Topologic Design Document

Matthew Stern, Benjamin Michalowicz

May-June 2020

Contents

1 Introduction

The goal of this library is to simulate DFAs and Probabilistic DFAs using context switching. DFAs will be simulated using vertices and edges. vertices are defined by created threads that compute some function. Edges are defined by

weights, functions, or probabilities in that the program can traverse from one vertex to another given based on a criteria. A program may start at multiple vertices and each of those vertices may be ran in parallel. Each of those vertices will be computed and then their edges will be tested to be check for acceptance. For each acceptable edge (or first acceptable if preferred) the program will switch context to the connected vertex's process. Should there be no acceptable edge the program will terminate. Should multiple vertices accept the same vertex by some edges, the vertex will be called multiple times in a queue with the proper results of the edges. vertices will be able to share information via edges and edges may pass additional information to vertices should the vertex desire it.
Output of this program can include a full traverse history with or without computations. The program can also create a snapshot output after completing some number of steps or whenever desired.
The library will build a graph of vertices and edges that will handle the connections.

2 Files

1. include/
    1.1. AVL.h
    1.2. context.h
    1.3. edge.h
    1.4. fireable.h
    1.5. graph.h
    1.6. header.h
    1.7. request.h
    1.8. stack.h
    1.9. test.h
    1.10. topologic.h
    1.11. topylogic.h
    1.12. vertex.h
    1.13. windows_wrap.h
2. parse/
    2.1. topological_parser.lex
    2.2. topological_parser.y
    2.3. topological_parser_cpp.lex
    2.4. topological_parser_cpp.ypp
3. src/

3.1. AVL.c
3.2. edge.c
3.3. fireable.c
3.4. graph.c
3.5. request.c
3.6. stack.c
3.7. topologic.c
3.8. topologic_json.c
3.9. vertex.c
3.10. windows_wrap.c

4. testing/
   4.1. avl_test.c
   4.2. graph_test.json
   4.3. graph_vertex_edge_test.c
   4.4. multi_edge.c
   4.5. none_context_test.c
   4.6. request_test.c
   4.7. self_edge.c
   4.8. single_context_test.c
   4.9. stack_test.c
   4.10. switch_context_test.c
5. .gitattributes
6. .gitignore
7. AUTHORS
8. LICENSE
9. Makefile
10. README.md
11. SECURITY.md

3 Functions

3.1 Graph/Vertex/Edge Creation

```
/**
 Wrapper function for fire, fire_1
@PARAM vargp: arguments
**/
void *fire_pthread(void *vargp);

/**
@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
**/
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,
                         enum REQUEST_FLAG request_flag);

/**
@PARAM graph: the graph
@PARAM f: a function with the parameters
          graph: the desired graph of the vertex
          vertex_result: the vertices and edges results
                         that get passed from one vertex/edge
                         to the next
         glbl: the vertices global vars
         edge_vars: the variables shared with the vertices edges
@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 thread 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.
**/
```

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

/**
@PARAM a: A vertex
@PARAM b: Another vertex (can be 'a')
@PARAM f: a function that takes two paramters:
          the vertex_result's edge_argv and the shared varaibles between
          the edge and the vertex
@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.
**/
struct edge *create_edge(struct vertex *a,
                         struct vertex *b,
                         int (*f)(int, void *, const void *const),
                         void *glbl);

/**
@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.
**/
int create_bi_edge(struct vertex *a,
                   struct vertex *b,
                   int (*f)(int, void *, const void *const),
                   void *glbl,
                   struct edge **edge_a_to_b,
                   struct edge **edge_b_to_a);

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

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

/**
@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
**/
int remove_edge_id(struct vertex *a,
                   int id);
/**
@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
**/
int remove_bi_edge(struct vertex *a,
                   struct vertex *b);

/**
@PARAM graph: the graph
@PARAM vertex: a vertex
@RETURN 0 for sucess;
        -1 for fail
Removes the vertex and all connected edges
**/
int remove_vertex(struct graph *graph,
                  struct vertex *vertex);

/**
@PARAM graph: the graph
@PARAM id: the vertex id
@RETURN 0 for sucess;
        -1 for fail
Removes the vertex and all connected edges
**/
int remove_vertex_id(struct graph *graph,
                     int id);

/**
@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
**/
int modify_vertex(struct vertex *vertex,
                  void (*f)(int, struct graph *, struct vertex_result *,
                            void *, void *),
                  void *glbl);

/**
@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
**/
int modify_shared_edge_vars(int, struct vertex *vertex,
                            void *edge_vars);

/**
@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.
**/
int modify_edge(struct vertex *a,
                struct vertex *b,
                int (*f)(int, void *, const void *const),
                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.
**/
int modify_bi_edge(struct vertex *a,
                   struct vertex *b,
                   int (*f)(int, void *, const void *const),
                   void *glbl);

/**
@PARAM graph: the graph
@PARAM vertex: A vertex to be ran
@PARAM args: arguments
@PARAM color: the state in which unlocks the firing thread
              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
**/
```

3.2 Running the program (Fire, Switch, Starting Set

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

/**
@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
**/
int start_set(struct graph *graph,
              int id[],
              int num_vertices);

/**
@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.
**/
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 when context=SINGLE
else aborts.
**/
int run_single(struct graph *graph,
        struct vertex_result **vertex_args);

/**
@PARAM graph: the graph
@PARAM vertex: a vertex in the graph
@PARAM vertex_result: the vertex_results to copy
@PARAM color: next vertices color
@PARAM iloop: the count of the number of times the vertex has been called
Create a fireable structure
**/
struct fireable *create_fireable(struct graph *graph,
                                 struct vertex *vertex,
                                 struct vertex_result *args,
                                 enum STATES color,
                                 int iloop);

/**
@PARAM fireable: the fireable to destroy
Frees the fireable struct
**/
int destroy_fireable(struct fireable *fireable);
```

3.3 Request Handling

```
/**
@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
**/
int submit_request(struct graph*,
                           struct request *request);

/**
@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
**/
struct request *create_request(enum REQUESTS request,
                               void *args,
                               void (*f)(void *));

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

/**
@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
**/
int process_requests(struct graph *graph);

/**
@PARAM request: a request
@RETURN -1 for fail;
        0 for success
Destroys and frees a request
**/
int destroy_request(struct request *request);
```

3.4 Pausing/Resuming the Graph's execution

```
/**
@PARAM graph: the graph
@RETURN 0 for success
        -1 if it fails
Resumes run
**/
int resume_graph(struct graph *graph);

/**
@PARAM graph: the graph
@RETURN 0 for success
        -1 if it fails
Pauses run
**/
int pause_graph(struct graph *graph);

/**
```

```
@PARAM graph: the graph
Prints the graph with desired output
**/
void print_graph(struct graph *graph);

/**
@PARAM graph: the graph
@RETURN -1 for fail;
        0 for success
Destroys and frees the graph
**/
int destroy_graph(struct graph *graph);
```

3.5 Parse JSON to Graph

With the optional topologic-parse library.

```
/**
@PARAM path: path to graph input file
@RETURN the graph specified in the file
NOTE: DOES NOT WORK IN PYTHON
**/
struct graph *parse_json(const char *path);
```

4 Definitions

```
#ifdef DEBUG
#define TOPOLOGIC_DEBUG 1
#else
#define TOPOLOGIC_DEBUG 0
#endif

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

#define MAX_LOOPS 100
#define MAX_ATTEMPTS
#define PTHREAD_SLEEP_TIME 50

#define GRAPH_INIT() \
graph_init(-1, START_STOP, MAX_LOOPS
NODES|EDGES|FUNCTIONS|GLOBALS, SINGLE, IGNORE_FAIL_REQUEST)

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

```
#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)

#define MAX_ATTEMPTS 4
//Maximum attempts for system to check for pthread if WAIT enum is selected
#define THREAD_ATTEMPT_SLEEP 3
//Sleep amount before trying to create thread again for WAIT
```

5 Data Structures

The library comes with three main, and several auxiliary, data structures to manage each edge and vertex

```
/**fireable**/
struct fireable
{
    struct graph *graph;
    struct vertex *vertex;
    struct vertex_result *args;
    enum STATES color;
    int iloop;
};

/**edge_type**/
enum edge_type
{
    EDGE = 0,
    BI_EDGE = 1,
        SELF_EDGE = 2
};
```

```
/** Edge **/
struct edge
{
    int id; //Hash for number passed in,
            //will compare with other Edges in graph
            //Also unique, a la vertex
            //Perhaps @nanosecond level
    int (*f)(void *, const void *const);
    void *glbl;
    const void *const *a_vars; //To be shared among vertex a and shared edge
    struct vertex *a;
    struct vertex *b;
    enum edge_type edge_type;
    struct edge *bi_edge;
    pthread_mutex_t bi_edge_lock;
};

/**edge_request**/
struct edge_request
{
    struct vertex *a;
    struct vertex *b;
    int (*f)(void *, const void *const);
    void *glbl;
};

/**destroy_edge_request**/
struct destroy_edge_request
{
    struct vertex *a;
    struct vertex *b;
};

struct destroy_edge_id_request
{
    struct vertex *a;
    int id;
};

/**vertex_result**/
struct vertex_result
{
    void *vertex_argv;
    size_t vertex_size;
    void *edge_argv;
```

```
    size_t edge_size;
};

/**shared_edge**/
union shared_edge {
    void *vertex_data;
    const void *const *edge_data;
};

/**vertex **/
struct vertex
{
    int id; //Hash for number passed in,
            //will compare with other vertices in graph
            //Must be unique. If non-unique ID, error
    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;
};

/**vertex_request**/
struct vertex_request
{
    struct graph *graph;
    int id; //Hash for number passed in,
            //will compare with other vertices in graph
            //Must be unique. If non-unique ID, error
    void (*f)(struct graph *, struct vertex_result *, void *, void *);
    void *glbl;
};

/**mod_vertex_request**/
struct mod_vertex_request
{
    struct vertex *vertex;
    void (*f)(struct graph *, struct vertex_result *, void *, void*);
    void *glbl;
};

/**mod_edge_vars_request**/
struct mod_edge_vars_request
```

```
{
    struct vertex *vertex;
    void *edge_vars;
};

/**destroy_vertex_request**/
struct destroy_vertex_request
{
    struct graph *graph;
    struct vertex *vertex;
};

/**destroy_vertex_id_request**/
struct destroy_vertex_id_request
{
    struct graph *graph;
    int id;
};

/**
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
**/
enum CONTEXT
{
    NONE = 0,
    NONE_UNSAFE = 8,
    SINGLE = 1,
    SWITCH = 2,
    SWITCH_UNSAFE = 10,
};

/**
Enum for memory handling when creating threads -- 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
**/
```

```
enum MEM_OPTION
{
    ABORT = 0,
    WAIT = 1,
    CONTINUE = 2
}

/**
Enum for handling failed requests
NO_FAIL_REQUEST: All request must succeed else end the graph
IGNORE_FAIL_REQUEST: Ignore a request if it fails
**/
enum REQUEST_FLAG
{
    NO_FAIL_REQUEST = 0,
    IGNORE_FAIL_REQUEST = 1
};


/**
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
**/
enum STATES
{
    PRINT = 0,
    RED = 1,
    BLACK = 2,
    TERMINATE = 3
};

/**
Enum for snapshots.
NONE: Record nothing
START_STOP: Record first and last state
**/
enum SNAPSHOT
{
    NO_SNAP = -1,
    START_STOP = 0
};

/**
```

```
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
**/
enum VERBOSITY
{
    NO_VERB = 0,
    VERTICES = 1,
    EDGES = 2,
    FUNCTIONS = 4,
    GLOBALS = 8
};

/** Graph **/
struct graph
{
    enum CONTEXT context;
    enum MEM_OPTION mem_option;
    enum REQUEST_FLAG request_flag;
    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;             //CURRENT STATE {PRINT, RED, BLACK}
    sig_atomic_t previous_color;    //LAST NODE COLOR TO FIRE
    sig_atomic_t print_flag;        //0 DID NOT PRINT; 1 FINISHED PRINT
    sig_atomic_t red_vertex_count; //Number of RED nodes not reaped
    sig_atomic_t black_vertex_count; //Number of BLACK nodes not reaped
    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;
};

/**
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
**/
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
};

/** Request **/
struct request
{
    enum REQUESTS request;
    void (*f)(void *);
    void *args;
};
```

6 Usage/Functionality

6.1 `make` variants

`make` will build the C library and all tests in the "testing" directory. `make cpp` builds a C++ library. `make python` builds a Python library for Python 3.6+. `make python2` builds a Python library for Python 2.7. `make rust` builds the rust library to call these functions. It calls upon software tools `bindgen` and `cargo`. Install these, and `rustc` if you have not done so already. `make csharp` will build the C-Sharp library of files to be included in any C-Sharp project you

decide to make with the library. `make clean` cleans all binaries and *.o files. In addition, it calls on `cargo clean` to clean up the rust project.

6.2 Linking To Source Code

Include `include/topologic.h` in your source code, and link with the resulting library file `libtopologic.a`.

6.3 Windows

For running on Windows machines, Cygwin is required. Please take care to install the proper `gcc, flex, bison, g++, python 2.7, python 3.6+,` etc., libraries during setup. Cygwin 32-bit and Cygwin 64-bit both should work for running on Windows.

7 Errata

7.1 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.

7.2 Graph modification

Trying to delete the graph while running will result in undefined behavior. 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.

7.3 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.

7.4 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.

7.5 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.

7.6 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.