge(), remove_edge_id(), remove_vertex(), and run_single().

4.7.2.10 id

```
int edge::id
```

Referenced by create_edge(), print_edges(), remove_edge(), remove_edge_id(), and remove_vertex().

The documentation for this struct was generated from the following file:

- include/edge.h

4.8 edge_request Struct Reference

```
#include <edge.h>
```

Data Fields

- struct vertex ∗ a
- struct vertex ∗ b
- int(∗ f)(void ∗, void ∗, const void ∗const, const void ∗const)
- void ∗ glbl

4.8.1 Field Documentation

4.8.1.1 a

```
struct vertex* edge_request::a
```

Referenced by procces_request().

4.8.1.2 b

```
struct vertex* edge_request::b
```

Referenced by procces_request().

4.8.1.3 f

```
int(* edge_request::f) (void *, void *, const void *const, const void *const)
```

Referenced by procces_request().

4.8.1.4 glbl

```
void* edge_request::glbl
```

Referenced by procces_request().

The documentation for this struct was generated from the following file:

- include/edge.h

4.9 fireable Struct Reference

```
#include <topologic.h>
```

Data Fields

- struct graph ∗ graph
- struct vertex ∗ vertex
- struct vertex_result ∗ args
- enum STATES color
- int iloop

4.9.1 Field Documentation

4.9.1.1 args

```
struct vertex_result* fireable::args
```

Referenced by fire(), fire_pthread(), run(), and switch_vertex().

4.9.1.2 color

```
enum STATES fireable::color
```

Referenced by fire(), fire_pthread(), run(), and switch_vertex().

4.9.1.3 graph

```
struct graph* fireable::graph
```

Referenced by fire_pthread(), run(), and switch_vertex().

4.9.1.4 iloop

```
int fireable::iloop
```

Referenced by fire(), fire_pthread(), run(), and switch_vertex().

4.9.1.5 vertex

```
struct vertex* fireable::vertex
```

Referenced by fire_pthread(), run(), and switch_vertex().

The documentation for this struct was generated from the following file:

- include/topologic.h

4.10 graph Struct Reference

```
#include <graph.h>
```

Data Fields

- enum CONTEXT context
- enum MEM_OPTION mem_option
- struct AVLTree ∗ vertices
- struct stack ∗ start
- struct stack ∗ modify
- struct stack ∗ remove_edges
- struct stack ∗ remove_vertices
- int max_state_changes
- int max_loop
- int snapshot_timestamp
- unsigned int lvl_verbose
- int state_count
- pthread_mutex_t lock
- pthread_mutex_t color_lock
- sig_atomic_t state
- sig_atomic_t previous_color
- sig_atomic_t print_flag
- sig_atomic_t red_vertex_count
- sig_atomic_t black_vertex_count
- sig_atomic_t pause
- sig_atomic_t red_locked
- sig_atomic_t black_locked
- sig_atomic_t num_vertices
- pthread_cond_t pause_cond
- pthread_cond_t red_fire
- pthread_cond_t black_fire

4.10.1 Detailed Description

Graph

4.10.2 Field Documentation

4.10.2.1 black_fire

```
pthread_cond_t graph::black_fire
```

Referenced by destroy_graph(), fire(), graph_init(), and run().

4.10.2.2 black_locked

```
sig_atomic_t graph::black_locked
```

Referenced by destroy_graph(), fire(), graph_init(), and run().

4.10.2.3 black_vertex_count

```
sig_atomic_t graph::black_vertex_count
```

Referenced by destroy_graph(), fire(), graph_init(), and run().

4.10.2.4 color_lock

```
pthread_mutex_t graph::color_lock
```

Referenced by destroy_graph(), graph_init(), and run().

4.10.2.5 context

```
enum CONTEXT graph::context
```

Referenced by create_vertex(), destroy_graph(), fire(), graph_init(), print_graph(), process_requests(), remove_↩
vertex(), remove_vertex_id(), run(), run_single(), start_set(), and submit_request().

4.10.2.6 lock

```
pthread_mutex_t graph::lock
```

Referenced by create_vertex(), destroy_graph(), fire(), graph_init(), pause_graph(), print_graph(), process_↩
requests(), remove_vertex(), remove_vertex_id(), resume_graph(), run(), run_single(), and submit_request().

4.10.2.7 lvl_verbose

```
unsigned int graph::lvl_verbose
```

Referenced by graph_init(), print_edges(), print_graph(), and print_state().

4.10.2.8 max_loop

```
int graph::max_loop
```

Referenced by fire(), graph_init(), print_graph(), and run_single().

4.10.2.9 max_state_changes

```
int graph::max_state_changes
```

Referenced by fire(), graph_init(), print_graph(), run(), and run_single().

4.10.2.10 mem_option

```
enum MEM_OPTION graph::mem_option
```

Referenced by graph_init(), run(), and switch_vertex().

4.10.2.11 modify

```
struct stack* graph::modify
```

Referenced by destroy_graph(), graph_init(), process_requests(), and submit_request().

4.10.2.12 num_vertices

```
sig_atomic_t graph::num_vertices
```

Referenced by fire(), graph_init(), print_graph(), run(), and run_single().

4.10.2.13 pause

```
sig_atomic_t graph::pause
```

Referenced by graph_init(), pause_graph(), resume_graph(), run(), and run_single().

4.10.2.14 pause_cond

```
pthread_cond_t graph::pause_cond
```

Referenced by destroy_graph(), graph_init(), pause_graph(), run(), and run_single().

4.10.2.15 previous_color

```
sig_atomic_t graph::previous_color
```

Referenced by graph_init(), and run().

4.10.2.16 print_flag

```
sig_atomic_t graph::print_flag
```

Referenced by graph_init(), and run().

4.10.2.17 red_fire

```
pthread_cond_t graph::red_fire
```

Referenced by destroy_graph(), fire(), graph_init(), and run().

4.10.2.18 red_locked

```
sig_atomic_t graph::red_locked
```

Referenced by destroy_graph(), fire(), graph_init(), and run().

4.10.2.19 red_vertex_count

```
sig_atomic_t graph::red_vertex_count
```

Referenced by destroy_graph(), fire(), graph_init(), and run().

4.10.2.20 remove_edges

```
struct stack* graph::remove_edges
```

Referenced by destroy_graph(), graph_init(), process_requests(), and submit_request().

4.10.2.21 remove_vertices

```
struct stack* graph::remove_vertices
```

Referenced by destroy_graph(), graph_init(), process_requests(), and submit_request().

4.10.2.22 snapshot_timestamp

```
int graph::snapshot_timestamp
```

Referenced by graph_init(), and print_graph().

4.10.2.23 start

```
struct stack* graph::start
```

Referenced by destroy_graph(), graph_init(), run(), run_single(), and start_set().

4.10.2.24 state

```
sig_atomic_t graph::state
```

Referenced by destroy_graph(), fire(), graph_init(), run(), and run_single().

4.10.2.25 state_count

```
int graph::state_count
```

Referenced by fire(), graph_init(), print_graph(), run(), and run_single().

4.10.2.26 vertices

```
struct AVLTree* graph::vertices
```

Referenced by create_vertex(), destroy_graph(), graph_init(), print_graph(), print_state(), remove_vertex(), remove_vertex_id(), and start_set().

The documentation for this struct was generated from the following file:

- include/graph.h

4.11 mod_edge_vars_request Struct Reference

```
#include <vertex.h>
```

Data Fields

- struct vertex ∗ vertex
- void ∗ edge_vars

4.11.1 Field Documentation

4.11.1.1 edge_vars

```
void* mod_edge_vars_request::edge_vars
```

Referenced by procces_request().

4.11.1.2 vertex

```
struct vertex* mod_edge_vars_request::vertex
```

Referenced by procces_request().

The documentation for this struct was generated from the following file:

- include/vertex.h

4.12 mod_vertex_request Struct Reference

```
#include <vertex.h>
```

Data Fields

- struct vertex ∗ vertex
- void(∗ f)(struct graph ∗, struct vertex_result ∗, void ∗, void ∗)
- void ∗ glbl

4.12.1 Field Documentation

4.12.1.1 f

```
void(* mod_vertex_request::f) (struct graph *, struct vertex_result *, void *, void *)
```

Referenced by procces_request().

4.12.1.2 glbl

```
void* mod_vertex_request::glbl
```

Referenced by procces_request().

4.12.1.3 vertex

```
struct vertex* mod_vertex_request::vertex
```

Referenced by procces_request().

The documentation for this struct was generated from the following file:

- include/vertex.h

4.13 request Struct Reference

```
#include <request.h>
```

Data Fields

- enum REQUESTS request
- void(∗ f)(void ∗)
- void ∗ args

4.13.1 Detailed Description

Request

4.13.2 Field Documentation

4.13.2.1 args

```
void* request::args
```

Referenced by create_request(), destroy_request(), and procces_request().

4.13.2.2 f

```
void(* request::f) (void *)
```

Referenced by create_request(), destroy_request(), and procces_request().

4.13.2.3 request

```
enum REQUESTS request::request
```

Referenced by create_request(), destroy_request(), procces_request(), and submit_request().

The documentation for this struct was generated from the following file:

- include/request.h

4.14 shared_edge Union Reference

```
#include <vertex.h>
```

Data Fields

- void ∗ vertex_data
- const void ∗const ∗ edge_data

4.14.1 Field Documentation

4.14.1.1 edge_data

```
const void∗ const∗ shared_edge::edge_data
```

Referenced by create_edge().

4.14.1.2 vertex_data

```
void∗ shared_edge::vertex_data
```

Referenced by create_vertex(), fire(), modify_shared_edge_vars(), remove_vertex(), and run_single().

The documentation for this union was generated from the following file:

- include/vertex.h

4.15 stack Struct Reference

```
#include <stack.h>
```

Data Fields

- struct stack_node ∗ root
- int length

4.15.1 Field Documentation

4.15.1.1 length

```
int stack::length
```

Referenced by destroy_stack(), get(), init_stack(), pop(), print_edges(), print_state(), push(), and run_single().

4.15.1.2 root

```
struct stack_node* stack::root
```

Referenced by destroy_stack(), get(), init_stack(), pop(), and push().

The documentation for this struct was generated from the following file:

- include/stack.h

4.16 stack_node Struct Reference

```
#include <stack.h>
```

Data Fields

- void * data
- struct stack_node * next

4.16.1 Field Documentation

4.16.1.1 data

```
void* stack_node::data
```

Referenced by destroy_stack(), get(), pop(), and push().

4.16.1.2 next

```
struct stack_node* stack_node::next
```

Referenced by destroy_stack(), get(), pop(), and push().

The documentation for this struct was generated from the following file:

- include/stack.h

4.17 vertex Struct Reference

```
#include <vertex.h>
```

Data Fields

- int id
- int is_active
- void(∗ f)(struct graph ∗, struct vertex_result ∗, void ∗, void ∗)
- void ∗ glbl
- union shared_edge ∗ shared
- pthread_mutex_t lock
- struct AVLTree ∗ edge_tree
- struct AVLTree ∗ joining_vertices
- enum CONTEXT context

4.17.1 Detailed Description

vertex

4.17.2 Field Documentation

4.17.2.1 context

```
enum CONTEXT vertex::context
```

Referenced by create_bi_edge(), create_edge(), create_vertex(), modify_edge(), modify_shared_edge_vars(), modify_vertex(), remove_edge(), remove_edge_id(), and remove_vertex().

4.17.2.2 edge_tree

```
struct AVLTree∗ vertex::edge_tree
```

Referenced by create_edge(), create_vertex(), fire(), modify_edge(), print_state(), remove_edge(), remove_edge←↩
_id(), remove_vertex(), and run_single().

4.17.2.3 f

```
void(∗ vertex::f) (struct graph ∗, struct vertex_result ∗, void ∗, void ∗)
```

Referenced by create_vertex(), fire(), modify_vertex(), print_state(), and run_single().

4.17.2.4 glbl

```
void* vertex::glbl
```

Referenced by create_vertex(), fire(), modify_vertex(), print_state(), remove_vertex(), and run_single().

4.17.2.5 id

```
int vertex::id
```

Referenced by create_edge(), create_vertex(), modify_edge(), print_state(), remove_edge(), remove_edge_id(), remove_vertex(), and start_set().

4.17.2.6 is_active

```
int vertex::is_active
```

Referenced by create_vertex(), fire(), print_state(), run(), and run_single().

4.17.2.7 joining_vertices

```
struct AVLTree* vertex::joining_vertices
```

Referenced by create_edge(), create_vertex(), remove_edge(), remove_edge_id(), and remove_vertex().

4.17.2.8 lock

```
pthread_mutex_t vertex::lock
```

Referenced by create_edge(), create_vertex(), fire(), modify_edge(), modify_shared_edge_vars(), modify_vertex(), remove_edge(), remove_edge_id(), and remove_vertex().

4.17.2.9 shared

```
union shared_edge* vertex::shared
```

Referenced by create_edge(), create_vertex(), fire(), modify_shared_edge_vars(), print_state(), remove_vertex(), and run_single().

The documentation for this struct was generated from the following file:

- include/vertex.h

4.18 vertex_request Struct Reference

```
#include <vertex.h>
```

Data Fields

- struct graph ∗ graph
- int id
- void(∗ f)(struct graph ∗, struct vertex_result ∗, void ∗, void ∗)
- void ∗ glbl

4.18.1 Field Documentation

4.18.1.1 f

```
void(* vertex_request::f) (struct graph *, struct vertex_result *, void *, void *)
```

Referenced by procces_request().

4.18.1.2 glbl

```
void* vertex_request::glbl
```

Referenced by procces_request().

4.18.1.3 graph

```
struct graph* vertex_request::graph
```

Referenced by procces_request().

4.18.1.4 id

```
int vertex_request::id
```

Referenced by procces_request().

The documentation for this struct was generated from the following file:

- include/vertex.h

4.19 vertex_result Struct Reference

```
#include <vertex.h>
```

Data Fields

- void ∗ vertex_argv
- size_t vertex_size
- void ∗ edge_argv
- size_t edge_size

4.19.1 Detailed Description

vertex_result

4.19.2 Field Documentation

4.19.2.1 edge_argv

```
void* vertex_result::edge_argv
```

Referenced by fire(), run_single(), and switch_vertex().

4.19.2.2 edge_size

```
size_t vertex_result::edge_size
```

Referenced by switch_vertex().

4.19.2.3 vertex_argv

```
void* vertex_result::vertex_argv
```

Referenced by fire(), run_single(), and switch_vertex().

4.19.2.4 vertex_size

```
size_t vertex_result::vertex_size
```

Referenced by switch_vertex().

The documentation for this struct was generated from the following file:

- include/vertex.h

Chapter 5

File Documentation

5.1 include/AVL.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include "./stack.h"
```

Data Structures

- struct AVLNode
- struct AVLTree

Functions

- struct AVLTree ∗ init_avl ()
- int insert (struct AVLTree ∗tree, void ∗data, int id)
- void ∗ remove_ID (struct AVLTree ∗tree, int id)
- void ∗ find (struct AVLTree ∗tree, int id)
- void inorder (struct AVLTree ∗tree, struct stack ∗stack)
- void postorder (struct AVLTree ∗tree, struct stack ∗stack)
- void preorder (struct AVLTree ∗tree, struct stack ∗stack)
- void stackify (struct AVLTree ∗tree, struct stack ∗stack)
- void destroy_avl (struct AVLTree ∗tree)

5.1.1 Function Documentation

5.1.1.1 destroy_avl()

```
void destroy_avl (
            struct AVLTree * tree )
```

References destroy_avl_nodes(), AVLTree::root, and AVLTree::size.

Referenced by create_vertex(), destroy_graph_avl(), and graph_init().

5.1.1.2 find()

```
void* find (
            struct AVLTree * tree,
            int id )
```

References find_node(), and AVLTree::root.

Referenced by create_edge(), modify_edge(), remove_vertex_id(), and start_set().

5.1.1.3 init_avl()

```
struct AVLTree* init_avl ( )
```

References AVLTree::root, and AVLTree::size.

Referenced by create_vertex(), and graph_init().

5.1.1.4 inorder()

```
void inorder (
            struct AVLTree * tree,
            struct stack * stack )
```

References inorder_nodes(), AVLTree::root, and AVLTree::size.

Referenced by print_state().

5.1.1.5 insert()

```
int insert (
            struct AVLTree * tree,
            void * data,
            int id )
```

References create_node(), AVLNode::data, insert_node(), AVLTree::root, and AVLTree::size.

Referenced by create_edge(), create_vertex(), and insert_node().

5.1.1.6 postorder()

```
void postorder (
             struct AVLTree * tree,
             struct stack * stack )
```

References postorder_nodes(), AVLTree::root, and AVLTree::size.

5.1.1.7 preorder()

```
void preorder (
             struct AVLTree * tree,
             struct stack * stack )
```

References preorder_nodes(), AVLTree::root, and AVLTree::size.

Referenced by destroy_graph_avl(), fire(), print_edges(), and run_single().

5.1.1.8 remove_ID()

```
void* remove_ID (
             struct AVLTree * tree,
             int id )
```

References AVLNode::data, remove_node(), AVLTree::root, and AVLTree::size.

Referenced by create_edge(), remove_edge(), remove_edge_id(), and remove_vertex().

5.1.1.9 stackify()

```
void stackify (
             struct AVLTree * tree,
             struct stack * stack )
```

References AVLTree::root, AVLTree::size, and stackify_nodes().

Referenced by remove_vertex().

5.2 include/context.h File Reference

Enumerations

- enum CONTEXT { NONE = 0 , SINGLE = 1 , SWITCH = 2 , SWITCH_UNSAFE = 3 }

5.2.1 Enumeration Type Documentation

5.2.1.1 CONTEXT

```
enum CONTEXT
```

Enum for how the graph handles context switches, or not at all NONE: First valid edge is taken only and the process does not change SINGLE: Is the same as NONE but only one vertex may be selected as start SWITCH: All valid edges are taken. A process per vertex is spawned and previous process is killed. SWITCH_UNSAGE: Same as SWITCH but will pass edge->b_vars to edge->f

Enumerator

NONE	
SINGLE	
SWITCH	
SWITCH_UNSAFE	

5.3 include/edge.h File Reference

```
#include <stdio.h>
#include "./vertex.h"
```

Data Structures

- struct edge
- struct edge_request
- struct destroy_edge_request
- struct destroy_edge_id_request

Enumerations

- enum edge_type { EDGE = 0 , BI_EDGE = 1 , SELF_EDGE = 2 }

5.3.1 Enumeration Type Documentation

5.3.1.1 edge_type

```
enum edge_type
```

Enumerator

EDGE	
BI_EDGE	
SELF_EDGE	

5.4 include/graph.h File Reference

```
#include <pthread.h>
#include <signal.h>
```

```
#include "./context.h"
```

Data Structures

- struct graph

Macros

- #define MAX_ATTEMPTS 4
- #define THREAD_ATTEMPT_SLEEP 3

Enumerations

- enum MEM_OPTION { ABORT = 0 , WAIT = 1 , CONTINUE = 2 }
- enum STATES { PRINT = 0 , RED = 1 , BLACK = 2 , TERMINATE = 3 }
- enum SNAPSHOT { NO_SNAP = -1 , START_STOP = 0 }
- enum VERBOSITY {
  NO_VERB = 0 , VERTICES = 1 , EDGES = 2 , FUNCTIONS = 4 ,
  GLOBALS = 8 }

5.4.1 Macro Definition Documentation

5.4.1.1 MAX_ATTEMPTS

```
#define MAX_ATTEMPTS 4
```

5.4.1.2 THREAD_ATTEMPT_SLEEP

```
#define THREAD_ATTEMPT_SLEEP 3
```

5.4.2 Enumeration Type Documentation

5.4.2.1 MEM_OPTION

```
enum MEM_OPTION
```

Enum for memory handling when forking – if mem amount is exceeded ABORT: Kill the program and clean WAIT: Wait until there is enough memory and try again CONTINUE: Ignore edge error and move on

Enumerator

ABORT	
WAIT	
CONTINUE	

5.4.2.2 SNAPSHOT

```
enum SNAPSHOT
```

Enum for snapshots. NONE: Record nothing START_STOP: Record first and last state

Enumerator

NO_SNAP	
START_STOP	

5.4.2.3 STATES

```
enum STATES
```

Enum for state of global manager – locking when printing and firing; Ensures proper locking between printing node information and firing information PRINT: Print, then lock RED: # Readers = 0 -> unlock print, goes to BLACK at finish BLACK: Wait until # Readers -> 0, then print

Enumerator

PRINT	
RED	
BLACK	
TERMINATE	

5.4.2.4 VERBOSITY

```
enum VERBOSITY
```

Enum for how verbose the records are NONE: Record nothing NODES: Record nodes EDGES: Record edges FUNCTIONS: Record the functions of nodes and/or edges GLOBALS: Record the globals of nodes and/or edges; Also will record shared edges

Enumerator

NO_VERB	
VERTICES	
EDGES	
FUNCTIONS	
GLOBALS	

5.5 include/header.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
```

5.6 include/request.h File Reference

Data Structures

- struct request

Enumerations

- enum REQUESTS {
  CREAT_VERTEX = 0 , CREAT_EDGE = 1 , CREAT_BI_EDGE = 2 , MOD_VERTEX = 3 ,
  MOD_EDGE_VARS = 4 , MOD_EDGE = 5 , MOD_BI_EDGE = 6 , DESTROY_VERTEX = 7 ,
  DESTROY_VERTEX_BY_ID = 8 , DESTROY_EDGE = 9 , DESTROY_BI_EDGE = 10 , DESTROY_EDGE_BY_ID = 11 ,
  GENERIC = 12 }

5.6.1 Enumeration Type Documentation

5.6.1.1 REQUESTS

`enum REQUESTS`

Enum for submiting a request to be handles MODIFY: Modify values in existing edges or vertices as well as add vertices or edges DESTROY_VERTEX: Remove vertex from graph DESTROY_EDGE: Remove edge from graph

Enumerator

CREAT_VERTEX	
CREAT_EDGE	
CREAT_BI_EDGE	
MOD_VERTEX	
MOD_EDGE_VARS	
MOD_EDGE	
MOD_BI_EDGE	
DESTROY_VERTEX	
DESTROY_VERTEX_BY_ID	
DESTROY_EDGE	
DESTROY_BI_EDGE	
DESTROY_EDGE_BY_ID	
GENERIC	

5.7 include/stack.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
```

Data Structures

- struct stack_node
- struct stack

Functions

- struct stack ∗ init_stack ()
- void ∗ get (struct stack ∗stack, int index)
- void ∗ pop (struct stack ∗stack)
- int push (struct stack ∗stack, void ∗data)
- void destroy_stack (struct stack ∗stack)

5.7.1 Function Documentation

5.7.1.1 destroy_stack()

```
void destroy_stack (
          struct stack ∗ stack )
```

References stack_node::data, stack::length, stack_node::next, and stack::root.

Referenced by destroy_graph_avl(), destroy_graph_stack(), fire(), graph_init(), print_edges(), print_state(), remove_vertex(), and run_single().

5.7.1.2 get()

```
void* get (
            struct stack * stack,
            int index )
```

References stack_node::data, stack::length, stack_node::next, and stack::root.

5.7.1.3 init_stack()

```
struct stack* init_stack ( )
```

References stack::length, and stack::root.

Referenced by destroy_graph_avl(), fire(), graph_init(), print_edges(), print_state(), remove_vertex(), and run_↩
single().

5.7.1.4 pop()

```
void* pop (
            struct stack * stack )
```

References stack_node::data, stack::length, stack_node::next, and stack::root.

Referenced by destroy_graph_avl(), destroy_graph_stack(), fire(), print_edges(), print_state(), process_requests(), remove_vertex(), run(), run_single(), and start_set().

5.7.1.5 push()

```
int push (
            struct stack * stack,
            void * data )
```

References stack_node::data, stack::length, stack_node::next, and stack::root.

Referenced by inorder_nodes(), postorder_nodes(), preorder_nodes(), stackify_nodes(), start_set(), and submit_↩
request().

5.8 include/topologic.h File Reference

```
#include "./header.h"
#include "./stack.h"
#include "./AVL.h"
#include "./edge.h"
#include "./vertex.h"
#include "./request.h"
#include "./graph.h"
```

Data Structures

- struct fireable

Macros

- #define PTHREAD_SLEEP_TIME 50
- #define _GNU_SOURCE
- #define TOPOLOGIC_DEBUG 0
- #define topologic_debug(fmt, ...)
- #define MAX_LOOPS 100
- #define GRAPH_INIT() graph_init(-1, START_STOP, MAX_LOOPS, VERTICES | EDGES | FUNCTIONS | GLOBALS, SINGLE, CONTINUE)
- #define CREATE_VERTEX(graph, f, id) create_vertex(graph, f, id, NULL, PROTECT_B_VARS)
- #define CREATE_VERTEX_GLBL(graph, f, id, glbl) create_vertex(graph, f, id, glbl, PROTECT_B_VARS)
- #define CREATE_EDGE(a, b, f) create_edge(a, b, f, NULL)
- #define CREATE_NULL_BI_EDGE(a, b, f) create_bi_edge(a, b, f, NULL, NULL, NULL)
- #define CREATE_BI_EDGE(a, b, f, a_to_b, b_to_a) create_bi_edge(a, b, f, NULL, a_to_b, b_to_a)
- #define MODIFY_VERTEX(vertex, f) modify_vertex(vertex, f, NULL)
- #define MODIFY_VERTEX_GLOBALS(vertex, glbl) modify_vertex(vertex, NULL, glbl)
- #define MODIFY_EDGE(a, b, f) modify_edge(a, b, f, NULL)
- #define MODIFY_EDGE_GLOBALS(a, b, glbl) modify_edge(a, b, NULL, glbl)
- #define MODIFY_BI_EDGE(a, b, f) modify_bi_edge(a, b, f, NULL)
- #define MODIFY_BI_EDGE_GLOBALS(a, b, glbl) modify_bi_edge(a, b, NULL, glbl)
- #define CREATE_REQUEST(request, args) create_request(request, args, NULL)

Functions

- void ∗ fire_pthread (void ∗vargp)
- struct graph ∗ graph_init (int max_state_changes, int snapshot_timestamp, int max_loop, unsigned int lvl_↩ verbose, enum CONTEXT context, enum MEM_OPTION mem_option)
- struct vertex ∗ create_vertex (struct graph ∗graph, void(∗f)(struct graph ∗, struct vertex_result ∗, void ∗, void ∗), int id, void ∗glbl)
- struct edge ∗ create_edge (struct vertex ∗a, struct vertex ∗b, int(∗f)(void ∗, void ∗, const void ∗const, const void ∗const), void ∗glbl)
- int create_bi_edge (struct vertex ∗a, struct vertex ∗b, int(∗f)(void ∗, void ∗, const void ∗const, const void ∗const), void ∗glbl, struct edge ∗∗edge_a_to_b, struct edge ∗∗edge_b_to_a)
- int remove_edge (struct vertex ∗a, struct vertex ∗b)
- int remove_edge_id (struct vertex ∗a, int id)
- int remove_bi_edge (struct vertex ∗a, struct vertex ∗b)
- int remove_vertex (struct graph ∗graph, struct vertex ∗vertex)
- int remove_vertex_id (struct graph ∗graph, int id)
- int modify_vertex (struct vertex ∗vertex, void(∗f)(struct graph ∗, struct vertex_result ∗, void ∗, void ∗), void ∗glbl)
- int modify_shared_edge_vars (struct vertex ∗vertex, void ∗edge_vars)
- int modify_edge (struct vertex ∗a, struct vertex ∗b, int(∗f)(void ∗, void ∗, const void ∗const, const void ∗const), void ∗glbl)
- int modify_bi_edge (struct vertex ∗a, struct vertex ∗b, int(∗f)(void ∗, void ∗, const void ∗const, const void ∗const), void ∗glbl)
- int fire (struct graph ∗graph, struct vertex ∗vertex, struct vertex_result ∗args, enum STATES color, int iloop)
- int switch_vertex (struct graph ∗graph, struct vertex ∗vertex, struct vertex_result ∗args, enum STATES color, int iloop)
- int start_set (struct graph ∗graph, int id[ ], int num_vertices)

- int submit_request (struct graph ∗, struct request ∗request)
- struct request ∗ create_request (enum REQUESTS request, void ∗args, void(∗f)(void ∗))
- int process_requests (struct graph ∗graph)
- int run (struct graph ∗graph, struct vertex_result ∗∗vertex_args)
- int resume_graph (struct graph ∗graph)
- int pause_graph (struct graph ∗graph)
- void print_graph (struct graph ∗graph)
- struct graph ∗ parse_json (const char ∗path)
- int destroy_graph (struct graph ∗graph)
- int destroy_request (struct request ∗request)

5.8.1 Macro Definition Documentation

5.8.1.1 _GNU_SOURCE

```
#define _GNU_SOURCE
```

5.8.1.2 CREATE_BI_EDGE

```
#define CREATE_BI_EDGE(
              a,
              b,
              f,
              a_to_b,
              b_to_a ) create_bi_edge(a, b, f, NULL, a_to_b, b_to_a)
```

5.8.1.3 CREATE_EDGE

```
#define CREATE_EDGE(
              a,
              b,
              f ) create_edge(a, b, f, NULL)
```

5.8.1.4 CREATE_NULL_BI_EDGE

```
#define CREATE_NULL_BI_EDGE(
              a,
              b,
              f ) create_bi_edge(a, b, f, NULL, NULL, NULL)
```

5.8.1.5 CREATE_REQUEST

```
#define CREATE_REQUEST(
              request,
              args ) create_request(request, args, NULL)
```

5.8.1.6 CREATE_VERTEX

```
#define CREATE_VERTEX(
              graph,
              f,
              id ) create_vertex(graph, f, id, NULL, PROTECT_B_VARS)
```

5.8.1.7 CREATE_VERTEX_GLBL

```
#define CREATE_VERTEX_GLBL(
              graph,
              f,
              id,
              glbl ) create_vertex(graph, f, id, glbl, PROTECT_B_VARS)
```

5.8.1.8 GRAPH_INIT

```
#define GRAPH_INIT( ) graph_init(-1, START_STOP, MAX_LOOPS, VERTICES | EDGES | FUNCTIONS | GLOBALS,
SINGLE, CONTINUE)
```

5.8.1.9 MAX_LOOPS

```
#define MAX_LOOPS 100
```

5.8.1.10 MODIFY_BI_EDGE

```
#define MODIFY_BI_EDGE(
              a,
              b,
              f ) modify_bi_edge(a, b, f, NULL)
```

5.8.1.11 MODIFY_BI_EDGE_GLOBALS

```
#define MODIFY_BI_EDGE_GLOBALS(
              a,
              b,
              glbl ) modify_bi_edge(a, b, NULL, glbl)
```

5.8.1.12 MODIFY_EDGE

```
#define MODIFY_EDGE(
              a,
              b,
              f ) modify_edge(a, b, f, NULL)
```

5.8.1.13 MODIFY_EDGE_GLOBALS

```
#define MODIFY_EDGE_GLOBALS(
              a,
              b,
              glbl ) modify_edge(a, b, NULL, glbl)
```

5.8.1.14 MODIFY_VERTEX

```
#define MODIFY_VERTEX(
              vertex,
              f ) modify_vertex(vertex, f, NULL)
```

5.8.1.15 MODIFY_VERTEX_GLOBALS

```
#define MODIFY_VERTEX_GLOBALS(
              vertex,
              glbl ) modify_vertex(vertex, NULL, glbl)
```

5.8.1.16 PTHREAD_SLEEP_TIME

```
#define PTHREAD_SLEEP_TIME 50
```

5.8.1.17 TOPOLOGIC_DEBUG

```
#define TOPOLOGIC_DEBUG 0
```

5.8.1.18 topologic_debug

```
#define topologic_debug(
                fmt,
                ... )
```

Value:

```
do {if (TOPOLOGIC_DEBUG) fprintf(stderr,
      "%s:%s:%s:%d:%s(): " fmt "\n", __DATE__, __TIME__, \
__FILE__,  __LINE__, __func__,
      __VA_ARGS__); } while (0)
```

5.8.2 Function Documentation

5.8.2.1 create_bi_edge()

```
int create_bi_edge (
            struct vertex * a,
            struct vertex * b,
            int(*)(void *, void *, const void *const, const void *const) f,
            void * glbl,
            struct edge ** edge_a_to_b,
            struct edge ** edge_b_to_a )
```

@RETURNS 0 for success; -1 for fail See create_edge Will create an bidirectional edge between vertex a and b with some criteria determined by the function f. Will store the edges in edge_a and edge_b. If edge_a_to_b or edge_b_to_a is NULL it will not.

References edge::a, edge::b, BI_EDGE, edge::bi_edge, edge::bi_edge_lock, vertex::context, create_edge(), edge::edge_type, edge::f, edge::glbl, remove_edge(), SINGLE, and topologic_debug.

Referenced by procces_request().

5.8.2.2 create_edge()

```
struct edge* create_edge (
            struct vertex * a,
            struct vertex * b,
            int(*)(void *, void *, const void *const, const void *const) f,
            void * glbl )
```

@PARAM a: A vertex @PARAM b: Another vertex (can be 'a') @PARAM f: a function @PARAM glbl: global variables @RETURN the edge connecting a to b Will create an edge from vertex a to b with some criteria determined by the function f. NOTE: NULL glbl will mean no global variables. f cannot be NULL.

References edge::a, edge::a_vars, edge::b, edge::b_vars, edge::bi_edge, vertex::context, EDGE, shared_edge↵::edge_data, vertex::edge_tree, edge::edge_type, edge::f, find(), edge::glbl, edge::id, vertex::id, insert(), vertex↵::joining_vertices, vertex::lock, remove_edge(), remove_ID(), SELF_EDGE, vertex::shared, SINGLE, SWITCH, and topologic_debug.

Referenced by create_bi_edge(), and procces_request().

5.8.2.3 create_request()

```
struct request* create_request (
            enum REQUESTS request,
            void * args,
            void(*)(void *) f )
```

@PARAM request: the desired request @PARAM args: the arguments needed for f @PARAM f: the function of the request @RETURN the request or NULL if it fails Creates a request structure to be called later

References request::args, CREAT_BI_EDGE, CREAT_EDGE, CREAT_VERTEX, DESTROY_BI_EDGE, DESTROY_EDGE, DESTROY_EDGE_BY_ID, DESTROY_VERTEX, DESTROY_VERTEX_BY_ID, request::f, GENERIC, MOD_BI_EDGE, MOD_EDGE, MOD_EDGE_VARS, MOD_VERTEX, request::request, and topologic↩_debug.

5.8.2.4 create_vertex()

```
struct vertex* create_vertex (
            struct graph * graph,
            void(*)(struct graph *, struct vertex_result *, void *, void *) f,
            int id,
            void * glbl )
```

@PARAM graph: the graph @PARAM f: a function @PARAM id: vertex id to be used: MUST BE UNIQUE @PARAM glbl: global variables @RETURN vertex: a vertex to be used in a graph On creation a process will be spawned for the vertex The vertex will compute function f when called NOTE: NULL glbl will mean no global variables. f cannot be NULL.

References graph::context, vertex::context, destroy_avl(), vertex::edge_tree, vertex::f, vertex::glbl, vertex::id, init_avl(), insert(), vertex::is_active, vertex::joining_vertices, graph::lock, vertex::lock, vertex::shared, SINGLE, topologic_debug, shared_edge::vertex_data, and graph::vertices.

Referenced by procces_request().

5.8.2.5 destroy_graph()

```
int destroy_graph (
            struct graph * graph )
```

@PARAM graph: the graph @RETURN -1 for fail; 0 for success Destroys and frees the graph

References graph::black_fire, graph::black_locked, graph::black_vertex_count, graph::color_lock, graph::context, destroy_graph_avl(), destroy_graph_stack(), graph::lock, graph::modify, graph::pause_cond, graph::red_fire, graph::red_locked, graph::red_vertex_count, graph::remove_edges, graph::remove_vertices, SINGLE, graph↩::start, graph::state, TERMINATE, topologic_debug, and graph::vertices.

5.8.2.6 destroy_request()

```
int destroy_request (
             struct request * request )
```

@PARAM request: a request @RETURN -1 for fail; 0 for success Destroys and frees a request

References request::args, request::f, request::request, and topologic_debug.

Referenced by destroy_graph_stack().

5.8.2.7 fire()

```
int fire (
             struct graph * graph,
             struct vertex * vertex,
             struct vertex_result * args,
             enum STATES color,
             int iloop )
```

@PARAM graph: the graph @PARAM vertex: A vertex to be ran @PARAM args: arguments @PARAM color: the state in which unlocks fire process if STATE is set to PRINT then fire will fail @PARAM iloop: the number of times that vertex fired in succession @RETURNS the result of the vertex fire will wake up the vertex and pass args to the vertex to compute its function and then call switch and clean itself up

References edge::a_vars, fireable::args, edge::b, edge::b_vars, BI_EDGE, edge::bi_edge_lock, BLACK, graph↩ ::black_fire, graph::black_locked, graph::black_vertex_count, fireable::color, graph::context, destroy_stack(), vertex_result::edge_argv, vertex::edge_tree, edge::edge_type, vertex::f, edge::f, fire(), edge::glbl, vertex::glbl, fireable::iloop, init_stack(), vertex::is_active, graph::lock, vertex::lock, graph::max_loop, graph::max_state_changes, NONE, graph::num_vertices, pop(), preorder(), PTHREAD_SLEEP_TIME, RED, graph::red_fire, graph::red_locked, graph::red_vertex_count, vertex::shared, sleep_ms(), graph::state, graph::state_count, SWITCH, SWITCH_↩ UNSAFE, switch_vertex(), TERMINATE, topologic_debug, vertex_result::vertex_argv, and shared_edge::vertex_↩ data.

Referenced by fire(), and fire_pthread().

5.8.2.8 fire_pthread()

```
void* fire_pthread (
             void * vargp )
```

Wrapper function for fire, fire_1 @PARAM vargp: arguments

References fireable::args, fireable::color, fire(), fireable::graph, fireable::iloop, PTHREAD_SLEEP_TIME, sleep_↩ ms(), topologic_debug, and fireable::vertex.

Referenced by run(), and switch_vertex().

5.8.2.9 graph_init()

```
struct graph* graph_init (
            int max_state_changes,
            int snapshot_timestamp,
            int max_loop,
            unsigned int lvl_verbose,
            enum CONTEXT context,
            enum MEM_OPTION mem_option )
```

@PARAM max_state_changes: # state changes before entering sink vertex due to infinite loop of states; -1 to ignore @PARAM snapshot_timestamp: printing out data at given timestamp for user; -1 for none; 0 for first and last state @PARAM lvl_verbose: how verbose timestamp print is @PARAM context: linear or context-switch based @RETURN an empty graph Creates a graph structures

References graph::black_fire, graph::black_locked, graph::black_vertex_count, graph::color_lock, graph::context, destroy_avl(), destroy_stack(), init_avl(), init_stack(), graph::lock, graph::lvl_verbose, graph::max_loop, graph↩ ::max_state_changes, graph::mem_option, graph::modify, NONE, graph::num_vertices, graph::pause, graph↩ ::pause_cond, graph::previous_color, graph::print_flag, RED, graph::red_fire, graph::red_locked, graph::red_↩ vertex_count, graph::remove_edges, graph::remove_vertices, SINGLE, graph::snapshot_timestamp, graph::start, graph::state, graph::state_count, topologic_debug, and graph::vertices.

5.8.2.10 modify_bi_edge()

```
int modify_bi_edge (
            struct vertex * a,
            struct vertex * b,
            int(*)(void *, void *, const void *const, const void *const) f,
            void * glbl )
```

@PARAM a: a vertex @PARAM b: another vertex @PARAM f: a function @PARAM glbl: global variables @↩ RETURN 0 for success; -1 for fail; -2 if only edge from a to b is modified; -3 if only edge from b to a is modified Modifies the edge between a and b's function and variables NOTE: NULL f, or glbl will mean no change.

References edge::a, edge::b, edge::f, edge::glbl, modify_edge(), and topologic_debug.

Referenced by procces_request().

5.8.2.11 modify_edge()

```
int modify_edge (
            struct vertex * a,
            struct vertex * b,
            int(*)(void *, void *, const void *const, const void *const) f,
            void * glbl )
```

@PARAM a: a vertex @PARAM b: another vertex @PARAM f: a function @PARAM glbl: global variables @↩ RETURN 0 for success; -1 for fail Modifies the edge connecting a to b's function NOTE: NULL f, or glbl will mean no change.

References edge::a, edge::b, vertex::context, vertex::edge_tree, edge::f, find(), edge::glbl, vertex::id, vertex::lock, SINGLE, and topologic_debug.

Referenced by modify_bi_edge(), and procces_request().

5.8.2.12 modify_shared_edge_vars()

```
int modify_shared_edge_vars (
            struct vertex * vertex,
            void * edge_vars )
```

@PARAM vertex: a vertex @PARAM edge_vars: shared variables @RETURN 0 for success; -1 for fail Modifies the vertices shared variables with it's edges

References vertex::context, vertex::lock, vertex::shared, SINGLE, topologic_debug, and shared_edge::vertex_data.

Referenced by procces_request().

5.8.2.13 modify_vertex()

```
int modify_vertex (
            struct vertex * vertex,
            void(*)(struct graph *, struct vertex_result *, void *, void *) f,
            void * glbl )
```

@PARAM vertex: a vertex @PARAM f: a function @PARAM glbl: global variables NOTE: NULL f, or glbl will mean no change. @RETURN 0 for success; -1 for fail Modifies the vertices function

References vertex::context, vertex::f, vertex::glbl, vertex::lock, SINGLE, and topologic_debug.

Referenced by procces_request().

5.8.2.14 parse_json()

```
struct graph* parse_json (
            const char * path )
```

@PARAM path: path to graph input file @RETURN the graph specified in the file

5.8.2.15 pause_graph()

```
int pause_graph (
            struct graph * graph )
```

@PARAM graph: the graph @RETURN 0 for success -1 if it fails Pauses run

References graph::lock, graph::pause, graph::pause_cond, and topologic_debug.

5.8.2.16 print_graph()

```
void print_graph (
            struct graph * graph )
```

@PARAM graph: the graph Prints the graph with desired output TODO: Print enums

References graph::context, graph::lock, graph::lvl_verbose, graph::max_loop, graph::max_state_changes, NO↩_VERB, graph::num_vertices, print_state(), SINGLE, AVLTree::size, graph::snapshot_timestamp, START_STOP, graph::state_count, topologic_debug, and graph::vertices.

Referenced by run(), and run_single().

5.8.2.17 process_requests()

```
int process_requests (
            struct graph * graph )
```

@PARAM graph: the graph @RETURN 0 if all got processed; -1 if a request failed will set ERRNO to the ENUM Process requests that are queued in the graph

References graph::context, graph::lock, graph::modify, pop(), procces_request(), graph::remove_edges, graph↩::remove_vertices, SINGLE, and topologic_debug.

Referenced by run(), and run_single().

5.8.2.18 remove_bi_edge()

```
int remove_bi_edge (
            struct vertex * a,
            struct vertex * b )
```

@PARAM a: a vertex @PARAM b: another vertex @RETURN 0 for success; -1 for fail; -2 if only edge from a to b is removed; -3 if only edge from b to a is removed Removes the edge connecting a to b

References edge::a, edge::b, remove_edge(), and topologic_debug.

Referenced by procces_request().

5.8.2.19 remove_edge()

```
int remove_edge (
            struct vertex * a,
            struct vertex * b )
```

@PARAM a: a vertex @PARAM b: another vertex @RETURN 0 for success; -1 for fail Removes the edge connecting a to b

References edge::a, edge::a_vars, edge::b, edge::b_vars, BI_EDGE, edge::bi_edge, edge::bi_edge_lock, vertex↩::context, EDGE, vertex::edge_tree, edge::edge_type, edge::f, edge::glbl, edge::id, vertex::id, vertex::joining_↩vertices, vertex::lock, remove_ID(), SINGLE, and topologic_debug.

Referenced by create_bi_edge(), create_edge(), procces_request(), and remove_bi_edge().

5.8.2.20 remove_edge_id()

```
int remove_edge_id (
            struct vertex * a,
            int id )
```

@PARAM a: a vertex @PARAM id: id of edge to remove @RETURN 0 for success; -1 for fail Removes the edge in a with that id

References edge::a, edge::a_vars, edge::b, edge::b_vars, BI_EDGE, edge::bi_edge, edge::bi_edge_lock, vertex↩::context, EDGE, vertex::edge_tree, edge::edge_type, edge::f, edge::glbl, edge::id, vertex::id, vertex::joining_↩vertices, vertex::lock, remove_ID(), SINGLE, and topologic_debug.

Referenced by procces_request(), and remove_vertex().

5.8.2.21 remove_vertex()

```
int remove_vertex (
            struct graph * graph,
            struct vertex * vertex )
```

@PARAM graph: the graph @PARAM vertex: a vertex @RETURN 0 for sucess; -1 for fail Removes the vertex and all connected edges

References edge::a, edge::b, BI_EDGE, edge::bi_edge, edge::bi_edge_lock, graph::context, vertex::context, destroy_stack(), EDGE, vertex::edge_tree, edge::edge_type, edge::f, edge::glbl, vertex::glbl, edge::id, vertex::id, init_stack(), vertex::joining_vertices, graph::lock, vertex::lock, pop(), remove_edge_id(), remove_ID(), SELF_EDGE, vertex::shared, SINGLE, stackify(), topologic_debug, shared_edge::vertex_data, and graph::vertices.

Referenced by destroy_graph_avl(), procces_request(), and remove_vertex_id().

5.8.2.22 remove_vertex_id()

```
int remove_vertex_id (
            struct graph * graph,
            int id )
```

@PARAM graph: the graph @PARAM id: the vertex id @RETURN 0 for sucess; -1 for fail Removes the vertex and all connected edges

References graph::context, find(), graph::lock, remove_vertex(), SINGLE, topologic_debug, and graph::vertices.

Referenced by procces_request().

5.8.2.23 resume_graph()

```
int resume_graph (
            struct graph * graph )
```

@PARAM graph: the graph @RETURN 0 for success -1 if it fails Resumes run

References graph::lock, graph::pause, and topologic_debug.

5.8.2.24 run()

```
int run (
            struct graph * graph,
            struct vertex_result ** vertex_args )
```

@PARAM graph: the graph @PARAM vertex_args: array of vertex arguments for f @RETURN 0 if run terminates normally -1 if it fails Attempts to run the graph else aborts.

References ABORT, fireable::args, BLACK, graph::black_fire, graph::black_locked, graph::black_vertex_count, fireable::color, graph::color_lock, graph::context, CONTINUE, fire_pthread(), fireable::graph, fireable::iloop, vertex↩ ::is_active, graph::lock, MAX_ATTEMPTS, graph::max_state_changes, graph::mem_option, graph::num_vertices, graph::pause, graph::pause_cond, pop(), graph::previous_color, PRINT, graph::print_flag, print_graph(), process↩ _requests(), RED, graph::red_fire, graph::red_locked, graph::red_vertex_count, run_single(), SINGLE, graph::start, graph::state, graph::state_count, TERMINATE, THREAD_ATTEMPT_SLEEP, topologic_debug, fireable::vertex, and WAIT.

5.8.2.25 start_set()

```
int start_set (
            struct graph * graph,
            int id[ ],
            int num_vertices )
```

@PARAM graph: the graph, @PARAM id: the ids of the vertices @PARAM num_vertices: number of vertices @RETURN -1 for fail if any vertex fails; 0 for success Creates multiple contexts that are ran in parallel Handle errors

Given vertx failed, so at this point, free the vertices and leave

References graph::context, find(), vertex::id, pop(), push(), SINGLE, graph::start, topologic_debug, and graph↩ ::vertices.

5.8.2.26 submit_request()

```
int submit_request (
             struct graph * graph,
             struct request * request )
```

@PARAM graph: the graph @PARAM request: the request to be processed @RETRUN -1 for fail; 0 for succes; Submits a request to be processed after all active nodes complete

References graph::context, DESTROY_BI_EDGE, DESTROY_EDGE, DESTROY_EDGE_BY_ID, DESTROY_↩VERTEX, DESTROY_VERTEX_BY_ID, graph::lock, graph::modify, push(), graph::remove_edges, graph::remove↩_vertices, request::request, SINGLE, and topologic_debug.

5.8.2.27 switch_vertex()

```
int switch_vertex (
             struct graph * graph,
             struct vertex * vertex,
             struct vertex_result * args,
             enum STATES color,
             int iloop )
```

@PARAM graph: the graph @PARAM vertex: The vertex in which just fire @PARAM args; The result of the vertex Upon call the switch function will compute the edge functions connected to the vertex @PARAM iloop: the number of times that vertex fired in succession @RETURNS 0 On success; the vertex connected to the successful edge will be fired; -1 on failure

References ABORT, fireable::args, fireable::color, CONTINUE, vertex_result::edge_argv, vertex_result::edge_size, fire_pthread(), fireable::graph, fireable::iloop, MAX_ATTEMPTS, graph::mem_option, THREAD_ATTEMPT_SLEEP, topologic_debug, fireable::vertex, vertex_result::vertex_argv, vertex_result::vertex_size, and WAIT.

Referenced by fire().

5.9 include/vertex.h File Reference

```
#include <signal.h>
#include <stdio.h>
#include "./stack.h"
#include "./AVL.h"
#include "./context.h"
#include "./graph.h"
```

Data Structures

- struct vertex_result
- union shared_edge
- struct vertex
- struct vertex_request
- struct mod_vertex_request
- struct mod_edge_vars_request
- struct destroy_vertex_request
- struct destroy_vertex_id_request

5.10 README.md File Reference

5.11 src/AVL.c File Reference

```
#include "../include/AVL.h"
```

Functions

- struct AVLTree ∗ init_avl ()
- int balance (struct AVLNode ∗node)
- int max_height (struct AVLNode ∗node)
- struct AVLNode ∗ right_rotate (struct AVLNode ∗node)
- struct AVLNode ∗ left_rotate (struct AVLNode ∗node)
- struct AVLNode ∗ create_node (void ∗data, int id)
- struct AVLNode ∗ insert_node (struct AVLNode ∗node, struct AVLNode ∗insert)
- int insert (struct AVLTree ∗tree, void ∗data, int id)
- void ∗ find_node (struct AVLNode ∗root, int id)
- void ∗ find (struct AVLTree ∗tree, int id)
- struct AVLNode ∗ minNode (struct AVLNode ∗node)
- struct AVLNode ∗ remove_node (struct AVLNode ∗root, int id, void ∗∗data)
- void ∗ remove_ID (struct AVLTree ∗tree, int id)
- void inorder_nodes (struct AVLNode ∗node, struct stack ∗stack)
- void inorder (struct AVLTree ∗tree, struct stack ∗stack)
- void preorder_nodes (struct AVLNode ∗node, struct stack ∗stack)
- void preorder (struct AVLTree ∗tree, struct stack ∗stack)
- void postorder_nodes (struct AVLNode ∗node, struct stack ∗stack)
- void postorder (struct AVLTree ∗tree, struct stack ∗stack)
- void stackify_nodes (struct AVLNode ∗node, struct stack ∗stack)
- void stackify (struct AVLTree ∗tree, struct stack ∗stack)
- void destroy_avl_nodes (struct AVLNode ∗node)
- void destroy_avl (struct AVLTree ∗tree)

5.11.1 Function Documentation

5.11.1.1 balance()

```
int balance (
            struct AVLNode ∗ node )
```

References AVLNode::height, AVLNode::left, and AVLNode::right.

Referenced by insert_node(), and remove_node().

5.11.1.2 create_node()

```
struct AVLNode* create_node (
             void * data,
             int id )
```

References AVLNode::data, AVLNode::height, AVLNode::id, AVLNode::left, and AVLNode::right.

Referenced by insert().

5.11.1.3 destroy_avl()

```
void destroy_avl (
             struct AVLTree * tree )
```

References destroy_avl_nodes(), AVLTree::root, and AVLTree::size.

Referenced by create_vertex(), destroy_graph_avl(), and graph_init().

5.11.1.4 destroy_avl_nodes()

```
void destroy_avl_nodes (
             struct AVLNode * node )
```

References AVLNode::data, AVLNode::height, AVLNode::left, and AVLNode::right.

Referenced by destroy_avl().

5.11.1.5 find()

```
void* find (
             struct AVLTree * tree,
             int id )
```

References find_node(), and AVLTree::root.

Referenced by create_edge(), modify_edge(), remove_vertex_id(), and start_set().

5.11.1.6 find_node()

```
void* find_node (
            struct AVLNode * root,
            int id )
```

References AVLNode::data, AVLNode::id, AVLNode::left, and AVLNode::right.

Referenced by find().

5.11.1.7 init_avl()

```
struct AVLTree* init_avl ( )
```

References AVLTree::root, and AVLTree::size.

Referenced by create_vertex(), and graph_init().

5.11.1.8 inorder()

```
void inorder (
            struct AVLTree * tree,
            struct stack * stack )
```

References inorder_nodes(), AVLTree::root, and AVLTree::size.

Referenced by print_state().

5.11.1.9 inorder_nodes()

```
void inorder_nodes (
            struct AVLNode * node,
            struct stack * stack )
```

References AVLNode::data, AVLNode::id, AVLNode::left, push(), and AVLNode::right.

Referenced by inorder().

5.11.1.10 insert()

```
int insert (
            struct AVLTree * tree,
            void * data,
            int id )
```

References create_node(), AVLNode::data, insert_node(), AVLTree::root, and AVLTree::size.

Referenced by create_edge(), create_vertex(), and insert_node().

5.11.1.11 insert_node()

```
struct AVLNode* insert_node (
            struct AVLNode * node,
            struct AVLNode * insert )
```

References balance(), AVLNode::height, AVLNode::id, insert(), AVLNode::left, left_rotate(), max_height(), AVLNode::right, and right_rotate().

Referenced by insert().

5.11.1.12 left_rotate()

```
struct AVLNode* left_rotate (
            struct AVLNode * node )
```

References AVLNode::height, AVLNode::left, max_height(), and AVLNode::right.

Referenced by insert_node(), and remove_node().

5.11.1.13 max_height()

```
int max_height (
            struct AVLNode * node )
```

References AVLNode::height, AVLNode::left, and AVLNode::right.

Referenced by insert_node(), left_rotate(), remove_node(), and right_rotate().

5.11.1.14 minNode()

```
struct AVLNode* minNode (
            struct AVLNode * node )
```

References AVLNode::left.

Referenced by remove_node().

5.11.1.15 postorder()

```
void postorder (
            struct AVLTree * tree,
            struct stack * stack )
```

References postorder_nodes(), AVLTree::root, and AVLTree::size.

5.11.1.16 postorder_nodes()

```
void postorder_nodes (
            struct AVLNode * node,
            struct stack * stack )
```

References AVLNode::data, AVLNode::id, AVLNode::left, push(), and AVLNode::right.

Referenced by postorder().

5.11.1.17 preorder()

```
void preorder (
            struct AVLTree * tree,
            struct stack * stack )
```

References preorder_nodes(), AVLTree::root, and AVLTree::size.

Referenced by destroy_graph_avl(), fire(), print_edges(), and run_single().

5.11.1.18 preorder_nodes()

```
void preorder_nodes (
            struct AVLNode * node,
            struct stack * stack )
```

References AVLNode::data, AVLNode::id, AVLNode::left, push(), and AVLNode::right.

Referenced by preorder().

5.11.1.19 remove_ID()

```
void* remove_ID (
            struct AVLTree * tree,
            int id )
```

References AVLNode::data, remove_node(), AVLTree::root, and AVLTree::size.

Referenced by create_edge(), remove_edge(), remove_edge_id(), and remove_vertex().

5.11.1.20 remove_node()

```
struct AVLNode* remove_node (
            struct AVLNode * root,
            int id,
            void ** data )
```

References balance(), AVLNode::data, AVLNode::height, AVLNode::id, AVLNode::left, left_rotate(), max_height(), minNode(), AVLNode::right, and right_rotate().

Referenced by remove_ID().

5.11.1.21 right_rotate()

```
struct AVLNode* right_rotate (
            struct AVLNode * node )
```

References AVLNode::height, AVLNode::left, max_height(), and AVLNode::right.

Referenced by insert_node(), and remove_node().

5.11.1.22 stackify()

```
void stackify (
            struct AVLTree * tree,
            struct stack * stack )
```

References AVLTree::root, AVLTree::size, and stackify_nodes().

Referenced by remove_vertex().

5.11.1.23 stackify_nodes()

```
void stackify_nodes (
            struct AVLNode * node,
            struct stack * stack )
```

References AVLNode::data, AVLNode::height, AVLNode::left, push(), and AVLNode::right.

Referenced by stackify().

5.12 src/edge.c File Reference

```
#include "../include/topologic.h"
```

Functions

- struct edge ∗ create_edge (struct vertex ∗a, struct vertex ∗b, int(∗f)(void ∗, void ∗, const void ∗const, const void ∗const), void ∗glbl)
- int create_bi_edge (struct vertex ∗a, struct vertex ∗b, int(∗f)(void ∗, void ∗, const void ∗const, const void ∗const), void ∗glbl, struct edge ∗∗edge_a_to_b, struct edge ∗∗edge_b_to_a)
- int remove_edge (struct vertex ∗a, struct vertex ∗b)
- int remove_edge_id (struct vertex ∗a, int id)
- int remove_bi_edge (struct vertex ∗a, struct vertex ∗b)
- int modify_edge (struct vertex ∗a, struct vertex ∗b, int(∗f)(void ∗, void ∗, const void ∗const, const void ∗const), void ∗glbl)
- int modify_bi_edge (struct vertex ∗a, struct vertex ∗b, int(∗f)(void ∗, void ∗, const void ∗const, const void ∗const), void ∗glbl)

5.12.1 Function Documentation

5.12.1.1 create_bi_edge()

```
int create_bi_edge (
            struct vertex * a,
            struct vertex * b,
            int(*)(void *, void *, const void *const, const void *const) f,
            void * glbl,
            struct edge ** edge_a_to_b,
            struct edge ** edge_b_to_a )
```

@RETURNS 0 for success; -1 for fail See create_edge Will create an bidirectional edge between vertex a and b with some criteria determined by the function f. Will store the edges in edge_a and edge_b. If edge_a_to_b or edge_b_to_a is NULL it will not.

References edge::a, edge::b, BI_EDGE, edge::bi_edge, edge::bi_edge_lock, vertex::context, create_edge(), edge::edge_type, edge::f, edge::glbl, remove_edge(), SINGLE, and topologic_debug.

Referenced by procces_request().

5.12.1.2 create_edge()

```
struct edge* create_edge (
             struct vertex * a,
             struct vertex * b,
             int(*)(void *, void *, const void *const, const void *const) f,
             void * glbl )
```

@PARAM a: A vertex @PARAM b: Another vertex (can be 'a') @PARAM f: a function @PARAM glbl: global variables @RETURN the edge connecting a to b Will create an edge from vertex a to b with some criteria determined by the function f. NOTE: NULL glbl will mean no global variables. f cannot be NULL.

References edge::a, edge::a_vars, edge::b, edge::b_vars, edge::bi_edge, vertex::context, EDGE, shared_edge↩ ::edge_data, vertex::edge_tree, edge::edge_type, edge::f, find(), edge::glbl, edge::id, vertex::id, insert(), vertex↩ ::joining_vertices, vertex::lock, remove_edge(), remove_ID(), SELF_EDGE, vertex::shared, SINGLE, SWITCH, and topologic_debug.

Referenced by create_bi_edge(), and procces_request().

5.12.1.3 modify_bi_edge()

```
int modify_bi_edge (
             struct vertex * a,
             struct vertex * b,
             int(*)(void *, void *, const void *const, const void *const) f,
             void * glbl )
```

@PARAM a: a vertex @PARAM b: another vertex @PARAM f: a function @PARAM glbl: global variables @↩ RETURN 0 for success; -1 for fail; -2 if only edge from a to b is modified; -3 if only edge from b to a is modified Modifies the edge between a and b's function and variables NOTE: NULL f, or glbl will mean no change.

References edge::a, edge::b, edge::f, edge::glbl, modify_edge(), and topologic_debug.

Referenced by procces_request().

5.12.1.4 modify_edge()

```
int modify_edge (
             struct vertex * a,
             struct vertex * b,
             int(*)(void *, void *, const void *const, const void *const) f,
             void * glbl )
```

@PARAM a: a vertex @PARAM b: another vertex @PARAM f: a function @PARAM glbl: global variables @↩ RETURN 0 for success; -1 for fail Modifies the edge connecting a to b's function NOTE: NULL f, or glbl will mean no change.

References edge::a, edge::b, vertex::context, vertex::edge_tree, edge::f, find(), edge::glbl, vertex::id, vertex::lock, SINGLE, and topologic_debug.

Referenced by modify_bi_edge(), and procces_request().

5.12.1.5 remove_bi_edge()

```
int remove_bi_edge (
            struct vertex * a,
            struct vertex * b )
```

@PARAM a: a vertex @PARAM b: another vertex @RETURN 0 for success; -1 for fail; -2 if only edge from a to b is removed; -3 if only edge from b to a is removed Removes the edge connecting a to b

References edge::a, edge::b, remove_edge(), and topologic_debug.

Referenced by procces_request().

5.12.1.6 remove_edge()

```
int remove_edge (
            struct vertex * a,
            struct vertex * b )
```

@PARAM a: a vertex @PARAM b: another vertex @RETURN 0 for success; -1 for fail Removes the edge connecting a to b

References edge::a, edge::a_vars, edge::b, edge::b_vars, BI_EDGE, edge::bi_edge, edge::bi_edge_lock, vertex↩ ::context, EDGE, vertex::edge_tree, edge::edge_type, edge::f, edge::glbl, edge::id, vertex::id, vertex::joining_↩ vertices, vertex::lock, remove_ID(), SINGLE, and topologic_debug.

Referenced by create_bi_edge(), create_edge(), procces_request(), and remove_bi_edge().

5.12.1.7 remove_edge_id()

```
int remove_edge_id (
            struct vertex * a,
            int id )
```

@PARAM a: a vertex @PARAM id: id of edge to remove @RETURN 0 for success; -1 for fail Removes the edge in a with that id

References edge::a, edge::a_vars, edge::b, edge::b_vars, BI_EDGE, edge::bi_edge, edge::bi_edge_lock, vertex↩ ::context, EDGE, vertex::edge_tree, edge::edge_type, edge::f, edge::glbl, edge::id, vertex::id, vertex::joining_↩ vertices, vertex::lock, remove_ID(), SINGLE, and topologic_debug.

Referenced by procces_request(), and remove_vertex().

5.13 src/graph.c File Reference

```
#include "../include/topologic.h"
```

Functions

- struct graph ∗ graph_init (int max_state_changes, int snapshot_timestamp, int max_loop, unsigned int lvl_←
verbose, enum CONTEXT context, enum MEM_OPTION mem_option)
- void destroy_graph_stack (struct stack ∗stack)
- void destroy_graph_avl (struct graph ∗graph, struct AVLTree ∗tree)
- int destroy_graph (struct graph ∗graph)

5.13.1 Function Documentation

5.13.1.1 destroy_graph()

```
int destroy_graph (
            struct graph * graph )
```

@PARAM graph: the graph @RETURN -1 for fail; 0 for success Destroys and frees the graph

References graph::black_fire, graph::black_locked, graph::black_vertex_count, graph::color_lock, graph::context, destroy_graph_avl(), destroy_graph_stack(), graph::lock, graph::modify, graph::pause_cond, graph::red_fire, graph::red_locked, graph::red_vertex_count, graph::remove_edges, graph::remove_vertices, SINGLE, graph←
::start, graph::state, TERMINATE, topologic_debug, and graph::vertices.

5.13.1.2 destroy_graph_avl()

```
void destroy_graph_avl (
            struct graph * graph,
            struct AVLTree * tree )
```

References destroy_avl(), destroy_stack(), init_stack(), pop(), preorder(), and remove_vertex().

Referenced by destroy_graph().

5.13.1.3 destroy_graph_stack()

```
void destroy_graph_stack (
            struct stack * stack )
```

References destroy_request(), destroy_stack(), and pop().

Referenced by destroy_graph().

5.13.1.4 graph_init()

```
struct graph* graph_init (
            int max_state_changes,
            int snapshot_timestamp,
            int max_loop,
            unsigned int lvl_verbose,
            enum CONTEXT context,
            enum MEM_OPTION mem_option )
```

@PARAM max_state_changes: # state changes before entering sink vertex due to infinite loop of states; -1 to ignore @PARAM snapshot_timestamp: printing out data at given timestamp for user; -1 for none; 0 for first and last state @PARAM lvl_verbose: how verbose timestamp print is @PARAM context: linear or context-switch based @RETURN an empty graph Creates a graph structures

References graph::black_fire, graph::black_locked, graph::black_vertex_count, graph::color_lock, graph::context, destroy_avl(), destroy_stack(), init_avl(), init_stack(), graph::lock, graph::lvl_verbose, graph::max_loop, graph↩ ::max_state_changes, graph::mem_option, graph::modify, NONE, graph::num_vertices, graph::pause, graph↩ ::pause_cond, graph::previous_color, graph::print_flag, RED, graph::red_fire, graph::red_locked, graph::red_↩ vertex_count, graph::remove_edges, graph::remove_vertices, SINGLE, graph::snapshot_timestamp, graph::start, graph::state, graph::state_count, topologic_debug, and graph::vertices.

5.14 src/request.c File Reference

```
#include "../include/topologic.h"
```

Functions

- struct request ∗ create_request (enum REQUESTS request, void ∗args, void(∗f)(void ∗))
- int submit_request (struct graph ∗graph, struct request ∗request)
- int procces_request (struct request ∗request)
- int process_requests (struct graph ∗graph)
- int destroy_request (struct request ∗request)

5.14.1 Function Documentation

5.14.1.1 create_request()

```
struct request* create_request (
            enum REQUESTS request,
            void * args,
            void(*)(void *) f )
```

@PARAM request: the desired request @PARAM args: the arguments needed for f @PARAM f: the function of the request @RETURN the request or NULL if it fails Creates a request structure to be called later

References request::args, CREAT_BI_EDGE, CREAT_EDGE, CREAT_VERTEX, DESTROY_BI_EDGE, DESTROY_EDGE, DESTROY_EDGE_BY_ID, DESTROY_VERTEX, DESTROY_VERTEX_BY_ID, request::f, GENERIC, MOD_BI_EDGE, MOD_EDGE, MOD_EDGE_VARS, MOD_VERTEX, request::request, and topologic↩ _debug.

5.14.1.2 destroy_request()

```
int destroy_request (
            struct request * request )
```

@PARAM request: a request @RETURN -1 for fail; 0 for success Destroys and frees a request

References request::args, request::f, request::request, and topologic_debug.

Referenced by destroy_graph_stack().

5.14.1.3 procces_request()

```
int procces_request (
            struct request * request )
```

References edge_request::a, destroy_edge_request::a, destroy_edge_id_request::a, request::args, edge_↩
request::b, destroy_edge_request::b, CREAT_BI_EDGE, CREAT_EDGE, CREAT_VERTEX, create_bi_edge(),
create_edge(), create_vertex(), DESTROY_BI_EDGE, DESTROY_EDGE, DESTROY_EDGE_BY_ID, DESTROY↩
_VERTEX, DESTROY_VERTEX_BY_ID, mod_edge_vars_request::edge_vars, vertex_request::f, mod_vertex↩
_request::f, request::f, edge_request::f, GENERIC, edge_request::glbl, vertex_request::glbl, mod_vertex_↩
request::glbl, vertex_request::graph, destroy_vertex_request::graph, destroy_vertex_id_request::graph, destroy↩
_edge_id_request::id, vertex_request::id, destroy_vertex_id_request::id, MOD_BI_EDGE, MOD_EDGE, MOD↩
_EDGE_VARS, MOD_VERTEX, modify_bi_edge(), modify_edge(), modify_shared_edge_vars(), modify_vertex(),
remove_bi_edge(), remove_edge(), remove_edge_id(), remove_vertex(), remove_vertex_id(), request::request,
topologic_debug, mod_vertex_request::vertex, mod_edge_vars_request::vertex, and destroy_vertex_request↩
::vertex.

Referenced by process_requests().

5.14.1.4 process_requests()

```
int process_requests (
            struct graph * graph )
```

@PARAM graph: the graph @RETURN 0 if all got processed; -1 if a request failed will set ERRNO to the ENUM Process requests that are queued in the graph

References graph::context, graph::lock, graph::modify, pop(), procces_request(), graph::remove_edges, graph↩
::remove_vertices, SINGLE, and topologic_debug.

Referenced by run(), and run_single().

5.14.1.5 submit_request()

```
int submit_request (
            struct graph * graph,
            struct request * request )
```

@PARAM graph: the graph @PARAM request: the request to be processed @RETRUN -1 for fail; 0 for succes; Submits a request to be processed after all active nodes complete

References graph::context, DESTROY_BI_EDGE, DESTROY_EDGE, DESTROY_EDGE_BY_ID, DESTROY_↩
VERTEX, DESTROY_VERTEX_BY_ID, graph::lock, graph::modify, push(), graph::remove_edges, graph::remove↩
_vertices, request::request, SINGLE, and topologic_debug.

5.15 src/stack.c File Reference

```
#include "../include/stack.h"
```

Functions

- struct stack ∗ init_stack ()
- void ∗ pop (struct stack ∗stack)
- void ∗ get (struct stack ∗stack, int index)
- int push (struct stack ∗stack, void ∗data)
- void destroy_stack (struct stack ∗stack)

5.15.1 Function Documentation

5.15.1.1 destroy_stack()

```
void destroy_stack (
            struct stack * stack )
```

References stack_node::data, stack::length, stack_node::next, and stack::root.

Referenced by destroy_graph_avl(), destroy_graph_stack(), fire(), graph_init(), print_edges(), print_state(), remove_vertex(), and run_single().

5.15.1.2 get()

```
void* get (
            struct stack * stack,
            int index )
```

References stack_node::data, stack::length, stack_node::next, and stack::root.

5.15.1.3 init_stack()

```
struct stack* init_stack ( )
```

References stack::length, and stack::root.

Referenced by destroy_graph_avl(), fire(), graph_init(), print_edges(), print_state(), remove_vertex(), and run_↩ single().

5.15.1.4 pop()

```
void* pop (
            struct stack * stack )
```

References stack_node::data, stack::length, stack_node::next, and stack::root.

Referenced by destroy_graph_avl(), destroy_graph_stack(), fire(), print_edges(), print_state(), process_requests(), remove_vertex(), run(), run_single(), and start_set().

5.15.1.5 push()

```
int push (
            struct stack * stack,
            void * data )
```

References stack_node::data, stack::length, stack_node::next, and stack::root.

Referenced by inorder_nodes(), postorder_nodes(), preorder_nodes(), stackify_nodes(), start_set(), and submit_↩ request().

5.16 src/topologic.c File Reference

```
#include "../include/topologic.h"
```

Functions

- void sleep_ms (int milliseconds)
- int start_set (struct graph ∗graph, int id[ ], int num_vertices)
- int run_single (struct graph ∗graph, struct vertex_result ∗∗init_vertex_args)
- int run (struct graph ∗graph, struct vertex_result ∗∗init_vertex_args)
- int fire (struct graph ∗graph, struct vertex ∗vertex, struct vertex_result ∗args, enum STATES color, int iloop)
- void ∗ fire_pthread (void ∗vargp)
- int switch_vertex (struct graph ∗graph, struct vertex ∗vertex, struct vertex_result ∗args, enum STATES color, int iloop)
- int pause_graph (struct graph ∗graph)
- int resume_graph (struct graph ∗graph)

5.16.1 Function Documentation

5.16.1.1 fire()

```
int fire (
            struct graph * graph,
            struct vertex * vertex,
            struct vertex_result * args,
            enum STATES color,
            int iloop )
```

@PARAM graph: the graph @PARAM vertex: A vertex to be ran @PARAM args: arguments @PARAM color: the state in which unlocks fire process if STATE is set to PRINT then fire will fail @PARAM iloop: the number of times that vertex fired in succession @RETURNS the result of the vertex fire will wake up the vertex and pass args to the vertex to compute its function and then call switch and clean itself up

References edge::a_vars, fireable::args, edge::b, edge::b_vars, BI_EDGE, edge::bi_edge_lock, BLACK, graph↩::black_fire, graph::black_locked, graph::black_vertex_count, fireable::color, graph::context, destroy_stack(), vertex_result::edge_argv, vertex::edge_tree, edge::edge_type, vertex::f, edge::f, fire(), edge::glbl, vertex::glbl, fireable::iloop, init_stack(), vertex::is_active, graph::lock, vertex::lock, graph::max_loop, graph::max_state_changes, NONE, graph::num_vertices, pop(), preorder(), PTHREAD_SLEEP_TIME, RED, graph::red_fire, graph::red_locked, graph::red_vertex_count, vertex::shared, sleep_ms(), graph::state, graph::state_count, SWITCH, SWITCH_↩UNSAFE, switch_vertex(), TERMINATE, topologic_debug, vertex_result::vertex_argv, and shared_edge::vertex_↩data.

Referenced by fire(), and fire_pthread().

5.16.1.2 fire_pthread()

```
void* fire_pthread (
            void * vargp )
```

Wrapper function for fire, fire_1 @PARAM vargp: arguments

References fireable::args, fireable::color, fire(), fireable::graph, fireable::iloop, PTHREAD_SLEEP_TIME, sleep_↩ms(), topologic_debug, and fireable::vertex.

Referenced by run(), and switch_vertex().

5.16.1.3 pause_graph()

```
int pause_graph (
            struct graph * graph )
```

@PARAM graph: the graph @RETURN 0 for success -1 if it fails Pauses run

References graph::lock, graph::pause, graph::pause_cond, and topologic_debug.

5.16.1.4 resume_graph()

```
int resume_graph (
            struct graph * graph )
```

@PARAM graph: the graph @RETURN 0 for success -1 if it fails Resumes run

References graph::lock, graph::pause, and topologic_debug.

5.16.1.5 run()

```
int run (
            struct graph * graph,
            struct vertex_result ** vertex_args )
```

@PARAM graph: the graph @PARAM vertex_args: array of vertex arguments for f @RETURN 0 if run terminates normally -1 if it fails Attempts to run the graph else aborts.

References ABORT, fireable::args, BLACK, graph::black_fire, graph::black_locked, graph::black_vertex_count, fireable::color, graph::color_lock, graph::context, CONTINUE, fire_pthread(), fireable::graph, fireable::iloop, vertex↩ ::is_active, graph::lock, MAX_ATTEMPTS, graph::max_state_changes, graph::mem_option, graph::num_vertices, graph::pause, graph::pause_cond, pop(), graph::previous_color, PRINT, graph::print_flag, print_graph(), process↩ _requests(), RED, graph::red_fire, graph::red_locked, graph::red_vertex_count, run_single(), SINGLE, graph::start, graph::state, graph::state_count, TERMINATE, THREAD_ATTEMPT_SLEEP, topologic_debug, fireable::vertex, and WAIT.

5.16.1.6 run_single()

```
int run_single (
            struct graph * graph,
            struct vertex_result ** init_vertex_args )
```

References edge::a_vars, edge::b, edge::b_vars, graph::context, destroy_stack(), vertex_result::edge_argv, vertex::edge_tree, vertex::f, edge::f, edge::glbl, vertex::glbl, init_stack(), vertex::is_active, stack::length, graph::lock, graph::max_loop, graph::max_state_changes, graph::num_vertices, graph::pause, graph::pause_cond, pop(), pre-order(), print_graph(), process_requests(), vertex::shared, SINGLE, graph::start, graph::state, graph::state_count, TERMINATE, topologic_debug, vertex_result::vertex_argv, and shared_edge::vertex_data.

Referenced by run().

5.16.1.7 sleep_ms()

```
void sleep_ms (
            int milliseconds )
```

Referenced by fire(), and fire_pthread().

5.16.1.8 start_set()

```
int start_set (
            struct graph * graph,
            int id[ ],
            int num_vertices )
```

@PARAM graph: the graph, @PARAM id: the ids of the vertices @PARAM num_vertices: number of vertices @RETURN -1 for fail if any vertex fails; 0 for success Creates multiple contexts that are ran in parallel Handle errors

Given vertx failed, so at this point, free the vertices and leave

References graph::context, find(), vertex::id, pop(), push(), SINGLE, graph::start, topologic_debug, and graph↩ ::vertices.

5.16.1.9 switch_vertex()

```
int switch_vertex (
            struct graph * graph,
            struct vertex * vertex,
            struct vertex_result * args,
            enum STATES color,
            int iloop )
```

@PARAM graph: the graph @PARAM vertex: The vertex in which just fire @PARAM args; The result of the vertex Upon call the switch function will compute the edge functions connected to the vertex @PARAM iloop: the number of times that vertex fired in succession @RETURNS 0 On success; the vertex connected to the successful edge will be fired; -1 on failure

References ABORT, fireable::args, fireable::color, CONTINUE, vertex_result::edge_argv, vertex_result::edge_size, fire_pthread(), fireable::graph, fireable::iloop, MAX_ATTEMPTS, graph::mem_option, THREAD_ATTEMPT_SLEEP, topologic_debug, fireable::vertex, vertex_result::vertex_argv, vertex_result::vertex_size, and WAIT.

Referenced by fire().

5.17 src/topologic_json.c File Reference

```
#include "../include/topologic.h"
```

Functions

- void print_edges (struct graph ∗graph, struct AVLTree ∗edges, const char ∗indent, FILE ∗out)
- void print_state (struct graph ∗graph, FILE ∗out)
- void print_graph (struct graph ∗graph)

5.17.1 Function Documentation

5.17.1.1 print_edges()

```
void print_edges (
             struct graph * graph,
             struct AVLTree * edges,
             const char * indent,
             FILE * out )
```

References edge::a, edge::b, edge::bi_edge, destroy_stack(), edge::edge_type, edge::f, FUNCTIONS, edge::glbl, GLOBALS, edge::id, init_stack(), stack::length, graph::lvl_verbose, pop(), preorder(), and topologic_debug.

Referenced by print_state().

5.17.1.2 print_graph()

```
void print_graph (
             struct graph * graph )
```

@PARAM graph: the graph Prints the graph with desired output TODO: Print enums

References graph::context, graph::lock, graph::lvl_verbose, graph::max_loop, graph::max_state_changes, NO↩ _VERB, graph::num_vertices, print_state(), SINGLE, AVLTree::size, graph::snapshot_timestamp, START_STOP, graph::state_count, topologic_debug, and graph::vertices.

Referenced by run(), and run_single().

5.17.1.3 print_state()

```
void print_state (
             struct graph * graph,
             FILE * out )
```

References destroy_stack(), vertex::edge_tree, EDGES, vertex::f, FUNCTIONS, edge::glbl, vertex::glbl, GLOBALS, vertex::id, init_stack(), inorder(), vertex::is_active, stack::length, graph::lvl_verbose, pop(), print_edges(), vertex↩ ::shared, AVLTree::size, topologic_debug, VERTICES, and graph::vertices.

Referenced by print_graph().

5.18 src/vertex.c File Reference

```
#include "../include/topologic.h"
```

Functions

- struct vertex ∗ create_vertex (struct graph ∗graph, void(∗f)(struct graph ∗, struct vertex_result ∗, void ∗, void ∗), int id, void ∗glbl)
- int remove_vertex (struct graph ∗graph, struct vertex ∗vertex)
- int remove_vertex_id (struct graph ∗graph, int id)
- int modify_vertex (struct vertex ∗vertex, void(∗f)(struct graph ∗, struct vertex_result ∗, void ∗, void ∗), void ∗glbl)
- int modify_shared_edge_vars (struct vertex ∗vertex, void ∗edge_vars)

5.18.1 Function Documentation

5.18.1.1 create_vertex()

```
struct vertex* create_vertex (
            struct graph * graph,
            void(*)(struct graph *, struct vertex_result *, void *, void *) f,
            int id,
            void * glbl )
```

@PARAM graph: the graph @PARAM f: a function @PARAM id: vertex id to be used: MUST BE UNIQUE @PARAM glbl: global variables @RETURN vertex: a vertex to be used in a graph On creation a process will be spawned for the vertex The vertex will compute function f when called NOTE: NULL glbl will mean no global variables. f cannot be NULL.

References graph::context, vertex::context, destroy_avl(), vertex::edge_tree, vertex::f, vertex::glbl, vertex::id, init_avl(), insert(), vertex::is_active, vertex::joining_vertices, graph::lock, vertex::lock, vertex::shared, SINGLE, topologic_debug, shared_edge::vertex_data, and graph::vertices.

Referenced by procces_request().

5.18.1.2 modify_shared_edge_vars()

```
int modify_shared_edge_vars (
            struct vertex * vertex,
            void * edge_vars )
```

@PARAM vertex: a vertex @PARAM edge_vars: shared variables @RETURN 0 for success; -1 for fail Modifies the vertices shared variables with it's edges

References vertex::context, vertex::lock, vertex::shared, SINGLE, topologic_debug, and shared_edge::vertex_data.

Referenced by procces_request().

5.18.1.3 modify_vertex()

```
int modify_vertex (
              struct vertex * vertex,
              void(*)(struct graph *, struct vertex_result *, void *, void *) f,
              void * glbl )
```

@PARAM vertex: a vertex @PARAM f: a function @PARAM glbl: global variables NOTE: NULL f, or glbl will mean no change. @RETURN 0 for success; -1 for fail Modifies the vertices function

References vertex::context, vertex::f, vertex::glbl, vertex::lock, SINGLE, and topologic_debug.

Referenced by procces_request().

5.18.1.4 remove_vertex()

```
int remove_vertex (
              struct graph * graph,
              struct vertex * vertex )
```

@PARAM graph: the graph @PARAM vertex: a vertex @RETURN 0 for sucess; -1 for fail Removes the vertex and all connected edges

References edge::a, edge::b, BI_EDGE, edge::bi_edge, edge::bi_edge_lock, graph::context, vertex::context, destroy_stack(), EDGE, vertex::edge_tree, edge::edge_type, edge::f, edge::glbl, vertex::glbl, edge::id, vertex::id, init_stack(), vertex::joining_vertices, graph::lock, vertex::lock, pop(), remove_edge_id(), remove_ID(), SELF_EDGE, vertex::shared, SINGLE, stackify(), topologic_debug, shared_edge::vertex_data, and graph::vertices.

Referenced by destroy_graph_avl(), procces_request(), and remove_vertex_id().

5.18.1.5 remove_vertex_id()

```
int remove_vertex_id (
              struct graph * graph,
              int id )
```

@PARAM graph: the graph @PARAM id: the vertex id @RETURN 0 for sucess; -1 for fail Removes the vertex and all connected edges

References graph::context, find(), graph::lock, remove_vertex(), SINGLE, topologic_debug, and graph::vertices.

Referenced by procces_request().

Index