CAIDA Graph Analyzer

- Reference -

Project Reference

Enrico Gandaglia, Matr. 779537

Università degli studi di Torino
Dipartimento di Informatica

Contents

Chapter 1

Introduction

This library permits to load a CAIDA snapshot from
http://data.caida.org/datasets/as-relationships/ and do some analyze of the relationship between autonomous systems and the valley free paths between them using igraph library. To use this library, you need to have installed the igraph C library found at https://igraph.org/.

The graph created by this library is an annotated graph, with direct edges for provider-to-customer edges (in CAIDA snapshots they have value -1) and undirected edges for peer-to-peer edges (in CAIDA snapshots they have value 0). Customer-to-provider edges (value 1) can be easily achieved noting that they are the inverse of provider-to-customer edges, and working with the graph as if it were an undirected graph.

The library uses a permanent hashtable to store the association between `<as_number, vertex_id>`. It permits at the user to specify which Autonomous system analyze without knowing which vertex of the graph is. It can be loaded and saved at anytime in the program using the given hashtable's function.

Chapter 2

Folder structure

`caida_graph_analyzer`	
`- bin`	contains executable files
`- build`	contains .o files
`- dataset`	contains a dataset test in CAIDA snapshot's format
`- include`	contains header files
`- src`	contains source files
`- hashtable.txt`	permanent hashtable
`- libcga.a`	caida graph analyzer static library
`- Makefile`	create library, compile source code, etc.

The source files of the library are: `as_relationship.c` `display.c` `hash.c` `hashset.c` `hashtable.c`.
`graph_analysis.c` is an example of a program that use this library.
`cga.h` (located in `include` folder) is the main interface where all the other interface's library are included.

2.1 How to use this library

Use Makefile to compile and execute the library or the source code.

- make all : create the static library and execute the program using it
- make library : create a static library
- make build : build all the source code of the library
- make exec : execute the program using the static library
- make exec2 : execute the program using the compiled source code

You may need to change dependencies according to where the igraph library is located and/or installed in your system.

Chapter 3

Error codes

Some functions which can fail return a single integer error code. The error codes are defined by the `cga_status_t` enumeration.
The types of codes and their meaning are:

- `SUCCESS` : Nothing to report
- `NOMEM` : There's not enough memory to do a memory allocation
- `DPLKTKEY` : There's a duplicate key in data structure
- `NFOUND` : The element is missing
- `WRFORMAT` : Wrong input format
- `NWPERM` : No write permission
- `NRPERM` : No read permission

Chapter 4

Data Structures

The following data structures use an hash function based on the FNV_1 algorithm created by Glenn Fowler, Landor Curt Noll and Kiem-Phong Vo.
One of FNV's key advantages is that it is fast and very simple to implement.

cga_hash - Produce the hash of the given value

```
unsigned long cga_hash(unsigned long key)
```

Arguments:

`key`: the value for which the hash is calculated

Returns:

The hash code of the given value

To resolve collisions both the hashtable and the hashset use concatenation.

Hash Table

About cga_hashtable_t object

Due to the fact that the AS Numbers are defined as 32-bit integers, and the actual size of the C integer types varies by implementation, the key field of hashtable is currently defined for unsigned long types (at least 32-bit in size).

Constructor and Destructor

`cga_hashtable_t` objects have to be created using `cga_ht_init()` constructor.

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

cga_ht_init - Creates a hashtable object

```
cga_hashtable_t* cga_ht_init(size_t size)
```

This function creates a `cga_hashtable_t` object of the given size.
Every hashtable object created by this function should be destroyed (ie. the memory allocated for it should be freed) when it is not needed anymore with the function `cga_ht_destroy()`.

Arguments:

`size`:	The maximum size of the hashtable

Returns:

`cga_hashtable_t*`:	pointer to the newly created hashtable object

cga_ht_destroy - Destroys a hashtable object

```
void cga_ht_destroy(cga_hashtable_t *ht)
```

All hashtable objects created by `cga_ht_init()` should be destroyed by this function.

Arguments:

`ht`:	Pointer to the (previously initialized) hashtable object to destroy

Functions

cga_ht_insert - Inserts the association key-val in the hashtable

```
cga_status_t cga_ht_insert(cga_hashtable_t *ht, unsigned
            long key, igraph_integer_t value)
```

Inserts the association <key, value> in the hashtable. Any duplicated key values won't be inserted.

Arguments:

`ht`:	Pointer to the hashtable object
`key`:	The key to insert
`value`:	The value associated with the key

Returns:

`cga_status_t`: `SUCCESS` if the operation completed without errors, `NOMEM` if there's not enough memory, `DPLKTKEY` if the given key is already in the hashtable.

cga_ht_delete - Deletes the association key-val in the hashtable

```
cga_status_t cga_ht_delete(cga_hashtable_t *ht, unsigned
                        long key)
```

Deletes the association <key, value> in the hashtable. If the specified key is not found, this function returns with an error.

Arguments:

`ht`:	Pointer to the hashtable object
`key`:	The key to delete

Returns:

`cga_status_t`: `SUCCESS` if the operation completed without errors, `NFOUND` if the given key is not in the hashtable.

cga_ht_nelems - Gives the number of keys stored in the hashtable object

```
size_t cga_ht_nelems(cga_hashtable_t *ht)
```

Arguments:

`ht`:	Pointer to the hashtable object

Returns:

The number of elements stored in the hashtable object

cga_ht_contains - Check whether the supplied key is contained in the hashtable

```
int cga_ht_contains(cga_hashtable_t *ht, unsigned long
                              key)
```

Arguments:

`ht`:	Pointer to the hashtable object
`key`:	The key to look for

Returns:

1 if the key is found, 0 otherwise

cga_ht_search - Search the supplied key in the hashtable

```
igraph_integer_t* cga_ht_search(cga_hashtable_t *ht,
                    unsigned long key)
```

The supplied key is searched in the hashtable object and the value to which the specified key is mapped is returned. Returns NULL if the supplied key is not found.

Arguments:

`ht`:	Pointer to the hashtable object
`key`:	The key to look for

Returns:

The value to which the specified key is mapped, NULL otherwise.

cga_ht_clear - Deletes all the <key, value> from the hashtable

```
void cga_ht_clear(cga_hashtable_t *ht)
```

This function deletes all the elements stored in the hashtable, it does not free the hashtable object. For that, you have to call `cga_ht_destroy()`.

Arguments:

`ht:`	Pointer to the hashtable object to clear

cga_ht_save_to_file - Saves the hashtable in a file

```
cga_status_t cga_ht_save_to_file(cga_hashtable_t *ht, FILE
*outstream)
```

Saves all the association <key, value> of the hashtable in the specified file.
The format of the resulting file corresponds to:

```
key value\n
...
key value\n
```

The file must be opened with the write privilege, or an error message will be displayed in `stderr` warning the user.

Arguments:

`ht:`	Pointer to the hashtable object
`outstream:`	FILE pointer opened with the write privilege

Returns:

`cga_status_t`: `SUCCESS` if the operation completed without errors, `NWPERM` if the given file descriptor has no write privilege.

cga_ht_load_from_file - Loads the hashtable from a file

```
cga_status_t cga_ht_load_from_file(cga_hashtable_t *ht,
                                   FILE *instream)
```

Loads all the association <key, value> from a given file into the hashtable.
The file must respect this format:

```
key value\n
...
key value\n
```

If there is a syntax error the hashtable will be partially loaded with the <key, value> parsed before the error occurred, and an error string will be printed in `stderr` showing the error and the line that caused it.
The file must be opened with the read privilege.

Arguments:

`ht:`	Pointer to the hashtable object
`instream:`	FILE pointer opened with the read privilege

Returns:

`cga_status_t`: `SUCCESS` if the operation completed without errors, `NRPERM` if the given file pointer has no read privilege, `WRFORMAT` if a syntax error occurred.

Hash Set

Constructor and Destructor

`cga_hashset_t` objects have to be created using `cga_hs_init()` constructor.

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

cga_hs_init - Creates a hashset object

```
cga_hashset_t* cga_hs_init(size_t size)
```

This function creates a `cga_hashset_t` object of the given size.
Every hashset object created by this function should be destroyed (ie. the memory allocated for it should be freed) when it is not needed anymore with the function `cga_hs_destroy()`.

Arguments:

`size`:	The maximum size of the hashset

Returns:

`cga_hashset_t*`:	pointer to the newly created hashset object

cga_hs_destroy - Destroys a hashset object

```
void cga_hs_destroy(cga_hashset_t *hs)
```

All hashset object created by `cga_hs_init()` should be destroyed by this function.

Arguments:

`hs`:	Pointer to the (previously initialized) hashset object to destroy

Functions

cga_hs_insert - Inserts an element in the hashset

```
cga_status_t cga_hs_insert(cga_hashset_t *hs,
                igraph_integer_t elem)
```

Inserts an element in the hashset. Any duplicated element won't be inserted.

Arguments:

`hs:`	Pointer to the hashset object
`elem:`	The element to insert

Returns:

`cga_status_t`: `SUCCESS` if the operation completed without errors, `NOMEM` if there's not enough memory, `DPLKTKEY` if the given element is already in the hashset.

cga_hs_delete - Deletes an element in the hashset

```
cga_status_t cga_hs_delete(cga_hashset_t *hs,
                igraph_integer_t elem)
```

Deletes an element in the hashset. If the specified element is not found, this function returns with an error.

Arguments:

`hs:`	Pointer to the hashset object
`elem:`	The element to delete

Returns:

`cga_status_t`: `SUCCESS` if the operation completed without errors, `NFOUND` if the given element is not in the hashset.

cga_hs_nelems - Gives the number of elements stored in the hashset object

```
size_t cga_hs_nelems(cga_hashset_t *hs)
```

Arguments:

`hs`:	Pointer to the hashset object

Returns:

The number of elements stored in the hashset object

cga_hs_contains - Check whether the supplied element is contained in the hashset

```
int cga_hs_contains(cga_hashset_t *hs, igraph_integer_t
                               elem)
```

Arguments:

`hs`:	Pointer to the hashset object
`elem`:	The element to look for

Returns:

1 if the element is found, 0 otherwise

cga_hs_clear - Deletes all the elements from the hashset

```
void cga_hs_clear(cga_hashset_t *hs)
```

This function deletes all the elements stored in the hashset, it does not free the hashset object. For that, you have to call `cga_hs_destroy()`.

Arguments:

`hs`:	Pointer to the hashset object to clear

Chapter 5

Parse and load CAIDA snapshots

cga_load_snapshot - Parse and load CAIDA snapshot data into the graph

```
void cga_load_snapshot(igraph_t *graph, cga_hashtable_t
                 *ht, FILE *instream)
```

This function read an as-rel dataset snapshot file provided by CAIDA and load its data into the graph.
The format of the snapshot file is:

```
<as_num_1>|<as_num_2>|relationship
```

The graph will be initialized as a partially directed graph: it will have directed provider-to-customer edges and undirected peer-to-peer edges.
Each vertex has a numeric attribute "label" that identify the as_number of the autonomous system.
Each edge has a numeric attribute "type" that identify if the edge is a provider-to-customer edge (-1) or a peer-to-peer edge (0).
N.B. In the main program it must be set the igraph attribute table with the following line: `igraph_i_set_attribute_table(&igraph_cattribute_table)`. Without it this function can't set the attributes of the graph.
To reference the vertex id in the graph with its as_number, this function store in the hashtable `ht` the association `<as_number, vertex_id>`. If the given file has no read privileges, this function abort the program with an error printed in stderr.

Arguments:

`graph:` Pointer to an uninitialized graph
`ht:` Pointer to an already initialized hashtable. It will be used to store the assotiation `<as_number, vertex_id>`
`instream:` The input CAIDA snapshot file pointer. It needs read privilege

Chapter 6

Valley free paths searching

Paths searching is done using a modified Depth First Search with backtracking that retrieves only the valley free paths between two nodes.
A path is valley free if and only if the following conditions hold true:

- A **provider-to-customer** edge can be followed by only provider-to-customer or sibling to sibling edges.
- A **peer-to-peer** edge can be followed by only provider-to-customer or sibling to sibling edges

`cga_dfs_vfree_rec` and `cga_dfs_vfree_it` get all valley free paths between two nodes, `cga_as_analysis` analyze all valley free paths from a node to all other reachable nodes in the graph, `cga_graph_analysis` analyze all valley free paths from all reachable nodes in the graph to all reachable nodes in the graph.
`cga_as_analysis` and `cga_graph_analysis` can be really slow for huge graphs, so they can use multiple threads to do their job. Ideally the number of threads should be a divisor of the graph's vertices number, or at least less than that number.

6.1 Check if a path is valley free

cga_is_valley_free - Check if the given path is a valley free path

```
int cga_is_valley_free(igraph_t *graph,
        igraph_vector_int_t *path)
```

Given a path, this function returns 1 if the path is valley free, 0 otherwise.
A valley free path respect one of the following patterns:

1. An uphill path (a sequence of edges that are either customer-to provider or sibling-to-sibling edges)
2. A downhill path (a sequence of edges that are either provider-to-customer or sibling-to-sibling edges)

3. An uphill path followed by a downhill path
4. An uphill path followed by a peer-to-peer edge
5. A peer-to-peer edge followed by a downhill path
6. An uphill path followed by a peer-to-peer edge, which is followed by a downhill path

Arguments:

`graph:` Pointer to the graph object
`path:` Pointer to a vector consisting of vertex_ids that forms the path to check

Returns:

1 if the path is valley free, 0 otherwise

6.2 Valley free paths searching between two nodes

6.2.1 About cga_dfs_vfree paths search

There are two functions that retrieve all the valley free paths between two nodes: `cga_dfs_vfree_rec` and `cga_dfs_vfree_it`.

`cga_dfs_vfree_rec` takes in input 2 nodes and recursively search all the paths between them. It uses a lazy adjacency list with incoming and outgoing edges (any multiple edges are removed) to search for neighbors and an hashset to store all the nodes in the current path (so they will not be explored again). At every new node in the current path, this function checks if it is still a valley free path. If not, the new node is removed and that possible path is pruned.
This function continues until all the possible valley free paths between the two given nodes are discovered.

`cga_dfs_vfree_it` is similar to the recursive one, except that, in addition to the lazy adjacency list and the hashset, it uses a stack data structure to explore and find all the possible valley free paths between two given nodes.
Initially, the start node and all of its neighbors are pushed inside the stack, and the current path is initialized with the starting node.
At every iteration, the function check the top of the stack. If the node is equal

to the last node of the current path, it means that its neighbors are already explored without reaching the end node, so the node is removed from the stack and from the current path. If not, the function checks if the path is a valley free path and continues like the recursive function until all the possible valley free paths between the two given nodes are discovered.

`cga_graph_analysis` and `cga_as_analysis` use `cga_dfs_vfree_it` to analyze the paths of the graph.

cga_dfs_vfree_rec - Recursively search all the valley free paths between two nodes

```
void cga_dfs_vfree_rec(igraph_t *graph,
igraph_vector_int_t *res, igraph_integer_t from,
igraph_integer_t to)
```

Recursively search all the valley free paths between two nodes.
The resulting paths are stored in `res`. All paths are separated by -1 markers.

Arguments:

`graph:` Pointer to the graph object
`res:` Initialized vector, all the resulting paths are stored here, separated by -1 markers. The paths are included in arbitrary order, as they are found.
`from:` The starting vertex_id
`to:` The ending vertex_id

cga_dfs_vfree_it - Iteratively search all the valley free paths between two nodes

```
void cga_dfs_vfree_it(igraph_t *graph, igraph_vector_int_t
*res, igraph_integer_t from, igraph_integer_t to)
```

Iteratively search all the valley free paths between two nodes.
The resulting paths are stored in `res`. All paths are separated by -1 markers.

Arguments:

`graph:` Pointer to the graph object
`res:` Initialized vector, all paths are returned here, separated by -1 markers. The paths are included in arbitrary order, as they are found.
`from:` The starting vertex_id
`to:` The ending vertex_id

6.3 Valley free paths analysis

6.3.1 Calculate valley free path cost

cga_path_cost - Calculate the cost of the valley free path

```
int cga_path_cost(igraph_t *graph, igraph_vector_int_t
                        *path)
```

Calculate the cost of the valley free path as an algebraic sum of the relationships between the Autonomous Systems.

- Customer-to-provider has value 1
- Provider-to-customer has value -1
- Peer-to-peer has value 0

Arguments:

`graph:` Pointer to the graph object
`path:` Pointer to a vector of vertex_ids, where each couple has an edge in the graph that connects them. If it does not exists then it will be treated as a customer-to-provider edge.

Returns:

Returns the algebraic sum of the relationships between the Autonomous Systems

6.3.2 Valley free paths analysis functions

cga_as_analysis - Analyze and print all the valley free paths from a node to all other nodes

```
cga_status_t cga_as_analysis(igraph_t *graph,
igraph_integer_t vertex, unsigned int nthreads, char
                    *filename)
```

This function analyze and print in n files all the valley free paths from a node to all other nodes, where n is the number of threads used to compute the analysis. The search for paths is done through the use of `cga_dfs_vfree_it` function.

`nthreads` must be at least 1. Specifying a number greater than 1 will use more threads to compute the analysis.
Ideally the number of threads should be a divisor of the graph's vertices number, or at least less than that number. Each thread will have a range of vertices of the graph to analyze, and the results will be stored in a file that uses the following naming convention:

For a generic thread *n*, given the name of the file `filename`, the file name will be `filename_n.csv`.

`filename` shouldn't have the extension of the file (it will be added automatically as .csv) and can be written as a path. If a path is given, and not only a filename, the folders forming the path must already exist.
The files' output consist of a header and the content.
The header `<from,to,length,cost>` represents the starting autonomous system, the target autonomous system, the path length and the cost given as the algebraic sum of the AS relationship in the path. Each line of the content represents the analysis of a valley free path between two nodes, with each information separated by a comma.

Arguments:

`graph:` Pointer to the graph object
`vertex:` The vertex_id to analyze. It will be the starting vertex from where the paths are calculated
`nthreads:` The number of threads used to analyze the vertex's paths. The given value must be at least greater or equal to 1
`filename:` Part of the name used to compose the name of the output file. It should not have the extension and can be a path (in this case the folders that compose the path must already exists)

cga_graph_analysis - Analyze and print all the valley free paths from all nodes to all other nodes

```
cga_status_t cga_graph_analysis(igraph_t *graph, unsigned
                int nthreads, char *filename)
```

This function analyze and print in *n* files all the valley free paths from all nodes to all other nodes, where *n* is the number of threads used to compute the analysis.
The search for paths is done through the use of `cga_dfs_vfree_it` function.
`nthreads` must be at least 1. Specifying a number greater than 1 will use more threads to compute the analysis.
Ideally the number of threads should be a divisor of the graph's vertices number, or at least less than that number. Each thread will have a range of vertices of the

graph to analyze, and the results will be stored in a file that uses the following naming convention:

For a generic thread n, given the name of the file `filename`, the file name will be `filename_n.csv`.

`filename` shouldn't have the extension of the file (it will be added automatically as .csv) and can be written as a path. If a path is given, and not only a filename, the folders forming the path must already exist.
The files' output consist of a header and the content.
Since there can be multiple valley free paths between two nodes, the header `<from, to, avg length, min length, max length, avg cost, min cost, max cost>` represents the starting autonomous system, the target autonomous system, the average path length, the minimum path length, the maximum path length, the average cost, the minimum and the maximum cost of all the paths between the two nodes.
The cost is given as the algebraic sum of the AS relationship in the path. Each line of the content represents the analysis of all the valley free paths between two nodes, with each information separated by a comma.

Arguments:

`graph:` Pointer to the graph object
`nthreads:` The number of threads used to analyze the vertex's paths. The given value must be at least greater or equal to 1
`filename:` Part of the name used to compose the name of the output file. It should not have the extension and can be a path (in this case the folders that compose the path must already exists)

Chapter 7

Degree of freedom of the paths between two nodes

cga_degree_freedom_path - Calculate the degree of freedom of the paths between two nodes

```
float cga_degree_freedom_path(igraph_t *graph,
igraph_integer_t vertex1_id, igraph_integer_t vertex2_id,
unsigned int *num_vfree, unsigned int *num_novfree)
```

Calculate the degree of freedom of the paths between two nodes.
Given all the paths between two nodes, this function count all the valley free and no valley free paths. The formula for the degree of freedom of the paths between two nodes is calculated as: `num_path_vfree / num_path_novfree`.
If `num_vfree` and `num_novfree` pointers are not `NULL`, this function stores in these pointers the number of valley free and no valley free paths.

Arguments:

`graph:` Pointer to the graph object
`vertex1_id:` The starting vertex
`vertex2_id:` The ending vertex
`num_vfree:` Pointer to an unsigned int variable. If not `NULL`, the number of valley free paths will be stored here.
`num_novfree:` Pointer to an unsigned int variable. If not `NULL`, the number of no valley free paths will be stored here.

Returns:

The degree of freedom of the paths between two nodes.

Chapter 8

Utils functions to print informations

cga_print_info - Print some informations about the graph

```
void cga_print_info(igraph_t *graph)
```

Prints in `stdout` the number of vertices, edges and the type of the given graph.

Arguments:

`graph`: Pointer to the graph object

cga_print_vector_label - Converts the vertex_ids in the input vector in as_num and prints them in the file

```
void cga_print_vector_label(igraph_t *graph,
   igraph_vector_int_t *v, FILE *ostream)
```

Arguments:

`graph`: Pointer to the graph object
`v`: Vector containing vertex_ids of the graph
`ostream`: File pointer used as output

cga_print_degree_freedom_path - Prints the degree of freedom info in stdout

```
void cga_print_degree_freedom_path(igraph_t *graph,
igraph_integer_t vertex1_id, igraph_integer_t vertex2_id)
```

Calculates the degree of freedom of the paths between two nodes using `cga_degree_freedom_path` and prints its output in stdout.

Arguments:

`graph:`	Pointer to the graph object
`vertex1_id:`	The starting vertex used to calculate the degree of freedom
`vertex2_id:`	The ending vertex used to calculate the degree of freedom

cga_print_adj - Prints the as_num of the graph as an adjacency list

```
void cga_print_adj(igraph_t *graph)
```

Prints the as_numbers of the graph as an adjacency list in stdout

Arguments:

`graph:`	Pointer to the graph object

cga_print_result_label_vfree - Prints a list of paths alongside valley free info in a file

```
void cga_print_result_label_vfree(igraph_t *graph,
     igraph_vector_int_t *res, FILE *ostream)
```

Given a vector containing a list of paths separated by -1 marker, this function prints in the file ostream all the containing paths line by line, converting the vertex_ids of the generic path in as_number.
Next to each path this function prints OK if the path is a valley free path, or NOPE if it is a no valley free path.

Arguments:

`graph:`	Pointer to the graph object
`res:`	Pointer to a vector containing a list of paths separated by -1 markers
`ostream:`	File pointer used as output

cga_print_result_label - Prints a list of paths in a file

```
void cga_print_result_label(igraph_t *graph,
  igraph_vector_int_t *res, FILE *ostream)
```

Given a vector containing a list of paths separated by -1 marker, this function prints in the file ostream all the containing paths line by line, converting the vertex_ids of the generic path in as_number.

Arguments:

`graph:`	Pointer to the graph object
`res:`	Pointer to a vector containing a list of paths separated by -1 markers
`ostream:`	File pointer used as output