Agraph Tutorial

Stephen C. North
AT&T Shannon Laboratory, Florham Park, NJ, USA
north@research.att.com

August 2, 2002

1 Introduction

Agraph 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.

Agraph 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 Agraph. Efforts were made in Agraph's design to strive for time and space efficiency. The basic (unattributed) graph representation takes 48 bytes per node and 64 bytes per edge, so storage of graphs with millions of objects is reasonable. For attributed graphs, Agraph 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.

Agraph 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 Agraph programs.

2 Graph Objects

Almost all Agraph 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

Agraph is responsible for its own memory management; allocation and deallocation of Agraph data structures is always done through Agraph 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 (self-arcs and multi-edges forbidden). [1]

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, 0);
```

The first argument to `agopen` is any string, and is not interpreted by Agraph, except it is recorded and preserved when the graph is written as a file.[2] The second argument is a graph type, 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 Agraph, and in most situations can just be `0`.

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,0);
```

[1]It would be nice to also have a graph type allowing self-arcs but not multi-edges; mixed graphs could also be useful.

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

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`).

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 it can later be reconstructed by calling `agread` on the same file.[3]

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

To delete a graph and its associated data structures, (freeing their memory):

```
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,0);
```

4 Nodes

In Agraph, a node is usually identified by a unique string name and a unique 32-bit internal ID assigned by Agraph. (For convenience, you can also create "anonymous" nodes by giving `NULL` as the node name.) A node also has in- and out-edge sets.

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 Agraph searches to locate an existing node with the given name.

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

[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.

The function `agdegree(n, in, out)` gives the degree of a node, where `in` and `out` select the edge sets. `agdegree(n,TRUE,FALSE)` returns in-degree, `agdegree(n,FALSE,TRUE)` returns out-degree, and `agdegree(n,TRUE,TRUE)` returns their sum.

`agnameof(n)` returns the printable string name of a node. Note that for various reasons this string may be a temporary buffer that is overwritten by subsequent calls. Thus, `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(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” (but there is no special interpretation of these fields in an undirected graph). An edge is made by the

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

    /* assume u and v are already defined */
    e = agedge(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, label) is treated as a unique identifier for edges between a particular node pair. That is, there can only be at most one edge labeled `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. Alternatively, `e->node` is the “other” endpoint with respect to the node from which e was obtained. A common idiom is: `for (e = agfstout(n); e; e = agnxtout(e)) f(e->node);`

`agedge` can also search for edges:

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

An edge can be deleted from a graph or subgraph by `agdeledge(e)`.

6 Traversals

Agraph has functions for walking 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(n)) {
    for (e = agfstout(n); e; n = agnxtout(e)) {
        /* do something with e */
    }
}
```

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

In the case of a directed edge, the meaning of "out" is somewhat obvious. For undirected graphs, Agraph assigns an arbitrary internal orientation to all edges for its internal bookkeeping. (It's from lower to higher internal ID.)

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

```
for (e = agfstedge(n); e; n = agnxtedge(e,n))
    /* do something with 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 14).

7 External Attributes

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

Agraph 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 helpful predefined constants AGRAPH, AGNODE and AGEDGE in calls to create, search and walk these dictionaries.

To create an attribute:

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

If this succeeeds, `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.)

You can search the attribute definitions of a graph.

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

Or walk 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, either using the string name or an `Agsym_t*` as an index. To use the string name:

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

8 Internal Attributes

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 Agraph. 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 that must already exist; `agdelrec` removes a record from an object.

Two other points:

1. For convenience, there is a way to "lock" the data pointer of a graph object to point to a given record. 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;
```

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);
```

9 Subgraphs

Subgraphs are an important construct in Agraph. They are intended for organizing subsets of graph objects and can be used interchangably with top-level graphs in almost all Agraph 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). Agraph has functions to create, search, and walk subgraphs.

```
Agraph_t     *agfstsubg(Agraph_t *g);
Agraph_t     *agnxtsubg(Agraph_t *subg);
```

For example,

```
Agraph_t *g, *h0, *h1;
g = agread(stdin,0);

h0 = agsubg(g,"mysubgraph",FALSE);  /* search for subgraph by name */
h1 = agsubg(g,"mysubgraph",TRUE);   /* create subgraph by name */

assert (g == agparent(h1));     /* agparent is up one level */
assert (g == agroot(h1));       /* agroot is the top level graph */
```

The functions `agsubnode` and `agsubedge` take a subgraph pointer, and a pointer to an object from another subgraph of the same graph (or possibly a top-level object) and rebind the pointer to a copy of the object from the requested subgraph. If `createflag` is nonzero, then the object is created if necessary; otherwise the request is only treated as a search and returns 0 for failure.

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

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

When a node is in more than one subgraph, distinct node structures are allocated for each instance. Thus, node pointers comparison cannot meaningfully be used for testing equality if the pointers could have come from different subgraphs - use ID instead.

```
    if (u == v) /* wrong */
    if (AGID(u) == AGID(v))  /* right */
```

10 Utility Functions and Macros

For convenience, Agraph provides some polymorphic functions and macros that apply to all Agraph 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 (a small integer)
- `AGID(obj)`: internal object ID (an unsigned long)
- `AGSEQ(obj)`: object creation timestamp (an integer)

- `AGDATA(obj)`: data record pointer (an `Agrec_t*`)

Other functions, as listed below, return the graph of an object, its string name, test whether it is a root graph object, and provide a polymorphic interface to remove objects.

```
Agraph_t *agraphof(void*);
char *agnameof(void*);
int  agisarootobj(void*);
int  agdelete(Agraph_t *g, void *obj);
```

11 Expert-level tweaks

Callbacks. There is a way to register client functions to be called whenever graph objects are inserted, or modified, or are about to be deleted from a graph or subgraph. The arguments to the callback functions for insertion and deletion (an `agobjfnt`) are the object and a pointer to a closure (a piece of data controlled by the client). The object update callback (an `agobjupdfn_t`) also receives the data dictionary entry pointer for the name-value pair that was changed.

```
typedef void    (*agobjfn_t)(Agobj_t *obj, void *arg);
typedef void    (*agobjupdfn_t)(Agobj_t *obj, void *arg, Agsym_t *sym);

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 a closure or client data structure `state` that is later passed to the callback function when it is invoked.

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

Callbacks are removed by `agpopdisc` [sic], 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:

```
int            agcallbacks(Agraph_t *g, int flag); /* return prev value */
```

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.)

Note: it is a small inconsistency that Agraph 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.

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

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

A pointer to this structure can be passed to `agopen` and `agread` (and `agconcat`) to override Agraph’s defaults.

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 Agraph 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.[4]

Other functions allow access to a graph’s heap for memory allocation. (It probably only makes sense to use this feature in combination with an optional arena-based memory manager, as described below under **Disciplines**.)

[4]This could be fixed.

```
typedef 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);
} Agmemdisc_t;

void                    *agalloc(Agraph_t *g, size_t size);
void                    *agrealloc(Agraph_t *g, void *ptr, size_t oldsize
void                    agfree(Agraph_t *g, void *ptr);
struct _vmalloc_s       *agheap(Agraph_t *g);


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

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

The file I/O functions were turned into a discipline (instead of hard-coding calls to the C stdio library into Agraph) because some Agraph clients could have their own stream I/O routines, or could "read" a graph from a memory buffer instead of an external file. (Note, it is also possible to separately redefine `agerror(char*)`, overriding Agraph's default, for example, to display error messages in a screen dialog instead of the standard error stream.)

The ID assignment discipline makes it possible for an Agraph client control this namespace. For instance, in one application, the client creates 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. `open` and `close` allow initialization and finalization for a given graph; `alloc` and `free` explicitly create and destroy IDs. `map` is called to convert an external 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. `print\` is called to convert an internal ID back to a string.

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

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

Flattened node and edge lists. For flat-out efficiency, there is a way of linearizing the splay trees in which node and sets are stored, converting them into flat lists. After this they can be walked very quickly. This is done by:

```
voi     agflatten(Agraph_t *g, int flag);
int     agisflattened(Agraph_t *g);
```

Shared string pool. As mentioned, Agraph maintains a shared string pool per graph. Agraph has functions to directly create and destroy references to shared strings.

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

Error Handling. Agraph invokes an internal function, `agerror` that prints a message on `stderr` to report a fatal error. An application may provide its own version of this function to override fatal error handling. The error codes are listed below. Most of these error should "never" occur.[5]

```
void agerror(int code, char *str);
#define AGERROR_SYNTAX  1       /* error encountered in lexing or parsing 
#define AGERROR_MEMORY  2       /* out of memory */
#define AGERROR_UNIMPL  3       /* unimplemented feature */
#define AGERROR_MTFLOCK 4       /* move to front lock has been set */
#define AGERROR_CMPND   5       /* conflict in restore_endpoint() */
#define AGERROR_BADOBJ  6       /* passed an illegal pointer */
#define AGERROR_IDOVFL  7       /* object ID overflow */
#define AGERROR_FLAT    8       /* attempt to break a flat lock */
#define AGERROR_WRONGGRAPH 9    /* mismatched graph and object */
```

[5]Also, a graph file syntax error really shouldn't be fatal; this can easily be remedied.

12 Related Libraries

Libgraph is a predecessor of Agraph and lives on as the base library of the *dot* and *neato* layout tools. Programs that invoke libdot or libneato APIs therefore need libgraph, not Agraph, at least until dot and neato are updated.

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 structs. 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.

In Agraph, 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.

Finally, there are some syntactic differences in the APIs. Where most Agraph calls take just a graph object, the equivalent Libgraph calls take both a graph/subgraph and an object, and Libgraph rebinds the object to the given graph or subgraph.

Lgraph is a successor to Agraph, written in C++ by Gordon Woodhull. It follows Agraph'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.research.att.com/sw/tools/graphviz/dgwin` for further details.

13 Interface to other languages

There is a 3rd-party PERL module for Graphviz that contains bindings for Libgraph (note, not Agraph) as well as the dot and neato layout programs. Graphviz itself contains a TCL/tk binding for Agraph as well as the other libraries.

14 Open Issues

Node and Edge Ordering. The intent in Agraph'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 Agraph 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.

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 Agraph's subgraphs and nesting of attribute defaults. 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.

Additional primitives. To be done: Object renaming, cloning, etc.

15 Example

The following is a simple Agraph filter that reads a graph and emits its strongly connected components, each as a separate graph plus an overview map of the relationship 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 tools/src.

About lines 40-50 are the declarations for internal records for nodes and edges. Line 50-80 define access functions and macros for fields in these records. Lines 90-130 define a simple stack structure needed for the strongly connected component algorithm and down to line 138 are some global definitions for the program.

The rest of the code can be read from back-to-front. From around line 300 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 265, which works on one graph at a time. After initializing the node and edge internal records, 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 outedges 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.

```
/*
    This software may only be used by you under license from AT&T Corp.
    ("AT&T"). A copy of AT&T's Source Code Agreement is available at
    AT&T's Internet website having the URL:
    <http://www.research.att.com/sw/tools/graphviz/license/source.html>
    If you received this software without first entering into a license
    with AT&T, you have an infringing copy of this software and cannot use
    it without violating AT&T's intellectual property rights.
*/

/*
 * Written by Stephen North
 * Updated by Emden Gansner
 */

/*
 * This is a filter that reads a graph, searches for strongly
 * connected components, and writes each as a separate graph
 * along with a map of the components.
 */
#ifdef HAVE_CONFIG_H
#include <gvconfig.h>
#endif

#include <stdio.h>
#ifdef HAVE_UNISTD_H
#include        <unistd.h>
#endif
#include <agraph.h>
#include <ingraphs.h>

#ifndef HAVE_GETOPT_DECL
#include <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;

#ifdef INLINE
#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)
#else
static Agnode_t *
getrep(Agraph_t *g)
{
   return (((Agraphinfo_t*)(g->base.data))->rep);
}
static void
setrep(Agraph_t *g, Agnode_t *rep)
{
   ((Agraphinfo_t*)(g->base.data))->rep = rep;
}
static Agraph_t *
getscc(Agnode_t *n)
{
   return (((Agnodeinfo_t*)(n->base.data))->scc);
}
static void
setscc(Agnode_t *n, Agraph_t *scc)
{
   ((Agnodeinfo_t*)(n->base.data))->scc = scc;
}
static int
getval(Agnode_t *n)
{
   return (((Agnodeinfo_t*)(n->base.data))->val);
}
static void
setval(Agnode_t *n, int v)
{
   ((Agnodeinfo_t*)(n->base.data))->val = v;
}
#endif

/********* 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);
}

#ifdef INLINE
#define push(sp,n) (*((sp)->ptr++) = n)
#define top(sp) (*((sp)->ptr - 1))
#define pop(sp) (*(--((sp)->ptr)))
#else
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);
}
#endif


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

typedef struct {
  int   Comp;
  int   ID;
  int    N_nodes_in_nontriv_SCC;
} sccstate;

int              wantDegenerateComp;
int              Silent;
int              Verbose;
char*            CmdName;
char**           Files;

static void
nodeInduce(Agraph_t *g)
{
  Agnode_t         *n, *rootn;
  Agedge_t         *e;

  for (n = agfstnode(g); n; n = agnxtnode(n)) {
    rootn = agsubnode(agroot(g),n,FALSE);
    for (e = agfstout(rootn); e; e = agnxtout(e)) {
      if (agsubnode(g,aghead(e),FALSE)) agsubedge(g,e,TRUE);
      else {
        if (getscc(aghead(e)) && getscc(agtail(e)))
          agedge(getrep(getscc(agtail(e))),getrep(getscc(aghead(e))),
            NIL(char*),TRUE);
      }
    }
  }
}

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); e; e = agnxtout(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);
      if (!Silent) agwrite(subg,stdout);
    }
  }
  return min;
}

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

  setval(n,1);
  nodecnt++;
  for (e = agfstedge(n); e; e = agnxtedge(e,n)) {
    (*edgecnt) += 1;
    if (e->node == n) e = agopp(e);
    if (!getval(e->node))
      nodecnt = label(e->node,nodecnt,edgecnt);
  }
  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;
  int           n_edges;
  int           n_nodes;
```

```
  Agnode_t*   n;

  for (n = agfstnode(g); n; n = agnxtnode(n)) {
    if (!getval(n)) {
      nc++;
      n_edges = 0;
      n_nodes = label(n,0,&n_edges);
      sum_edges += n_edges;
      sum_nontree += (n_edges − n_nodes + 1);
    }
  }
  if (max_degree) {
    int maxd = 0;
    for (n = agfstnode(g); n; n = agnxtnode(n)) {
      deg = agdegree(n,TRUE,TRUE);
      if (maxd < deg) maxd = deg;
      setval(n,0);
    }
    *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;
}

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

  initStack(&stack, agnnodes(G) + 1);
  map = agopen("scc_map",Agdirected,(Agdisc_t *)0);
  for (n = agfstnode(G); n; n = agnxtnode(n))
```

```
    if (getval(n) == 0) visit(n,map,&stack,&state);
  freeStack(&stack);
  if (!Silent) agwrite(map,stdout);
  agclose(map);

  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
    fprintf(stderr,"%d nodes, %d edges, %d strong components\n",
      agnnodes(G), agnedges(G), state.Comp);

}

static char* useString =
"Usage: %s [-sdv?] <files>\n\
  -s - silent\n\
  -d - allow degenerate components\n\
  -v - verbose\n\
  -? - print usage\n\
If no files are specified, stdin is used\n";

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

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

  CmdName = argv[0];
  while ((c = getopt(argc,argv,":?sdv")) != EOF) {
    switch (c) {
    case 's':
      Silent = 1; break;
    case 'd':
      wantDegenerateComp = 1; break;
    case 'v':
      Verbose = 1; break;
    case '?':
      if (optopt == '?') usage(0);
      else fprintf(stderr,"%s: option -%c unrecognized - ignored\n",
        CmdName, c);
      break;
    }
  }
  argv += optind;
  argc −= optind;

  if (argc) Files = argv;
}

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;
}
```