NAME

Agraph − abstract graph library

SYNOPSIS

#include <graphviz/agraph.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 *file, Agdisc_t *);
Agraph_t                *agconcat(Agraph_t *g, void *chan, Agdisc_t *disc)
int         agwrite(Agraph_t *g, void *file);
int                             agnnodes(Agraph_t *g),agnedges(Agraph_t *g);
```

SUBGRAPHS

```
Agraph_t    *agsubg(Agraph_t *g, char *name, int createflag);
Agraph_t    *agfstsubg(Agraph_t *g), agnxtsubg(Agraph_t *);
Agraph_t    *agparent(Agraph_t *g),*agroot(Agraph_t *g);
```

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);
int         agdelnode(Agraph_t *g, Agnode_t *n);
int         agrename(Agraph_t *g, Agnode_t *n, char *newname);
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    *agsubedge(Agraph_t *g, Agedge_t *e, int createflag);
int         agdeledge(Agraph_t *g, Agedge_t *e);

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);
```

FLATTENED LISTS

```
int         agflatten(Agraph_t *graph, int flag);
Agnode_t    *agfstn(Agraph_t *g), *agnxtn(Agnode_t *n);
Agedge_t    *agfout(Agnode_t *n), *agfin(Agnode_t *n), *agnxte(Agedge_t *e);
```

STRING ATTRIBUTES

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

RECORDS

```
void        *agnewrec(Agraph_t *g, void *obj, char *name, unsigned int size);
Agrec_t     *aggetrec(void *obj, char *name, int move_to_front);
int         agdelrec(Agraph_t *g, void *obj, char *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*);
char                *agnameof(void*);
int                         agisarootobj(void*);
Agrec_t             *AGDATA(void *obj);
ulong               AGID(void *obj);
int                         AGTYPE(void *obj);
```

DESCRIPTION

Libagraph supports graph programming by maintaining graphs in memory and reading and writing graph files. Graphs, nodes and edges may be attributed with programmer-defined records and string name-value pairs. Graphs are composed of nodes, edges, and nested subgraphs. Internally, Libagraph depends extensively on Libcdt (formerly Libdict) for set representation.

All of Libagraph's global symbols have the prefix **ag** (case varying).

GRAPHS

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. By default **data** points to runtime records containing application-dependent data, keyed by name (see Attributes). **desc** records if a graph is directed or undirected, and if it is strict or allows multi-edges and self-arcs.

agopen creates a new graph with the given name and graph kind descriptor (global values are **Agdirected**, **Agundirected**, **Agstrictdirected**, and **Agstrictundirected**). **agclose** deletes a graph, freeing all its associated storage. **agread** and **agwrite** perform file I/O (see Graph File Language bellow). **agsubg** creates a new subgraph, which always inherits the graph kind of its parent. The new subgraph is initially empty. Nested subgraph trees may be created. The name of a subgraph is interpreted only relative to the given parent graph. **agsubglist** returns a list (possibly empty) of subgraphs of a given graph.

By default, nodes are kept in ordered sets in **n_dict**, allowing efficient random access to insert, find, and delete nodes. Similarly the edges of each node are kept in ordered sets. The sets are maintained as splay

tree dictionaries. **agflatten** allows flattening trees into linked lists, which may thereafter be traversed very quickly without function calls for low overhead in critical sections of code. In this mode, sets are locked to prevent updates or random access searches, though it is still legal to call Libagraph to scan lists sequentially. The flag argument requests flattening and locking (if boolean true), or unlocking (if false). In-line functions or macros for list traversal are given below under Nodes and Edges. Note that flattening a graph does not automatically flatten its subgraphs.

agnnodes, **agnedges**, and **agdegree** return the cardinalities of node and edge sets. The latter takes flags to select in-edges, out-edges, or both.

Agdisc_t specifies callbacks invoked when initializing, modifying, or finalizinf graph objects. (Casual users can ignore the following.) Disciplines are kept on a stack. Libagraph automatically calls the methods on the stack, top-down. A method can obtain its data (closure) via aggetuserptr.

When Libagraph is compiled with Vmalloc, 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 identified uniquely by name and graph pointer. Node pointers are not unique— separate node structs are created per subgraph. Name pointers are unique, though, because each graph has its own shared string pool.

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. **agsubnode** takes an existing node as a template, usually to find or insert a node in a subgraph.

The default ordering of nodes is by order of creation (sequence). Internally, Libagraph switches between ID searching and sequence ordering as necessary. **agfstnode** and **agnxtnode** are the usual functions for scanning node lists. When node sets are flattened it is permissible to call **agfstnode** and **agnxtnode**, but conflicting attempts to insert, delete, or search for nodes cause a runtime error.

EDGES

An abstract edge is represented by two edge structs. There is one pointing to each terminal node, and residing in an edge list of the opposite node. The object tag distinguishes between these otherwise symmetric records, to allow obtaining head and tail. If a graph has multi-edges between the same nodes, the name field serves 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. **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. In traversing lists, e->node always points to the ‘‘other’’ node of the edge, To resolve ambiguity between in- and out-edge structs, **aghead** and **agtail** are macros or inline functions to find endpoints by checking object tags. **agopp** returns the ‘‘opposite’’ edge struct. Similarly **agfout**, **agfin**, and **agnedge** operate on flattened edge lists.

STRING 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 are handled automatically in reading and writing graph files. Uninterpreted records are ignored; any desired conversion must be coded explicitly by application programmers.

A string attribute is identified by name and by an internal symbol table entry (**Agsym_t**) created by Libagraph. 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[] */
} 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.

agdictof returns a Libdict set of all the attributes of a given kind. **agdictsym** is a utility function that finds an entry in one of these dictionary sets.

agget and **agset** read and update string attributes. The first argument should be a graph, node, or edge struct pointer. **agxset** and **agxset** take a symbol table entry reference instead of a name, to avoid the cost of looking up attribute names inside loops. Note that Libagraph performs its own storage management of strings. The calling program 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 have a common header as shown below.

```
typedef struct Agrec_s {
   char              *name;
   struct Agrec_s    *next;
   /* programmer-defined follows */
} Agrec_t;
```

Records are created and managed by Libagraph. In each graph, node, or edge, **data** points to a circular list of records. The **name** field distinguishes various types of records, and is programmer defined (Libagraph reserves the prefix **_ag**). **next** stores the list pointers. The remainder of a record may contain application-dependent fields. **agnewrec** creates one new record of the given size and attaches it to the given object (graph, node, or edge). **agdelrec** is the corresponding function to delete records. **aggetrec** finds a record with the given name.

To allow referencing application-dependent data without function calls or linear search, Libagraph allows setting and locking the **data** field of a graph, node, or edge on a particular record. 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.

A cast (generally using a macro or inline function) is then needed to convert the **data** pointer to an appropriate programmer-defined type.

DISCIPLINES

Programmer-defined disciplines customize certain resources- ID namespace, memory, and I/O - needed by Libagraph. 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 Libagraph 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/agraph.h>
typedef struct mydata_s {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*))) {
        /* dtsize() is a Libdict primitive */
      fprintf(stderr,"%s has %d node attributes0,
         dtsize(agdictof(g,AGNODE)));
      attr = agattr(g,AGNODE,"color","blue");

      /* create a new graph */
      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++;

      /* using inline functions or macros, attach records to edges */
      agflatten(g);
      for (v = agfstn(g); v; v = agnxtn(v))
         for (e = agfout(v); e; e; = agnxte(e)) {
            p = (mydata*) agnewrec(g,e,"mydata",sizeof(mydata));
            p->x = 27;  /* meaningless 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

The root graph **name** is treated as a comment.

There is no way to delete string attributes or modify edge keys.

Strings and uninterpreted records could be treatly more uniformly.

AUTHOR

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