# Cgraph Tutorial

Stephen C. North
scnorth@gmail.com

Emden R. Gansner
erg@graphviz.com

February 7, 2014

# Contents

# 1 Introduction

Cgraph is a C library for graph programming. It defines data types and operations for graphs comprised of attributed nodes, edges and subgraphs. Attributes may be string name-value pairs for convenient file I/O, or internal C data structures for efficient algorithm implementation.

Cgraph is aimed at graph representation; it is not an library of higher-level algorithms such as shortest path or network flow. We envision these as higher-level libraries written on top of Cgraph. Efforts were made in Cgraph's design to strive for time and space efficiency. The basic (unattributed) graph representation takes 104 bytes per node and 64 bytes per edge, so storage of graphs with millions of objects is reasonable. For attributed graphs, Cgraph also maintains an internal shared string pool, so if all the nodes of a graph have `color=red`, only one copy of `"color"` and `"red"` are made. There are other tricks that experts can exploit for flat-out coding efficiency. For example, there are ways to inline the instructions for edge list traversal and internal data structure access.

Cgraph uses Phong Vo's dictionary library, libcdt, to store node and edge sets. This library provides a uniform interface to hash tables and splay trees, and its API is usable for general programming (such as storing multisets, hash tables, lists and queues) in Cgraph programs.

**Notation** In the following, we use `TRUE` to denote a non-zero value, and `FALSE` to denote zero.

# 2 Graph Objects

Almost all Cgraph programming can be done with pointers to these data types:

- `Agraph_t`: a graph or subgraph
- `Agnode_t`: a node from a particular graph or subgraph
- `Agedge_t`: an edge from a particular graph or subgraph
- `Agsym_t`: a descriptor for a string-value pair attribute
- `Agrec_t`: an internal C data record attribute of a graph object

Cgraph is responsible for its own memory management; allocation and deallocation of Cgraph data structures is always done through Cgraph calls.

# 3 Graphs

A top-level graph (also called a root graph) defines a universe of nodes, edges, subgraphs, data dictionaries and other information. A graph has a name and two properties: whether it is directed or undirected, and whether it is strict (multi-edges are forbidden). [1]

[1] It is possible to specify that a graph is simple (neither multi-edges nor loops), or can have multi-edges but not loops.

Note that nodes, edges and subgraphs exists in exactly one root graph. They cannot be used independently of that graph, or attached to another root graph.

The following examples use the convention that `G` and `g` are `Agraph_t*` (graph pointers), `n`, `u`, `v`, `w` are `Agnode_t*` (node pointers), and `e`, `f` are `Agedge_t*` (edge pointers).

To make a new, empty top-level directed graph:

```
Agraph_t    *g;
g = agopen("G", Agdirected, NULL);
```

The first argument to `agopen` is any string, and is not interpreted by Cgraph, except it is recorded and preserved when the graph is written as a file.[2] The second argument indicates the type of graph, and should be one of `Agdirected`, `Agstrictdirected`, `Agundirected`, or `Agstrictundirected`. The third argument is an optional pointer to a collection of methods for overriding certain default behaviors of Cgraph, and in most situations can just be `NULL`.

You can get the name of a graph by `agnameof(g)`, and you can get its properties by the predicate functions `agisdirected(g)` and `agisstrict(g)`.

You can also construct a new graph by reading a file:

```
g = agread(stdin,NULL);
```

Here, the graph's name, type and contents including attributes depend on the file contents. (The second argument is the same optional method pointer mentioned above for `agopen`).

Sometimes it is convenient to have the graph represented concretely as a character string `str`. In this case, the graph can be created using:

```
g = agmemread (str);
```

You can write a representation of a graph to a file:

```
g = agwrite(g,stdout);
```

`agwrite` creates an external representation of a graph's contents and attributes (except for internal attributes), that can later be reconstructed by calling `agread` on the same file.[3]

`agnnodes(g)`, `agnedges(g)` and `agnsubg(g)` return the count of nodes, edges and (immediate) subgraphs in a graph (or subgraph).

To delete a graph and its associated data structures, freeing their memory, one uses:

```
agclose(g);
```

Finally, there is an interesting if obscure function to concatenate the contents of a graph file onto an existing graph, as shown here.

```
g = agconcat(g,stdin,NULL);
```

[2] An application could, of course, maintain its own graph catalogue using graph names.

[3] It is the application programmer's job to convert between internal attributes to external strings when graphs are read and written, if desired. This seemed better than inventing a complicated way to automate this conversion.

# 4 Nodes

In Cgraph, a node is usually identified by a unique string name and a unique internal integer ID assigned by Cgraph. (For convenience, you can also create "anonymous" nodes by giving `NULL` as the node name.) A node also has in- and out-edge sets, even in undirected graphs.

Once you have a graph, you can create or search for nodes this way:

```
Agnode_t    *n;
n = agnode(g,"node28",TRUE);
```

The first argument is a graph or subgraph in which the node is to be created. The second is the name (or NULL for anonymous nodes.) When the third argument is `TRUE`, the node is created if it doesn't already exist. When it's `FALSE`, as shown below, then Cgraph searches to locate an existing node with the given name, returning NULL if none is found.

```
n = agnode(g,"node28",FALSE);
```

The function `agdegree(g, n, in, out)` gives the degree of a node in (sub)graph `g`, where `in` and `out` select the edge sets.

- `agdegree(g,n,TRUE,FALSE)` returns the in-degree.
- `agdegree(g,n,FALSE,TRUE)` returns the out-degree.
- `agdegree(g,n,TRUE,TRUE)` returns their sum.

The function `agcountuniqedges` is identical to `agdegree` except when the last two arguments are both `TRUE`. In this case, a loop is only counted once.

`agnameof(n)` returns the printable string name of a node. Note that for various reasons this string may be a temporary buffer that can be overwritten by subsequent calls. Thus, the usage

```
printf("%s %s\n",agnameof(agtail(e)),agnameof(aghead(e)));
```

is unsafe because the buffer may be overwritten when the arguments to printf are being computed.

A node can be deleted from a graph or subgraph by `agdelnode(g,n)`.

# 5 Edges

An edge is a node pair: an ordered pair in a directed graph, unordered in an undirected graph. For convenience there is a common edge data structure for both kinds and the endpoints are the fields `tail` and `head`. [4]

Because an edge is implemented as an edge pair, there are two valid pointers to the same edge, so simple pointer comparison does not work for edge equality. The function `ageqedge(Agedge_t *e0, Agedge_t *e1)`

[4] When an edge is created, the first node will be used as the tail node, and the second node as the head.

evaluates to true if the two pointers represent the same abstract edge, and should usually be used for edge comparisons.

An edge is made using

```
Agnode_t    *u, *v;
Agedge_t    *e;

/* assume u and v are already defined */
e = agedge(g,u,v,"e28",TRUE);
```

`u` and `v` must belong to the same graph or subgraph for the operation to succeed. The "name" of an edge (more correctly, identifier) is treated as a unique identifier for edges between a particular node pair. That is, there can only be at most one edge with name `e28` between any given `u` and `v`, but there can be many other edges `e28` between other nodes.

`agtail(e)` and `aghead(e)` return the endpoints of `e`. If `e` is created as in the call to `agedge` above, `u` will be the tail node and `v` will be the head node. This holds true even for undirected graphs.

The value `e->node` is the "other" endpoint with respect to the node from which e was obtained. That is, if `e` is an out-edge of node `n` (equivalently, `n` is the tail of `e`), then `e->node` is the head of `e`. A common idiom is:

```
for (e = agfstout(g,n); e; e = agnxtout(g,e))
  /* do something with e->node */
```

`agedge` can also search for edges:

```
/* finds any u,v edge */
e = agedge(g,u,v,NULL,FALSE);
/* finds a u,v edge with name "e8" */
e = agedge(g,u,v,"e8",FALSE);
```

In an undirected graph, an edge search will consider the given vertices as both tail and head nodes.

An edge can be deleted from a graph or subgraph by `agdeledge(g,e)`. The `agnameof` function can be used to get the "name" of an edge. Note that this will be the identifier string provided at edge creation. The names of the head and tail nodes will not be part of the string. In addition, it returns NULL for anonymous edges.

# 6 Traversals

Cgraph has functions for iterating over graph objects. For example, we can scan all the edges of a graph (directed or undirected) by the following:

```
for (n = agfstnode(g); n; n = agnxtnode(g,n)) {
  for (e = agfstout(g,n); e; e = agnxtout(g,e)) {
    /* do something with e */
  }
}
```

The functions `agfstin(g,n)` and `afnxtin(g,e)` are provided for walking in-edge lists.

In the case of a directed edge, the meanings of "in" and "out" are obvious. For undirected graphs, Cgraph assigns an orientation based on the analogous order of the two nodes when the edge is created.

To visit all the edges of a node in an undirected graph:

```
for (e = agfstedge(g,n); e; e = agnxtedge(g,e,n))
  /* do something with e */
```

Be careful if your code deletes an edge or node during the traversal, as then the object will no longer be valid to get the next object. This is typically handled by code like:

```
for (e = agfstedge(g,n); e; e = f) {
  f = agnxtedge(g,e,n))
  /* delete e */
}
```

Traversals are guaranteed to visit the nodes of a graph, or edges of a node, in their order of creation in the root graph (unless we allow programmers to override object ordering, as mentioned in section 16).

# 7 External Attributes

Graph objects may have associated string name-value pairs. When a graph file is read, Cgraph's parser takes care of the details of this, so attributes can just be added anywhere in the file. In C programs, values must be declared before use.

Cgraph assumes that all objects of a given kind (graphs/subgraphs, nodes, or edges) have the same attributes - there's no notion of subtyping within attributes. Information about attributes is stored in data dictionaries. Each graph has three (for graphs/subgraphs, nodes, and edges) for which you'll need the predefined constants AGRAPH, AGNODE and AGEDGE in calls to create, search and walk these dictionaries.

Thus, to create an attribute for nodes, one uses:

```
Agsym_t *sym;
sym = agattr(g,AGNODE,"shape","box");
```

If this succeeds, `sym` points to a descriptor for the newly created (or updated) attribute. (Thus, even if `shape` was previously declared and had some other default value, it would be set to `box` by the above.)

By using a NULL pointer as the value, you can use the same function to search the attribute definitions of a graph.

```
sym = agattr(g,AGNODE,"shape",0);
if (sym)
  printf("The default shape is %s.\n",sym->defval);
```

If you have the pointer to some graph object, you can also use the function `agattrsym`.

```
Agnode_t* n;
Agsym_t* sym = agattrsym (n,"shape");
if (sym)
  printf("The default shape is %s.\n",sym->defval);
```

Both functions return NULL if the attribute is not defined.

Instead of looking for a particular attribute, it is possible to iterate over all of them:

```
sym = 0;    /* to get the first one */
while (sym = agnxtattr(g,AGNODE,sym)
  printf("%s = %s\n",sym->name,sym->defval);
```

Assuming an attribute already exists for some object, its value can be obtained or set using its string name or its `Agsym_t` descriptor. To use the string name, we have:

```
str = agget(n,"shape");
agset(n,"shape","hexagon");
```

If an attribute will be referenced often, it is faster to use its descriptor as an index, as shown here:

```
Agsym_t *sym = agattr(g,AGNODE,"shape","box");
str = agxget(n,sym);
agxset(n,sym,"hexagon");
```

Cgraph provides two helper functions for dealing with attributes. The function `agsafeset(void *obj, char *nam` first checks that the attribute has been defined, defining it with the default value `def` if not. It then uses `value` as the specific value assigned to `obj`.

It is sometimes useful to copy all of the values from one object to another. This can be easily done using `agcopyattr(void *src, void* tgt)`. This assumes that the source and target are the same type of graph objects, and that the attributes of `src` have already been defined for `tgt`. If `src` and `tgt` belong to the same root graph, this will automatically be true.

# 8 Internal Attributes

It would be possible to do everything using just string-valued attributes. In general, though, this will be too inefficient. To deal with this, each graph object (graph, node or edge) may have a list of associated internal data records. The layout of each such record is programmer-defined, except each must have an `Agrec_t` header. The records are allocated through Cgraph. For example:

```
typedef struct mynode_s {
  Agrec_t     h;
  int         count;
} mynode_t;

mynode_t    *data;
Agnode_t    *n;
n = agnode(g,"mynodename",TRUE);
data = (mynode_t*)agbindrec(n,"mynode_t",
  sizeof(mynode_t),FALSE);
data->count = 1;
```

In a similar way, `aggetrec` searches for a record, returning a pointer to the record if it exists and NULL otherwise; `agdelrec` removes a record from an object.

Although each graph object may have its own unique, individual collection of records, for convenience, there are functions that update an entire graph by allocating or removing the same record from all nodes, edges or subgraphs at the same time. These functions are:

```
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);
```

Note that in addition to `agdelrec` and `agclean`, records are removed and their storage freed when their associated graph object is deleted. Only the record data structure is freed. If the application has attached any additional heap memory to a record, it is the responsibility of the application to handle this before the actual record is deleted.

For further efficiency, there is a way to "lock" the data pointer of a graph object to point to a given record. This can be done by using `TRUE` as the last argument in `agbindrec`, `aginit` or `aggetrec`. If this is done, in the above example we could then simply cast this pointer to the appropriate type for direct (un-typesafe) access to the data.

```
(mydata_t*) (n->base.data)->count = 1;
```

Typically, it is convenient to encapsulate this access using macros. For example, we may have:

```
#define ND_count(n) (((mydata_t*)(AGDATA(n)))->count)
ND_count(n) = 1;
```

As this is unsafe if the record was not allocated for some object, it is good form to two versions of the macro:

```
#ifdef DEBUG
#define ND_count(n) \
  (assert(aggetrec(n,"mynode_t",1)),((mynode_t*)(AGDATA(n)))->count)
#else
#define ND_count(n) (((mydata_t*)(AGDATA(n)))->count)
#endif
```

# 9 Subgraphs

Subgraphs are an important construct in Cgraph. They are intended for organizing subsets of graph objects and can be used interchangeably with top-level graphs in almost all Cgraph functions.

A subgraph may contain any nodes or edges of its parent. (When an edge is inserted in a subgraph, its nodes are also implicitly inserted if necessary. Similarly, insertion of a node or edge automatically implies insertion in all containing subgraphs up to the root.) Subgraphs of a graph form a hierarchy (a tree). Cgraph has functions to create, search, and iterate over subgraphs.

For example,

```
Agraph_t *g, *h;

  /* search for subgraph by name */
h = agsubg(g,"mysubgraph",FALSE);
if (!h)
    /* create subgraph by name */
  h = agsubg(g,"mysubgraph",TRUE);

for (h = agfstsubg(g); h; h = agnxtsubg(h)) {
    /* agparent is up one level */
  assert (g == agparent(h));
  /* Use subgraph h */
}
```

The function `agparent` returns the (sub)graph immediately containing the argument subgraph. The iteration done using `agfstsubg` and `agnxtsubg` returns only immediate subgraphs. To find subgraphs further down the hierarchy requires a recursive search.

It is not uncommon to want to populate a subgraph with nodes and edges that have already been created. This can be done using the functions `agsubnode` and `agsubedge`,

```
Agnode_t *agsubnode(Agraph_t *g, Agnode_t *n, int create);
Agedge_t *agsubedge(Agraph_t *g, Agedge_t *e, int create);
```

which take a subgraph, and an object from another subgraph of the same graph (or possibly a top-level object) and add it to the argument subgraph if the `create` flag is `TRUE`. It is also added to all enclosing subgraphs, if necessary. If the `create` flag is `FALSE`, then the request is only treated as a search and returns NULL for failure.

A subgraph can be removed by `agdelsubg(g,subg)` or by `agclose(subg)`.

# 10 Utility Functions and Macros

For convenience, Cgraph provides some polymorphic functions and macros that apply to all Cgraph objects. (Most of these functions could be implemented in terms of others already described, or by accessing fields in the `Agobj_t` base object.

- `AGTYPE(obj)`: object type - AGRAPH, AGNODE, or AGEDGE
- `AGID(obj)`: internal object ID (an unsigned long)
- `AGSEQ(obj)`: object creation timestamp (an integer)
- `AGDATA(obj)`: data record pointer (an `Agrec_t*`)

Other useful functions include:

```
  /* Returns root graph of obj */
Agraph_t *agroot(void* obj);
  /* Returns root graph of obj or obj if a (sub)graph */
Agraph_t *agraphof(void* obj);
  /* True of obj belongs to g */
int agcontains(Agraph_t *g, void *obj);
  /* Delete obj from the (sub)graph */
int agdelete(Agraph_t *g, void *obj);
  /* Synonym of AGTYPE */
int agobjkind(void *obj);
```

A root graph `g` will always have

```
g == agroot(g) == agraphof(g)
```

# 11 Error Handling

Cgraph provides some basic error handling functions, hampered by the lack of exceptions in C. At present, there are basically two types of anomalies: warnings and errors.

To report an anomaly, one uses:

```
typedef enum { AGWARN, AGERR, AGMAX, AGPREV
} agerrlevel_t;

int agerr(agerrlevel_t level, const char *fmt, ...);
```

The `agerr` function has a `printf`-interface, with the first argument indicating the severity of the problem. A message is only written if its severity is higher 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.

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 function `agwarningf` is shorthand for `agerr(AGWARN,...)`; similarly, `agerrorf` is shorthand for `agerr(AGERR,...)`.

Some applications desire to directly control the writing of messages. Such an application can use the function `agseterrf` to register a function that the library should call to actually write the message:

```
typedef int (*agusererf) (char*);

agusererrf agseterrf(agusererrf);
```

The previous error function is returned. By default, the messages are written to `stderr`. Errors not written are stored in a log file. The last recorded error can be retrieved by calling `aglasterr`. The function `agerrors` returns the maximum severity reported to `agerr`. The function `agreseterrors` is identical, except it also resets the error level as though no errors have been reported.

# 12 Strings

As mentioned, Cgraph maintains a reference-counted shared string pool for each graph. As there are often many identical strings used in a graph, this helps to reduce the memory overhead.

```
char *agstrdup(Agraph_t *, char *);
char *agstrbind(Agraph_t *, char*);
int agstrfree(Agraph_t *, char *);
```

```
char *agstrdup_html(Agraph_t *, char *);
int aghtmlstr(char *);

char *agcanonStr(char *str);
char *agstrcanon(char *, char *);
char *agcanon(char *, int);
```

Cgraph has functions to directly create and destroy references to shared strings. As with any reference-counted object, you can use these as ordinary strings, though the data should not be modified. If you need to store the pointer in some data structure, you should call `agstrdup(g,s)`. This will create a new copy of `s` if necessary, and increment the count of references, ensuring that the string will not be freed by some other part of the program while you are still using it. Since `agstrdup` makes a copy of the string, the argument string can use temporary storage.

A call to `agstrfree(g,s)` decrements the reference count and, if this becomes zero, frees the string. The function `agstrbind(g,s)` checks if a shared string identical to `s` exists and, if so, returns it. Otherwise, it returns NULL.

The DOT language supports a special type of string, containing HTML-like markups. To make sure that the HTML semantics are used, the application should call `agstrdup_html(g,s)` rather than `agstrdup(g,s)`. The following code makes sure the node's label will be interpreted as an HTML-like string and appear as a table. If `agstrdup` were used, the label would appear as the literal string `s`.

```
Agraph_t* g;
Agnode_t* n;
char* s = "<TABLE><TR><TD>one cell</TD></TR></TABLE>";
agset (n, "label", agsgtrdup_html (g,s));
```

The predicate `aghtmlstr` returns `TRUE` if the argument string is marked as an HTML-like string. **N.B.** This is only valid if the argument string is a shared string. HTML-like strings are also freed using `agstrfree`.

The DOT language uses various special characters and escape sequences. When writing out strings as concrete text, it is important that these lexical features are used in order that the string can be re-read as DOT and be interpreted correctly. Cgraph provides three functions to handle this. The simplest is `agcanonStr(s)`. It returns a pointer to a canonical version of the input string. It uses an internal buffer, so the returned string should be written or copied before another call to `agcanonStr`. `agstrcanon(s,buf)` is identical, except the calling function also supplies a buffer where the canonical version may be written. An application should only use the returned pointer, as it is possible the buffer will not be used at all. The buffer needs to be large enough to hold the canonical version. Normally, an expansion of `2*strlen(s)+2` is sufficient.

Both `agcanonStr` and `agstrcanon` assume that string argument is a shared string. For convenience, the library also provides the function `agcanon(s,h)`. This is identical to `agcanonStr(s)`, except `s` can be any string. If `h` is `TRUE`, the canonicalization assumes `s` is an HTML-like string.

# 13 Expert-level Tweaks

## 13.1 Callbacks

There is a way to register client functions to be called whenever graph objects are inserted into or deleted from a graph or subgraph, or have their string attributes modified. The arguments to the callback functions for insertion and deletion (an `agobjfn_t`) are the containing (sub)graph, the object and a pointer to a piece of state data supplied by the application. The object update callback (an `agobjupdfn_t`) also receives the data dictionary entry pointer for the name-value pair that was changed. The graph argument will be the root graph.

```
typedef void (*agobjfn_t)(Agraph_t*, Agobj_t*, void*);
typedef void (*agobjupdfn_t)(Agraph_t*, Agobj_t*,
              void*, Agsym_t*);

struct Agcbdisc_s {
  struct {
    agobjfn_t       ins;
    agobjupdfn_t    mod;
    agobjfn_t       del;
  } graph, node, edge;
};
```

Callback functions are installed by `agpushdisc`, which also takes a pointer to the client data structure `state` that is later passed to the callback function when it is invoked.

```
agpushdisc(Agraph_t *g, Agcbdisc_t *disc, void *state);
```

Callbacks are removed by `agpopdisc`, which deletes a previously installed set of callbacks anywhere in the stack. This function returns zero for success. (In real life, this function isn't used much; generally callbacks are set up and left alone for the lifetime of a graph.)

```
int   agpopdisc(Agraph_t *g, Agcbdisc_t *disc);
```

The default is for callbacks to be issued synchronously, but it is possible to hold them in a queue of pending callbacks to be delivered on demand. This feature is controlled by the interface:

```
  /* returns previous value */
int     agcallbacks(Agraph_t *g, int flag);
```

If the flag is zero, callbacks are kept pending. If the flag is one, pending callbacks are immediately issued, and the graph is put into immediate callback mode. (Therefore the state must be reset via `agcallbacks` if they are to be kept pending again.)

**N.B.**: it is a small inconsistency that Cgraph depends on the client to maintain the storage for the callback function structure. (Thus it should probably not be allocated on the dynamic stack.) The semantics of `agpopdisc` currently identify callbacks by the address of this structure so it would take a bit of reworking to fix this. In practice, callback functions are usually passed in a static struct.

## 13.2 Disciplines

A graph has an associated set of methods ("disciplines") for file I/O, memory management and graph object ID assignment.

```
typedef struct {
  Agmemdisc_t                    *mem;
  Agiddisc_t                     *id;
  Agiodisc_t                     *io;
} Agdisc_t;
```

A pointer to an `Agdisc_t` structure is used as an argument when a graph is created or read using `agopen`, `agread` and `agconcat`. If the pointer is NULL, the default Cgraph disciplines are used. An application can pass in its own disciplines to override the defaults. Note that it doesn't need to provide disciplines for all three fields. If any field is the NULL pointer, Cgraph will use the default discipline for that task.

The default disciplines are also accessible by name.

```
Agmemdisc_t AgMemDisc;
Agiddisc_t  AgIdDisc;
Agiodisc_t  AgIoDisc;
Agdisc_t    AgDefaultDisc;
```

This is useful because, unlike with `Agdisc_t`, all of the fields of specific disciplines must be non-NULL.

Cgraph copies the three individual pointers. Thus, these three structures must remain allocated for the life of the graph, though the `Agdisc_t` may be temporary.

### 13.2.1 Memory management

The memory management discipline allows calling alternative versions of malloc, particularly, Vo's Vmalloc, which offers memory allocation in arenas or pools. The benefit is that Cgraph can allocate a graph and its objects within a shared pool, to provide fine-grained tracking of its memory resources and the option of freeing the entire graph and associated objects by closing the area in constant time, if finalization of individual graph objects isn't needed.[5]

[5] This could be fixed.

```
typedef struct Agmemdisc_s {
  void *(*open)(void);
  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);
} Agmemdisc_t;
```

When a graph is created, but before any memory is allocated, Cgraph calls the discipline's `open` function. The application should then perform any necessary initializations of its memory system, and return a pointer to any state data structure necessary for subsequent memory allocation. When the graph is closed, Cgraph will call the discipline's `close` function, passing it the associated state, so the application can finalize and free any resources involved.

The other three discipline functions `alloc`, `resize` and `free` should have semantics roughly identical to the standard C library functions `malloc`, `realloc` and `free`. The main difference is that any new memory returned by `alloc` and `resize` should be zeroed out, and that `resize` is passed the old buffer size in addition to the requested new size. In addition, they all take the memory state as the first argument.

To simplify the use of the memory discipline, Cgraph provides three wrapper functions that hide the task of obtaining the memory state. These are the same functions Cgraph uses to handle memory for a graph.

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

### 13.2.2 I/O management

The I/O discipline is probably the most frequently used of the three, as the I/O requirements of applications vary widely.

```
typedef struct Agiodisc_s {
  int (*afread)(void *chan, char *buf, int bufsize);
  int (*putstr)(void *chan, char *str);
  int (*flush)(void *chan);
} Agiodisc_t ;
```

The default I/O discipline uses stdio and the FILE structure for reading and writing. The functions `afread`, `putstr` and `flush` should have semantics roughly equivalent to `fread`, `fputs` and `fflush`, with the obvious permutation of arguments.

The implementation of the `agmemread` function of Cgraph provides a typical example of using a tailored I/O discipline. The idea is to read a graph from a given string of characters. The implementation of the function

is given below. The `rdr_t` provides a miniature version of FILE, providing the necessary state information. The function `memiofread` fills the role of `afread` using the state provided by `rdr_t`. The `agmemread` puts everything together, creating the needed state using the argument string, and constructing a discipline structure using `memiofread` and the defaults. It then calls `agread` with the state and discipline to actually create the graph.

```
typedef struct {
    const char *data;
    int len;
    int cur;
} rdr_t;

static int memiofread(void *chan, char *buf, int bufsize)
{
    const char *ptr;
    char *optr;
    char c;
    int l;
    rdr_t *s;

    if (bufsize == 0) return 0;
    s = (rdr_t *) chan;
    if (s->cur >= s->len)
        return 0;
    l = 0;
    ptr = s->data + s->cur;
    optr = buf;
    do {
        *optr++ = c = *ptr++;
        l++;
    } while (c && (c != '\n') && (l < bufsize));
    s->cur += l;
    return l;
}

static Agiodisc_t memIoDisc = {memiofread, 0, 0};

Agraph_t *agmemread(const char *cp)
{
    rdr_t rdr;
    Agdisc_t disc;
    Agiodisc_t memIoDisc;

    memIoDisc.putstr = AgIoDisc.putstr;
    memIoDisc.flush = AgIoDisc.flush;
    rdr.data = cp;
    rdr.len = strlen(cp);
    rdr.cur = 0;
```

```
    disc.mem = NULL;
    disc.id = NULL;
    disc.io = &memIoDisc;
    return agread (&rdr, &disc);
}
```

---

### 13.2.3 ID management

Graph objects (nodes, edges, subgraphs) use an uninterpreted long integer value as keys. The ID discipline gives the application control over how these keys are allocated, and how they are mapped to and from strings. The ID discipline makes it possible for a Cgraph client control this mapping. For instance, in one application, the client may create IDs that are pointers into another object space defined by a front-end interpreter. In general, the ID discipline must provide a map between internal IDs and external strings.

```
typedef unsigned long ulong;

typedef struct Agiddisc_s {
  void *(*open)(Agraph_t *g, Agdisc_t*);
  long (*map)(void *state, int objtype, char *name,
              ulong *id, int createflag);
  long (*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);
  void (*idregister) (void *state, int objtype, void *obj);
} Agiddisc_t;
```

The `open` function 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. When the graph is closed, the discipline's `close` function is called to allow for finalizing and freeing any resources used.

The `alloc` and `free` functions explicitly create and destroy IDs. The former is used by Cgraph to see if the given ID is available for use. If it is available, the function should allocate it and return `TRUE`; otherwise, it should return `FALSE`. If it is not available, the calling function will abort the operation. `free` is called to inform the ID manager that the object labeled with the given ID is about to be deleted.

If supported buy the ID discipline, the `map` function is called to convert a string name into an ID for a given object type (AGRAPH, AGNODE, or AGEDGE), with an optional flag that tells if the ID should be allocated if it does not already exist.

There are four cases:

`name` **&&** `createflag` Map the string (e.g., a name in a graph file) into an ID. If the ID manager can comply, then it stores the result in the `id` parameter 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 case, at least in order for the reading and writing of graph files to work.

**!**`name` **&&** `createflag` Create a unique new ID. It may return FALSE if it does not support anonymous objects, but this is strongly discouraged in order for Cgraph to support "local names" in graph files.

`name` **&& !**`createflag` Test if `name` has already been mapped and allocated. If so, the ID should be returned in the `id` parameter. 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` **&& !**`createflag` Never used.

The `print` function is called to convert an internal ID back to a string. It 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.

Note that the `alloc` and `map` functions do not receive pointers to any newly created object. If a client needs to install object pointers in a handle table, it can obtain them via new object callbacks (Section 13.1).

To make this mechanism accessible, Cgraph provides functions to create objects by ID instead of external name:

```
Agnode_t *agidnode(Agraph_t *g, unsigned long id, int create);
Agedge_t *agidedge(Agraph_t *g, Agnode_t *t, Agnode_t *h,
                   unsigned long id, int create);
Agraph_t *agidsubg(Agraph_t *g, unsigned long id, int create);
```

Note that, with the default ID discipline, these functions return NULL.

## 13.3 Flattened node and edge lists

For random access, nodes and edges are usually stored in splay trees. This adds a small but noticeable overhead when traversing the "lists." For flat-out efficiency, there is a way of linearizing the splay trees in which node and edge sets are stored, converting them into flat lists. After this they can be traversed very quickly. The function `agflatten(Agraph_t *g, int flag)` will flatten the trees if `flag` is `TRUE`, or reconstruct the trees if `flag` is `FALSE`. Note that if any call adds or removes a graph object, the corresponding list is automatically returned to its tree form.

The library provides various macros to automate the flattening and simplify the standard traversals. For example, the following code performs the usual traversal over all out-edges of a graph:

```
Agnode_t* n;
Agedge_t* e;
```

```
  Agnoderef_t* nr;
  Agedgeref_t* er;
  for (nr = FIRSTNREF(g); nr; nr = NEXTNREF(g,nr)) {
    n = NODEOF(nr);
    /* do something with node n */
    for (er = FIRSTOUTREF(g,nr); er; er = NEXTEREF(g,er)) {
      e = EDGEOF(er);
      /* do something with edge e */
    }
  }
```

Compare this with the code in shown on page 7.

## 14 Related Libraries

**Libgraph** is a predecessor of Cgraph and is now considered obsolete. All of the Graphviz code is now written using Cgraph. As some older applications using libgraph may need to be converted to Cgraph, we note some of the main differences.

A key difference between the two libraries is the handling of C data structure attributes. Libgraph hard-wires these at the end of the graph, node and edge structuress. That is, an application programmer defines the structs `graphinfo`, `nodeinfo` and `edgeinfo` before including `graph.h`, and the library inquires of the size of these structures at runtime so it can allocate graph objects of the proper size. Because there is only one shot at defining attributes, this design creates an impediment to writing separate algorithm libraries. The dynamic `Agrec_t` structures, described in Section 8, allows each algorithm to attach its own required data structure.

As noted in Section 5, edges are implemented slightly differently in the two libraries, so comparison of edge pointers for equality should be replaced with `ageqedge` unless you are certain that the two pointers have consistent types. As an example where problems could arise, we have the following code:

```
void traverse(Agraph_t *g, Agedge_t *e0)
{
  Agnode_t *n = aghead (e0);
  Agedge_t *e;

  for (e = agfstout(g, n); n; n = agnxtout(g, e)) {
    if (e == e0) continue;   /* avoid entry edge */
    /* do something with e */
  }
```

If `e0` is an in-edge (`AGTYPE(e) == AGINEDGE`), the comparison with `e` will always fail, as the latter is an out-edge.

In Cgraph, the nesting of subgraphs forms a tree. In libgraph, a subgraph can belong to more than one parent, so they form a DAG (directed acyclic graph). Libgraph actually represents this DAG as a special meta-graph that is navigated by libgraph calls. After gaining experience with libgraph, we decided this complexity was not worth its cost. In libgraph, the code for traversing subgraphs would have a form something like:

```
Agedge_t* me;
Agnode_t* mn;
Agraph_t* mg = g->meta_node->graph;
Agraph_t* subg;
for (me = agfstout(mg, g->meta_node); me; me = agnxtout(mg, me)) {
  mn = me->head;
  subg = agusergraph(mn);
    /* use subgraph subg */
}
```

The similar traversal using Cgraph would have the form

```
Agraph_t* subg;
for (subg = agfstsubg(g); subg; subg = agnxtsubg(subg)) {
    /* use subgraph subg */
}
```

Finally, there are some small syntactic differences in the APIs. For example, in libgraph, the name of a node or graph and the head and tail nodes of an edge are directly accessible via pointers while in Cgraph, it is necessary to use the functions `agnameof`, `agtail` and `aghead`. Libgraph tends to have separate functions for creating and finding an object, or for handling different types of objects, e.g., `agnode` and `agfindnode`. Instead, Cgraph will use a single function with an additional parameter. Overall, the two libraries are vary close, both syntactically and semantically, so conversion is fairly straightforward.

Tables 1–3 list the common constants and operations in libgraph, and the corresponding value or procedure in Cgraph, if any.

**Lgraph** is a successor to Cgraph, written in C++ by Gordon Woodhull. It follows Cgraph's overall graph model (particularly, its subgraphs and emphasis on efficient dynamic graph operations) but uses the C++ type system of templates and inheritance to provide typesafe, modular and efficient internal attributes. (LGraph relies on cdt for dictionary sets, with an STL-like C++ interface layer.) A fairly mature prototype of the Dynagraph system (a successor to dot and neato to handle online maintenance of dynamic diagrams) has been prototyped in LGraph. See the dgwin (Dynagraph for Windows) page `http://www.dynagraph.org/dgwin/` for further details.

# 15 Interfaces to Other Languages

If enabled, the Graphviz package contains bindings for Cgraph in a variety of languages, including Java, Perl, PHP, Tcl, Python and Ruby.

| *libgraph* | *Cgraph* |
|---|---|
| `AGRAPH` | `Agundirected` |
| `AGRAPHSTRICT` | `Agstrictundirected` |
| `AGDIGRAPH` | `Agdirected` |
| `AGDIGRAPHSTRICT` | `Agstrictdirected` |
| `aginit` | not necessary[6] |
| `agopen(name,type)` | `agopen(name,type,NULL)` |
| `agread(filep)` | `agread(filep,NULL)` |
| `agread_usergets(filep,gets)` | `Agmemdisc_t mem = AgMemDisc;`<br>`Agiddisc_t id = AgIdDisc;`<br>`Agiodisc_t io = AgIoDisc;`<br>`io.afread = gets;`[7]<br>`agread(filep,disc);` |
| `agsetiodisc` | no direct analogue; see above |
| `obj->name` | `agnameof(obj);` |
| `graph->root` | `agroot(graph);` |
| `node->graph` | `aggraphof(node);` |
| `edge->head` | `aghead(edge);` |
| `edge->tail` | `agtail(edge);` |
| `AG_IS_DIRECTED(graph)` | `agisdirected(graph)` |
| `AG_IS_STRICT(graph)` | `agisstrict(graph)` |
| `agobjkind(obj)` | `AGTYPE(obj)` |
| `agsubg(parent,name)` | `agsubg(parent,name,1)` |
| `agfindsubg(parent,name)` | `agsubg(parent,name,0)` |
| `g->meta_node_graph`<br>`agusergraph(graph)` | `agfstsubg/agnxtsubg`<br>See the examples on Page 21 |
| `aginsert(graph, obj)` | `agsubnode(graph,obj);` if obj is a node<br>`agsubedge(graph,obj);` if obj is a edge<br>not allowed if obj is a graph |

Table 1: Graph function conversions

# 16 Open Issues

**Node and Edge Ordering.** The intent in Cgraph's design was to eventually support user-defined node and edge set ordering, overriding the default (which is object creation timestamp order). For example, in topologically embedded graphs, edge lists could potentially be sorted in clockwise order around nodes. Because Cgraph assumes that all edge sets in the same `Agraph_t` have the same ordering, there should probably be a new primitive to switching node or edge set ordering functions. Please contact the author if you need this feature.

| *libgraph* | *Cgraph* |
|---|---|
| `agnode(graph,name)` | `agnode(graph,name,1)` |
| `agfindnode(graph,name)` | `agnode(graph,name,0)` |
| `agedge(graph,tail,head)` | `agedge(graph,tail,head,NULL,1)` |
| `agfindedge(graph,tail,head)` | `agedge(graph,tail,head,NULL,0)` |

Table 2: Node and edge function conversions

| *libgraph* | *Cgraph* |
|---|---|
| `agprotograph()` | no analogue |
| `agprotonode(graph)` | not used |
| `agprotoedge(graph)` | not used |
| `agattr(obj, name, default)` | `agattr(agroot(obj),AGTYPE(obj),name,default)` |
| `agfindattr(obj, name)` | `agattrsym(obj,name)` |
| `agraphattr(graph, name default)` | `agattr(graph,AGRAPH,name,default)` |
| `agnodeattr(graph, name, default)` | `agattr(graph,AGNODE,name,default)` |
| `agedgeattr(graph, name, default)` | `agattr(graph,AGEDGE,name,default)` |
| `agfstattr(obj)` | `agnxtattr(agroot(obj),AGTYPE(obj),NULL)` |
| `agnxtattr(obj,sym)` | `agnxtattr(agroot(obj),AGTYPE(obj),sym)` |
| `aglstattr(obj)` | no analogue |
| `agprvattr(obj, sym)` | no analogue |

Table 3: Attribute function conversions

**XML.** XML dialects such as GXL and GraphML have been proposed for graphs. Although it is simple to encode nodes and edges in XML, there are subtleties to representing more complicated structures, such as Cgraph's subgraphs and nesting of attribute defaults. On the other hand, GXL and GraphML provide notions of compound graphs, which are not available in DOT. We've prototyped an XML parser and would like to complete and release this work if we had a compelling application. (Simple XML output of graphs is not difficult to program.)

**Graph views; external graphs.** At times it would be convenient to relate one graph's objects to another's without making one into a subgraph of another. At other times there is a need to populate a graph from objects delivered on demand from a foreign API (such as a relational database that stores graphs). We are now experimenting with attacks on some of these problems.

[6]The Cgraph library has a function called `aginit`, but it has a different signature and semantics.

[7]Note that the order of the parameters differs from the `gets` used in libgraph.

# 17 Example

The following is a simple Cgraph filter that reads a graph and emits its strongly connected components, each as a separate graph, plus an overview map of the relationships between the components. To save space, the auxiliary functions in the header ingraph.h are not shown; the entire program can be found in the Graphviz source code release under `cmd/tools`.

About lines 32-41 are the declarations for internal records for graphs and nodes. Line 43-48 define access macros for fields in these records. Lines 52-83 define a simple stack structure needed for the strongly connected component algorithm and down to line 97 are some global definitions for the program.

The rest of the code can be read from back-to-front. From around line 262 to the end is boilerplate code that handles command-line arguments and opening multiple graph files. The real work is done starting with the function *process* about line 223, which works on one graph at a time. After initializing the node and graph internal records using it aginit, it creates a new graph for the overview map, and it calls *visit* on unvisited nodes to find components. *visit* implements a standard algorithm to form the next strongly connected component on a stack. When one has been completed, a new subgraph is created and the nodes of the component are installed. (There is an option to skip trivial components that contain only one node.) *nodeInduce* is called to process the out-edges of nodes in this subgraph. Such edges either belong to the component (and are added to it), or else point to a node in another component that must already have been processed.

---

```
/*************************************************************************
 * Copyright (c) 2011 AT&T Intellectual Property
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License v1.0
 * which accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/epl-v10.html
 *
 * Contributors: See CVS logs. Details at http://www.graphviz.org/
 *************************************************************************/                                    10

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <stdio.h>
#include <stdlib.h>
#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif                                                                     20
#include "cgraph.h"
#include "ingraphs.h"

#ifdef HAVE_GETOPT_H
#include <getopt.h>
#else
```

```
#include "compat_getopt.h"
#endif

#define INF ((unsigned int)(-1))

typedef struct Agraphinfo_t {
    Agrec_t h;
    Agnode_t *rep;
} Agraphinfo_t;

typedef struct Agnodeinfo_t {
    Agrec_t h;
    unsigned int val;
    Agraph_t *scc;
} Agnodeinfo_t;

#define getrep(g)      (((Agraphinfo_t*)(g->base.data))->rep)
#define setrep(g,rep)      (getrep(g) = rep)
#define getscc(n)      (((Agnodeinfo_t*)((n)->base.data))->scc)
#define setscc(n,sub)      (getscc(n) = sub)
#define getval(n)      (((Agnodeinfo_t*)((n)->base.data))->val)
#define setval(n,newval)      (getval(n) = newval)

/********* stack **********/
typedef struct {
    Agnode_t **data;
    Agnode_t **ptr;
} Stack;

static void initStack(Stack * sp, int sz)
{
    sp->data = (Agnode_t **) malloc(sz * sizeof(Agnode_t *));
    sp->ptr = sp->data;
}

static void freeStack(Stack * sp)
{
    free(sp->data);
}

static void push(Stack * sp, Agnode_t * n)
{
    *(sp->ptr++) = n;
}

static Agnode_t *top(Stack * sp)
{
    return *(sp->ptr - 1);
}
```

```
static Agnode_t *pop(Stack * sp)
{
    sp->ptr--;
    return *(sp->ptr);                                                    80
}

/********* end stack ***********/

typedef struct {
    int Comp;
    int ID;
    int N_nodes_in_nontriv_SCC;
} sccstate;
                                                                          90
static int wantDegenerateComp;
static int Silent;
static int StatsOnly;
static int Verbose;
static char *CmdName;
static char **Files;
static FILE *outfp;                    /* output; stdout by default */

static void nodeInduce(Agraph_t * g, Agraph_t * map)
{                                                                         100
    Agnode_t *n;
    Agedge_t *e;
    Agraph_t *rootg = agroot(g);


    for (n = agfstnode(g); n; n = agnxtnode(g, n)) {
        for (e = agfstout(rootg, n); e; e = agnxtout(rootg, e)) {
            if (agsubnode(g, aghead(e), FALSE))
                agsubedge(g, e, TRUE);
            else {                                                        110
                Agraph_t *tscc = getscc(agtail(e));
                Agraph_t *hscc = getscc(aghead(e));
                if (tscc && hscc)
                    agedge(map, getrep(tscc),
                           getrep(hscc), NIL(char *), TRUE);
            }
        }
    }
}
                                                                          120
static int visit(Agnode_t * n, Agraph_t * map, Stack * sp, sccstate * st)
{
    unsigned int m, min;
    Agnode_t *t;
```

```
    Agraph_t *subg;
    Agedge_t *e;

    min = ++(st->ID);
    setval(n, min);
    push(sp, n);

    for (e = agfstout(n->root, n); e; e = agnxtout(n->root, e)) {
        t = aghead(e);
        if (getval(t) == 0)
            m = visit(t, map, sp, st);
        else
            m = getval(t);
        if (m < min)
            min = m;
    }

    if (getval(n) == min) {
        if (!wantDegenerateComp && (top(sp) == n)) {
            setval(n, INF);
            pop(sp);
        } else {
            char name[32];
            Agraph_t *G = agraphof(n);;
            sprintf(name, "cluster_%d", (st->Comp)++);
            subg = agsubg(G, name, TRUE);
            agbindrec(subg, "scc_graph", sizeof(Agraphinfo_t), TRUE);
            setrep(subg, agnode(map, name, TRUE));
            do {
                t = pop(sp);
                agsubnode(subg, t, TRUE);
                setval(t, INF);
                setscc(t, subg);
                st->N_nodes_in_nontriv_SCC++;
            } while (t != n);
            nodeInduce(subg, map);
            if (!StatsOnly)
                agwrite(subg, outfp);
        }
    }
    return min;
}

static int label(Agnode_t * n, int nodecnt, int *edgecnt)
{
    Agedge_t *e;

    setval(n, 1);
    nodecnt++;
```

```
    for (e = agfstedge(n->root, n); e; e = agnxtedge(n->root, e, n)) {
        (*edgecnt) += 1;
        if (e->node == n)
            e = agopp(e);
        if (!getval(e->node))
            nodecnt = label(e->node, nodecnt, edgecnt);
    }                                                                    180
    return nodecnt;
}

static int
countComponents(Agraph_t * g, int *max_degree, float *nontree_frac)
{
    int nc = 0;
    int sum_edges = 0;
    int sum_nontree = 0;
    int deg;                                                             190
    int n_edges;
    int n_nodes;
    Agnode_t *n;

    for (n = agfstnode(g); n; n = agnxtnode(g, n)) {
        if (!getval(n)) {
            nc++;
            n_edges = 0;
            n_nodes = label(n, 0, &n_edges);
            sum_edges += n_edges;                                        200
            sum_nontree += (n_edges - n_nodes + 1);
        }
    }
    if (max_degree) {
        int maxd = 0;
        for (n = agfstnode(g); n; n = agnxtnode(g, n)) {
            deg = agdegree(g, n, TRUE, TRUE);
            if (maxd < deg)
                maxd = deg;
            setval(n, 0);                                                210
        }
        *max_degree = maxd;
    }
    if (nontree_frac) {
        if (sum_edges > 0)
            *nontree_frac = (float) sum_nontree / (float) sum_edges;
        else
            *nontree_frac = 0.0;
    }
    return nc;                                                           220
}
```

```
static void process(Agraph_t * G)
{
    Agnode_t *n;
    Agraph_t *map;
    int nc = 0;
    float nontree_frac = 0;
    int Maxdegree = 0;
    Stack stack;                                                          230
    sccstate state;

    aginit(G, AGRAPH, "scc_graph", sizeof(Agraphinfo_t), TRUE);
    aginit(G, AGNODE, "scc_node", sizeof(Agnodeinfo_t), TRUE);
    state.Comp = state.ID = 0;
    state.N_nodes_in_nontriv_SCC = 0;

    if (Verbose)
        nc = countComponents(G, &Maxdegree, &nontree_frac);
                                                                          240
    initStack(&stack, agnnodes(G) + 1);
    map = agopen("scc_map", Agdirected, (Agdisc_t *) 0);
    for (n = agfstnode(G); n; n = agnxtnode(G, n))
        if (getval(n) == 0)
            visit(n, map, &stack, &state);
    freeStack(&stack);
    if (!StatsOnly)
        agwrite(map, outfp);
    agclose(map);
                                                                          250
    if (Verbose)
        fprintf(stderr, "%d %d %d %d %.4f %d %.4f\n",
                agnnodes(G), agnedges(G), nc, state.Comp,
                state.N_nodes_in_nontriv_SCC / (double) agnnodes(G),
                Maxdegree, nontree_frac);
    else if (!Silent)
        fprintf(stderr, "%d nodes, %d edges, %d strong components\n",
                agnnodes(G), agnedges(G), state.Comp);

}                                                                         260

static FILE *openFile(char *name, char *mode)
{
    FILE *fp;
    char *modestr;

    fp = fopen(name, mode);
    if (!fp) {
        if (*mode == 'r')
            modestr = "reading";                                          270
        else
```

```
            modestr = "writing";
        fprintf(stderr, "gvpack: could not open file %s for %s\n",
                name, modestr);
        exit(1);
    }
    return (fp);
}

static char *useString = "Usage: %s [-sdv?] <files>\n\                                    280
  -s         - only produce statistics\n\
  -S         - silent\n\
  -d         - allow degenerate components\n\
  -o<outfile> - write to <outfile> (stdout)\n\
  -v         - verbose\n\
  -?         - print usage\n\
If no files are specified, stdin is used\n";

static void usage(int v)
{                                                                                        290
    printf(useString, CmdName);
    exit(v);
}

static void scanArgs(int argc, char **argv)
{
    int c;

    CmdName = argv[0];
    opterr = 0;                                                                          300
    while ((c = getopt(argc, argv, ":o:sdvS")) != EOF) {
        switch (c) {
        case 's':
            StatsOnly = 1;
            break;
        case 'd':
            wantDegenerateComp = 1;
            break;
        case 'o':
            outfp = openFile(optarg, "w");                                               310
            break;
        case 'v':
            Verbose = 1;
            break;
        case 'S':
            Verbose = 0;
            Silent = 1;
            break;
        case '?':
            if (optopt == '?')                                                           320
```

```
                usage(0);
            else
                fprintf(stderr, "%s: option -%c unrecognized - ignored\n",
                        CmdName, optopt);
            break;
        }
    }
    argv += optind;
    argc -= optind;

    if (argc)
        Files = argv;
    if (!outfp)
        outfp = stdout;          /* stdout the default */
}

static Agraph_t *gread(FILE * fp)
{
    return agread(fp, (Agdisc_t *) 0);
}

int main(int argc, char **argv)
{
    Agraph_t *g;
    ingraph_state ig;

    scanArgs(argc, argv);
    newIngraph(&ig, Files, gread);

    while ((g = nextGraph(&ig)) != 0) {
        if (agisdirected(g))
            process(g);
        else
            fprintf(stderr, "Graph %s in %s is undirected - ignored\n",
                    agnameof(g), fileName(&ig));
        agclose(g);
    }

    return 0;
}
```