## NAME

**libcgraph** − abstract graph library

## SYNOPSIS

#include <graphviz/cgraph.h>

### TYPES

Agraph_t;
Agnode_t;
Agedge_t;
Agdesc_t;
Agdisc_t;
Agsym_t;

### 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            *agconcat(Agraph_t *g, void *channel, Agdisc_t *disc)
int         agwrite(Agraph_t *g, void *channel);
int                         agnnodes(Agraph_t *g),agnedges(Agraph_t *g);
void                agreadline(int line_no);
void                agsetfile(char *file_name);
int                         agisdirected(Agraph_t * g),agisundirected(Agraph_t * g),agisstrict(Agraph_t * g);
```

### 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),*agroot(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(Agnode_t *n);
Agnode_t    *agprvnode(Agnode_t *n);
Agnode_t    *aglstnode(Agnode_t *n);
int         agdelnode(Agraph_t *g, Agnode_t *n);
int                         agdegree(Agnode_t *n, int use_inedges, int use_outedges);
```

### EDGES

```
Agedge_t    *agedge(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(Agnode_t *n);
Agedge_t    *agnxtedge(Agedge_t *e, Agnode_t *n);
Agedge_t    *agfstin(Agnode_t *n);
Agedge_t    *agnxtin(Agedge_t *e);
Agedge_t    *agfstout(Agnode_t *n);
Agedge_t    *agnxtout(Agedge_t *e);
```

```
int         agdeledge(Agraph_t *g, Agedge_t *e);
```

**STRING ATTRIBUTES**

```
Agsym_t             *agattr(Agraph_t *g, int kind, char *name, 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);
```

**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);
int                 agcopyattr(void *, void *);
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**

```
Agcbdisc_t    *agpopdisc(Agraph_t *g);
void        agpushdisc(Agraph_t *g, Agcbdisc_t *disc);
void        agmethod(Agraph_t *g, void *obj, Agcbdisc_t *disc, int initflag);
```

**MEMORY**

```
void            *agalloc(Agraph_t *g, size_t request);
void            *agrealloc(Agraph_t *g, void *ptr, size_t oldsize, size_t newsize);
void            agfree(Agraph_t *g, void *ptr);
```

**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);
Agrec_t         *AGDATA(void *obj);
ulong           AGID(void *obj);
int                 AGTYPE(void *obj);
```

## 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).

## 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 32-bit 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.) **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. **agsetfile** and **agreadline** are helper functions that simply set the current file name and input line number for subsequent error reporting.

**agsubg** finds or creates a subgraph by name. A new subgraph is 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**).

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 **agdegree** return the sizes of node and edge sets of a graph. The **agdegree** returns the size of the edge set of a nodes, and takes flags to select in-edges, out-edges, or both.

An **Agdisc_t** defines callbacks to be invoked by libcgraph when initializing, modifying, or finalizing graph objects. (Casual users can ignore the following.) 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**.

(Casual users may ignore the following. When Libcgraph is compiled with Vmalloc (which is not the default), each graph has its own heap. Programmers may allocate application-dependent data within the same heap as the rest of the graph. The advantage is that a graph can be deleted by atomically freeing its entire heap without scanning each individual node and edge.

## NODES

A node is created by giving a unique string name or programmer defined 32-bit 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. If not found, if **createflag** is boolean true a new node is created and returned, otherwise a nil pointer is returned. **agidnode** allows a programmer to specify the node by a unique 32-bit 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 the 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 interchangable. 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 of subgraph for an edge between the given endpoints (with an optional multi-edge selector name) and returns it if found. Otherwise, if **createflag** is boolean true, a new edge is created and returned: otherwise a nil pointer is returned. If the **name** is (char*)0 then an anonymous internal value is generated. **agidedge** allows a programmer to create an edge by giving its unique 32-bit ID. **agfstin**, **agnxtint**, **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.

## INTERNAL ATTRIBUTES

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

## 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** is listed below, followed by primitives to operate on string attributes.

```
typedef struct Agsym_s {       /* symbol in one of the above dictionaries */
   Dtlink_t       link;
   char           *name;     /* attribute's name */
   char           *defval;   /* its default value for initialization */
   int            id;        /* its index in attr[] */
   unsigned char  kind;              /* referent object type */
   unsigned char  fixed;             /* immutable value */
} Agsym_t;
```

**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 NIL, 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. **agnxtattrP permits traversing the list of attributes of a given type. If NIL is passed as an argument it gets the first attribute, otherwise it returns the next one in succession or returns NIL 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). agsafeset is a convenience function that ensures the given attribute is declared before setting it locally on an object.**

**Note that Libcgraph performs its own storage management of strings. The caller does not need to dynamically allocate storage.**

## 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_s {
   Agrec_t                 header;
   /* programmer-defined fields follow */
} Agrec_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**. The **name** argument 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. **agdelrec** is the deletes records one at a time. **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 **AG_MTF_FALSE**, **AG_MTF_SOFT**, or **AG_MTF_HARD** accordingly. The **AG_MTF_SOFT** field is only a hint that decreases overhead in subsequent calls of **aggetrec**; **AG_MTF_HARD** guarantees that a lock was obtained. To release locks, use **AG_MTF_SOFT** or **AG_MTF_FALSE**. Use of this feature implies cooperation or at least isolation from other functions also using the move-to-front convention.

## DISCIPLINES

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

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

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

An ID allocator discipline allows a client to control assignment of IDs (uninterpreted 32-bit values) to objects, and possibly how they are mapped to and from strings.

```
struct Agiddisc_s {                 /* object ID allocator */
        void    *(*open)(Agraph_t *g);    /* associated with a graph */
        int             (*map)(void *state, int objtype, char *str, ulong *id, int createflag);
        int             (*alloc)(void *state, int objtype, ulong id);
        void    (*free)(void *state, int objtype, ulong id);
        char    *(*print)(void *state, int objtype, ulong id);
        void    (*close)(void *state);
} ;
```

open permits the ID discipline to initialize any data structures that maintains per individual graph. Its return value is then passed as the first argument 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 32 bit values.)

name == NULL and createflag == 0: forbidden.

print should return 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.

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

struct Agmemdisc_s {    /* memory allocator */
        void    *(*open)(void);             /* independent of other resources */
        void    *(*alloc)(void *state, size_t req);
        void    *(*resize)(void *state, void *ptr, size_t old, size_t req);
        void    (*free)(void *state, void *ptr);
        void    (*close)(void *state);
} ;
```

## EXAMPLE PROGRAM

```
#include <graphviz/cgraph.h>
typedef struct mydata_s {Agrec_t hdr; int x,y,z;} mydata;

main(int argc, char **argv)
{
   Agraph_t    *g;
   Agnode_t    *v;
   Agedge_t    *e;
   Agsym_t     *attr;
   Dict_t      *d
   int         cnt;
   mydata      *p;

   if (g = agread(stdin,NIL(Agdisc_t*))) {
                    cnt = 0; attr = 0;
                    while (attr = agnxtattr(g, AGNODE, attr)) cnt++;
                    printf("The graph %s has %d attributes0,agnameof(g),cnt);

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

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

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

      /* attach records to edges */
      for (v = agfstnode(g); v; v = agnxtnode(g,v))
```

```
        for (e = agfstout(g,v); e; e; = agnxtout(g,e)) {
          p = (mydata*) agbindrec(g,e,"mydata",sizeof(mydata),TRUE);
          p->x = 27;  /* meaningless data access example */
                                    ((mydata*)(AGDATA(e)))->y = 999; /* another example */
        }
      }
    }
```

## 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

Libcdt(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.

## AUTHOR

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