NAME

libcgraph – abstract graph library

SYNOPSIS

#include <graphviz/cgraph.h>

TYPES

Agraph_t;
Agnode_t;
Agedge_t;
Agdesc_t;
Agdisc_t;
Agsym_t;
Agrec_t;
Agcbdisc_t;

GLOBALS

Agiddisc_t AgIdDisc;
Agiodisc_t AgIoDisc;
Agdisc_t AgDefaultDisc;

GRAPHS

Agraph_t *agopen(char *name, Agdesc_t kind, Agdisc_t *disc);
int agclose(Agraph_t *g);
Agraph_t *agread(void *channel, Agdisc_t *);
Agraph_t *agmemread(char *);
Agraph_t *agconcat(Agraph_t *g, const char *filename, void *channel, Agdisc_t *disc);
int agwrite(Agraph_t *g, void *channel);
size_t agnnodes_z(const Agraph_t *g);
int agnnodes(Agraph_t *g),agnedges(Agraph_t *g), agnsubg(Agraph_t * g);
int agisdirected(Agraph_t * g),agisundirected(Agraph_t * g),agisstrict(Agraph_t * g), agissimple(Agraph_t * g);
bool graphviz_acyclic(Agraph_t *g, const graphviz_acyclic_options_t *opts, size_t *num_rev);
void graphviz_tred(Agraph_t *g, const graphviz_tred_options_t *opts);
void graphviz_unflatten(Agraph_t *g, const graphviz_unflatten_options_t *opts);

SUBGRAPHS

Agraph_t *agsubg(Agraph_t *g, char *name, int createflag);
Agraph_t *agidsubg(Agraph_t * g, unsigned long id, int cflag);
Agraph_t *agfstsubg(Agraph_t *g), agnxtsubg(Agraph_t *);
Agraph_t *agparent(Agraph_t *g);
int agdelsubg(Agraph_t * g, Agraph_t * sub); /* same as agclose() */

NODES

Agnode_t *agnode(Agraph_t *g, char *name, int createflag);
Agnode_t *agidnode(Agraph_t *g, ulong id, int createflag);
Agnode_t *agsubnode(Agraph_t *g, Agnode_t *n, int createflag);
Agnode_t *agfstnode(Agraph_t *g);
Agnode_t *agnxtnode(Agraph_t *g, Agnode_t *n);
Agnode_t *agprvnode(Agraph_t *g, Agnode_t *n);
Agnode_t *aglstnode(Agraph_t *g);
int agdelnode(Agraph_t *g, Agnode_t *n);
int agdegree(Agraph_t *g, Agnode_t *n, int use_inedges, int use_outedges);
int agcountuniqedges(Agraph_t * g, Agnode_t * n, int in, int out);

EDGES

Agedge_t *agedge(Agraph_t* g, Agnode_t *t, Agnode_t *h, char *name, int createflag);
Agedge_t *agidedge(Agraph_t * g, Agnode_t * t, Agnode_t * h, unsigned long id, int createflag);

```
Agedge_t    *agsubedge(Agraph_t *g, Agedge_t *e, int createflag);
Agnode_t    *aghead(Agedge_t *e), *agtail(Agedge_t *e);
Agedge_t    *agfstedge(Agraph_t* g, Agnode_t *n);
Agedge_t    *agnxtedge(Agraph_t* g, Agedge_t *e, Agnode_t *n);
Agedge_t    *agfstin(Agraph_t* g, Agnode_t *n);
Agedge_t    *agnxtin(Agraph_t* g, Agedge_t *e);
Agedge_t    *agfstout(Agraph_t* g, Agnode_t *n);
Agedge_t    *agnxtout(Agraph_t* g, Agedge_t *e);
int         agdeledge(Agraph_t *g, Agedge_t *e);
Agedge_t    *agopp(Agedge_t *e);
int         ageqedge(Agedge_t *e0, Agedge_t *e1);
```

STRING ATTRIBUTES

```
Agsym_t     *agattr_text(Agraph_t *g, int kind, char *name, const char *value);
Agsym_t     *agattrsym(void *obj, char *name);
Agsym_t     *agnxtattr(Agraph_t *g, int kind, Agsym_t *attr);
char        *agget(void *obj, char *name);
char        *agxget(void *obj, Agsym_t *sym);
int         agset(void *obj, char *name, char *value);
int         agxset(void *obj, Agsym_t *sym, char *value);
int         agsafeset(void *obj, char *name, char *value, char *def);
int         agcopyattr(void *, void *);
```

RECORDS

```
void        *agbindrec(void *obj, char *name, unsigned int size, move_to_front);
Agrec_t     *aggetrec(void *obj, char *name, int move_to_front);
int         agdelrec(Agraph_t *g, void *obj, char *name);
void        aginit(Agraph_t * g, int kind, char *rec_name, int rec_size, int move_to_front);
void        agclean(Agraph_t * g, int kind, char *rec_name);
```

CALLBACKS

```
int             *agpopdisc(Agraph_t *g);
void        agpushdisc(Agraph_t *g, Agcbdisc_t *disc);
int             agcallbacks(Agraph_t * g, int flag);
```

STRINGS

```
char        *agstrdup(Agraph_t *, char *);
char        *agstrdup_html(Agraph_t *, char *);
int         aghtmlstr(char *);
char        *agstrbind(Agraph_t * g, char *);
int         strfree(Agraph_t *, char *);
char        *agstrcanon(char *, char *);
```

GENERIC OBJECTS

```
Agraph_t    *agraphof(void*);
Agraph_t    *agroot(void*);
int         agcontains(Agraph_t*, void*);
char        *agnameof(void*);
void        agdelete(Agraph_t *g, void *obj);
int         agobjkind(void *obj);
Agrec_t     *AGDATA(void *obj);
ulong       AGID(void *obj);
int         AGTYPE(void *obj);
```

ERROR REPORTING

```
typedef enum { AGWARN, AGERR, AGMAX, AGPREV } agerrlevel_t;
typedef int (*agusererf) (char*);
```

agerrlevel_t	agerrno;
agerrlevel_t	agseterr(agerrlevel_t);
char	*aglasterr(void);
int	agerr(agerrlevel_t level, char *fmt, ...);
void	agerrorf(char *fmt, ...);
void	agwarningf(char *fmt, ...);
int	agerrors(void);
agusererrf	agseterrf(agusererrf);

DESCRIPTION

Libcgraph supports graph programming by maintaining graphs in memory and reading and writing graph files. Graphs are composed of nodes, edges, and nested subgraphs. These graph objects may be attributed with string name-value pairs and programmer-defined records (see Attributes).

All of Libcgraph's global symbols have the prefix **ag** (case varying). In the following, if a function has a parameter **int createflag** and the object does not exist, the function will create the specified object if **createflag** is non-zero; otherwise, it will return NULL.

GRAPH AND SUBGRAPHS

A ''main'' or ''root'' graph defines a namespace for a collection of graph objects (subgraphs, nodes, edges) and their attributes. Objects may be named by unique strings or by integer IDs.

agopen creates a new graph with the given name and kind. (Graph kinds are **Agdirected**, **Agundirected**, **Agstrictdirected**, and **Agstrictundirected**. A strict graph cannot have multi-edges or self-arcs.) The final argument points to a discpline structure which can be used to tailor I/O and ID allocation. Typically, a NULL value will be used to indicate the default discipline **AgDefaultDisc**. **agclose** deletes a graph, freeing its associated storage. **agread**, **agwrite**, and **agconcat** perform file I/O using the graph file language described below. **agread** constructs a new graph while **agconcat** merges the file contents with a pre-existing graph. Though I/O methods may be overridden, the default is that the channel argument is a stdio FILE pointer. In that case, if any of the streams are wide-oriented, the behavior is undefined. **agmemread** attempts to read a graph from the input string.

The functions **agisdirected**, **agisundirected**, **agisstrict**, and **agissimple** can be used to query if a graph is directed, undirected, strict (at most one edge with a given tail and head), or simple (strict with no loops), respectively,

agsubg finds or creates a subgraph by name. **agidsubg** allows a programmer to specify the subgraph by a unique integer ID. A new subgraph is initially empty and is of the same kind as its parent. Nested subgraph trees may be created. A subgraph's name is only interpreted relative to its parent. A program can scan subgraphs under a given graph using **agfstsubg** and **agnxtsubg**. A subgraph is deleted with **agdelsubg** (or **agclose**). The **agparent** function returns the immediate parent graph of a subgraph, or itself if the graph is already a root graph.

By default, nodes are stored in ordered sets for efficient random access to insert, find, and delete nodes. The edges of a node are also stored in ordered sets. The sets are maintained internally as splay tree dictionaries using Phong Vo's cdt library.

agnnodes, **agnedges**, and **agnsubg** return the sizes of node, edge and subgraph sets of a graph. **agnnodes_z** performs the same function as **agnnodes** but its result is a larger unsigned type, so allows dealing with larger graphs. The function **agdegree** returns the size of the edge set of a nodes, and takes flags to select in-edges, out-edges, or both. The function **agcountuniqedges** returns the size of the edge set of a nodes, and takes flags to select in-edges, out-edges, or both. Unlike **agdegree**, each loop is only counted once.

NODES

A node is created by giving a unique string name or programmer defined integer ID, and is represented by a unique internal object. (Node equality can checked by pointer comparison.)

agnode searches in a graph or subgraph for a node with the given name, and returns it if found. **agidnode** allows a programmer to specify the node by a unique integer ID. **agsubnode** performs a similar operation on an

existing node and a subgraph.

agfstnode and **agnxtnode** scan node lists. **agprvnode** and **aglstnode** are symmetric but scan backward. The default sequence is order of creation (object timestamp.) **agdelnode** removes a node from a graph or subgraph.

EDGES

An abstract edge has two endpoint nodes called tail and head where all outedges of the same node have it as the tail value and similarly all inedges have it as the head. In an undirected graph, head and tail are interchangeable. If a graph has multi-edges between the same pair of nodes, the edge's string name behaves as a secondary key.

agedge searches in a graph or subgraph for an edge between the given endpoints (with an optional multi-edge selector name) and returns it if found or created. Note that, in undirected graphs, a search tries both orderings of the tail and head nodes. If the **name** is NULL, then an anonymous internal value is generated. **agidedge** allows a programmer to create an edge by giving its unique integer ID. **agsubedge** performs a similar operation on an existing edge and a subgraph. **agfstin**, **agnxtin**, **agfstout**, and **agnxtout** visit directed in- and out- edge lists, and ordinarily apply only in directed graphs. **agfstedge** and **agnxtedge** visit all edges incident to a node. **agtail** and **aghead** get the endpoint of an edge. **agdeledge** removes an edge from a graph or subgraph.

Note that an abstract edge has two distinct concrete representations: as an in-edge and as an out-edge. In particular, the pointer as an out-edge is different from the pointer as an in-edge. The function **ageqedge** canonicalizes the pointers before doing a comparison and so can be used to test edge equality. The sense of an edge can be flipped using **agopp**.

INTERNAL ATTRIBUTES

Programmer-defined values may be dynamically attached to graphs, subgraphs, nodes, and edges. Such values are either character string data (for I/O) or uninterpreted binary records (for implementing algorithms efficiently).

STRING ATTRIBUTES

String attributes are handled automatically in reading and writing graph files. A string attribute is identified by name and by an internal symbol table entry (**Agsym_t**) created by Libcgraph. Attributes of nodes, edges, and graphs (with their subgraphs) have separate namespaces. The contents of an **Agsym_t** have a **char* name** for the attribute's name, a **char* defval** field for the attribute's default value, and an **int id** field containing the index of the attribute's specific value for an object in the object's array of attribute values.

agattr creates or looks up attributes. **kind** may be **AGRAPH**, **AGNODE**, or **AGEDGE**. If **value** is **(char*)0)**, the request is to search for an existing attribute of the given kind and name. Otherwise, if the attribute already exists, its default for creating new objects is set to the given value; if it does not exist, a new attribute is created with the given default, and the default is applied to all pre-existing objects of the given kind. If **g** is NULL, the default is set for all graphs created subsequently. **agattrsym** is a helper function that looks up an attribute for a graph object given as an argument. **agnxtattr** permits traversing the list of attributes of a given type. If **NULL** is passed as an argument it gets the first attribute; otherwise it returns the next one in succession or returns **NULL** at the end of the list. **agget** and **agset** allow fetching and updating a string attribute for an object taking the attribute name as an argument. **agxget** and **agxset** do this but with an attribute symbol table entry as an argument (to avoid the cost of the string lookup). Note that **agset** will fail unless the attribute is first defined using **agattr**. **agsafeset** is a convenience function that ensures the given attribute is declared before setting it locally on an object.

It is sometimes convenient to copy all of the attributes from one object to another. This can be done using **agcopyattr**. This fails and returns non-zero of argument objects are different kinds, or if all of the attributes of the source object have not been declared for the target object.

STRINGS

Libcgraph performs its own storage management of strings as reference-counted strings. The caller does not need to dynamically allocate storage.

agstrdup returns a pointer to a reference-counted copy of the argument string, creating one if necessary.

agstrbind returns a pointer to a reference-counted string if it exists, or NULL if not. All uses of cgraph strings need to be freed using **agstrfree** in order to correctly maintain the reference count.

The cgraph parser handles HTML-like strings. These should be indistinguishable from other strings for most purposes. To create an HTML-like string, use **agstrdup_html**. The **aghtmlstr** function can be used to query if a string is an ordinary string or an HTML-like string.

agstrcanon returns a pointer to a version of the input string canonicalized for output for later re-parsing. This includes quoting special characters and keywords. The application passes in a buffer as the second argument. Note that the buffer may not be used; if the input string is in canonical form, the function will just return a pointer to it. The input string must have been created using **agstrdup** or **agstrdup_html**.

RECORDS

Uninterpreted records may be attached to graphs, subgraphs, nodes, and edges for efficient operations on values such as marks, weights, counts, and pointers needed by algorithms. Application programmers define the fields of these records, but they must be declared with a common header as shown below.

```
typedef struct {
  Agrec_t       header;
  /* programmer-defined fields follow */
} user_data_t;
```

Records are created and managed by Libcgraph. A programmer must explicitly attach them to the objects in a graph, either to individual objects one at a time via **agbindrec**, or to all the objects of the same class in a graph via **aginit**. (Note that for graphs, aginit is applied recursively to the graph and its subgraphs if rec_size is negative (of the actual rec_size.)) The **name** argument of a record distinguishes various types of records, and is programmer defined (Libcgraph reserves the prefix **_ag**). If size is 0, the call to **agbindrec** is simply a lookup. The function **aggetrec** can also be used for lookup. **agdelrec** deletes a named record from one object. **agclean** does the same for all objects of the same class in an entire graph.

Internally, records are maintained in circular linked lists attached to graph objects. To allow referencing application-dependent data without function calls or search, Libcgraph allows setting and locking the list pointer of a graph, node, or edge on a particular record. This pointer can be obtained with the macro **AGDATA(obj)**. A cast, generally within a macro or inline function, is usually applied to convert the list pointer to an appropriate programmer-defined type.

To control the setting of this pointer, the **move_to_front** flag may be **TRUE** or **FALSE**. If **move_to_front** is **TRUE**, the record will be locked at the head of the list, so it can be accessed directly by **AGDATA(obj)**. The lock can be subsequently released or reset by a call to **aggetrec**.

DISCIPLINES

(This section is not intended for casual users.) Programmer-defined disciplines customize certain resources-ID namespace and I/O - needed by Libcgraph. A discipline struct (or NULL) is passed at graph creation time.

```
struct Agdisc_s {          /* user's discipline */
  Agiddisc_t        *id;
  Agiodisc_t        *io;
} ;
```

A default discipline is supplied when NULL is given for any of these fields.

ID DISCIPLINE

An ID allocator discipline allows a client to control assignment of IDs (uninterpreted integer values) to objects, and possibly how they are mapped to and from strings.

```
struct Agiddisc_s {          /* object ID allocator */
```

```
  void *(*open) (Agraph_t * g, Agdisc_t*);      /* associated with a graph */
  long (*map) (void *state, int objtype, char *str, unsigned long *id, int createflag);
  long (*alloc) (void *state, int objtype, unsigned long id);
  void (*free) (void *state, int objtype, unsigned long id);
  char *(*print) (void *state, int objtype, unsigned long id);
  void (*close) (void *state);
};
```

open permits the ID discipline to initialize any data structures that it maintains per individual graph. Its return value is then passed as the first argument (void *state) to all subsequent ID manager calls.

alloc informs the ID manager that Libcgraph is attempting to create an object with a specific ID that was given by a client. The ID manager should return TRUE (nonzero) if the ID can be allocated, or FALSE (which aborts the operation).

free is called to inform the ID manager that the object labeled with the given ID is about to go out of existence.

map is called to create or look-up IDs by string name (if supported by the ID manager). Returning TRUE (nonzero) in all cases means that the request succeeded (with a valid ID stored through *result*. There are four cases:

- *name != NULL* and *createflag == 1*: This requests mapping a string (e.g. a name in a graph file) into a new ID. If the ID manager can comply, then it stores the result and returns TRUE. It is then also responsible for being able to *print* the ID again as a string. Otherwise the ID manager may return FALSE but it must implement the following (at least for graph file reading and writing to work):
- *name == NULL* and *createflag == 1*: The ID manager creates a unique new ID of its own choosing. Although it may return FALSE if it does not support anonymous objects, but this is strongly discouraged (to support "local names" in graph files.)
- *name != NULL* and *createflag == 0*: This is a namespace probe. If the name was previously mapped into an allocated ID by the ID manager, then the manager must return this ID. Otherwise, the ID manager may either return FALSE, or may store any unallocated ID into result. (This is convenient, for example, if names are known to be digit strings that are directly converted into integer values.)
- *name == NULL* and *createflag == 0*: forbidden.

print is allowed to return a pointer to a static buffer; a caller must copy its value if needed past subsequent calls. *NULL* should be returned by ID managers that do not map names.

The *map* and *alloc* calls do not pass a pointer to the newly allocated object. If a client needs to install object pointers in a handle table, it can obtain them via new object callbacks.

IO DISCIPLINE

The I/O discipline provides an abstraction for the reading and writing of graphs.

```
struct Agiodisc_s {
  int     (*fread)(void *chan, char *buf, int bufsize);
  int     (*putstr)(void *chan, char *str);
  int     (*flush)(void *chan);   /* sync */
} ;
```

Normally, the **FILE** structure and its related functions are used for I/O. At times, though, an application may need to use a totally different type of character source. The associated state or stream information is provided by the *chan* argument to **agread** or **agwrite**. The discipline function *fread* and *putstr* provide the corresponding functions for read and writing.

CALLBACKS

An **Agcbdisc_t** defines callbacks to be invoked by Libcgraph when initializing, modifying, or finalizing graph objects. Disciplines are kept on a stack. Libcgraph automatically calls the methods on the stack, top-down. Callbacks are installed with **agpushdisc**, uninstalled with **agpopdisc**, and can be held pending or released via **agcallbacks**.

GENERIC OBJECTS

agroot takes any graph object (graph, subgraph, node, edge) and returns the root graph in which it lives. **agraphof** does the same, except it is the identity function on graphs and subgraphs. Note that there is no function to return the least subgraph containing an object, in part because this is not well-defined as nodes and edges may be in incomparable subgraphs.

agcontains(*g*,*obj*) returns non-zero if *obj* is a member of (sub)graph *g*. **agdelete**(*g*,*obj*) is equivalent to **agclose**, **agdelnode**, and **agdeledge** for *obj* being a graph, node or edge, respectively. It returns -1 if *obj* does not belong to *g*.

AGDATA, **AGID**, and **AGTYPE** are macros returning the specified fields of the argument object. The first is described in the **RECORDS** section above. The second returns the unique integer ID associated with the object. The last returns **AGRAPH**, **AGNODE**, and **AGEDGE** depending on the type of the object.

agnameof returns a string descriptor for the object. It returns the name of the node or graph, and the key of an edge. **agobjkind** is a synonym for **AGTYPE**.

ERROR REPORTING

The library provides a variety of mechanisms to control the reporting of errors and warnings. At present, there are basically two types of messages: warnings and errors. A message is only written if its type has higher priority than a programmer-controlled minimum, which is **AGWARN** by default. The programmer can set this value using **agseterr**, which returns the previous value. Calling **agseterr(AGMAX)** turns off the writing of messages.

The function **agerr** is the main entry point for reporting an anomaly. The first argument indicates the type of message. Usually, the first argument is **AGWARN** or **AGERR** to indicate warnings and errors, respectively. Sometimes additional context information is only available in functions calling the function where the error is actually caught. In this case, the calling function can indicate that it is continuing the current error by using **AGPREV** as the first argument. The remaining arguments to **agerr** are the same as the arguments to **printf**.

The functions **agwarningf** and **agerrorf** are shorthand for **agerr(AGWARN,...)** and **agerr(AGERR,...)**, respectively.

Some applications desire to directly control the writing of messages. Such an application can use the function **agseterrf** to register the function that the library should call to actually write the message. The previous error function is returned. By default, the message is written to **stderr**.

Errors not written are stored in a log file. The last recorded error can be retrieved by calling **aglasterr**. Unless the printing of error messages has been completely disabled by a call to **agseterr(AGMAX)**, standard error must not be wide-oriented, even if a user-provided error printing function is provided.

The function **agerrors** returns non-zero if errors have been reported.

EXAMPLE PROGRAM

```
#include <cgraph.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdio.h>

typedef struct {
  Agrec_t hdr;
  int x;
  int y;
```

```
  int z;
} mydata;

int main(int argc, char **argv) {
  Agraph_t *g;
    mydata      *p;

  if ((g = agread(stdin, NULL))) {
    int cnt = 0;
    Agsym_t *attr = NULL;
    while ((attr = agnxtattr(g, AGNODE, attr))) {
      cnt++;
    }
    printf("The graph %s has %d attributes\n", agnameof(g), cnt);

    // make the graph have a node color attribute, default is blue
    attr = agattr_text(g, AGNODE, "color", "blue");

    // create a new graph of the same kind as g
    Agraph_t *h = agopen("tmp", g->desc, NULL);

    // this is a way of counting all the edges of the graph
    cnt = 0;
    for (Agnode_t *v = agfstnode(g); v != NULL; v = agnxtnode(g, v)) {
      for (Agedge_t *e = agfstout(g, v); e != NULL; e = agnxtout(g, e)) {
        cnt++;
      }
    }

    // attach records to edges
    for (Agnode_t *v = agfstnode(g); v != NULL; v = agnxtnode(g, v)) {
      for (Agedge_t *e = agfstout(g, v); e != NULL; e = agnxtout(g, e)) {
        p = (mydata *)agbindrec(e, "mydata", sizeof(mydata), true);
        p->x = 27; // meaningless data access example
        ((mydata *)(AGDATA(e)))->y = 999; // another example
      }
    }
  }
  return 0;
}
```

EXAMPLE GRAPH FILES

```
digraph G {
    a -> b;
    c [shape=box];
    a -> c [weight=29,label="some text"];
    subgraph anything {
        /* the following affects only x,y,z */
        node [shape=circle];
        a; x; y -> z; y -> z;  /* multiple edges */
    }
}
```

```
strict graph H {
   n0 -- n1 -- n2 -- n0;  /* a cycle */
   n0 -- {a b c d};       /* a star */
   n0 -- n3;
   n0 -- n3 [weight=1];   /* same edge because graph is strict */
}
```

SEE ALSO

cdt(3)

BUGS

It is difficult to change endpoints of edges, delete string attributes or modify edge keys. The work-around is to create a new object and copy the contents of an old one (but new object obviously has a different ID, internal address, and object creation timestamp).

The API lacks convenient functions to substitute programmer-defined ordering of nodes and edges but in principle this can be supported.

The library is not thread safe.

AUTHOR

Stephen North, north@research.att.com, AT&T Research.