CMPS 101
Algorithms and Abstract Data Types
Fall 2006

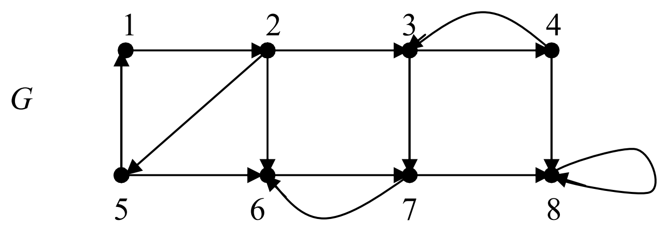
Programming Assignment 4

Due Friday November 17, 10:00 pm

In this assignment you will build a Graph module in C, implement Depth First Search (DFS), and use your Graph module to find the strongly connected components of a directed graph. Begin by reading sections 22.3-22.5 in the text.

A directed graph (or digraph) $G = (V, E)$ is said to be *strongly connected* if for every pair of vertices $u, v \in V$, u is reachable from v and v is reachable from u. Most directed graphs are not strongly connected. In general we say a subset $X \subseteq V$ is *strongly connected* if every vertex in *X* is reachable from every other vertex in *X*. A strongly connected subset which is maximal with respect to this property is called a *strongly connected component* of *G*. In other words, $X \subseteq V$ is a strongly connected component of *G* if and only if (i) *X* is strongly connected, and (ii) the addition of one more vertex to *X* would create a subset which is not strongly connected.

Example



It's easy to see that there are 4 strong components of *G*: $C_1 = \{1, 2, 5\}$, $C_2 = \{3, 4\}$, $C_3 = \{6, 7\}$, and $C_4 = \{8\}$.

To find the strong components of a digraph *G* first call $\text{DFS}(G)$, then as vertices are finished, place them on a stack. When this is complete the stack stores the vertices ordered by decreasing finish times. Next compute the transpose G^T of *G*, which is obtained by reversing the directions on all edges of *G*. Finally, run $\text{DFS}(G^T)$, but in the main loop (lines 5-7) of DFS, process vertices by decreasing finish times. (Here we mean finish times from the first call to DFS.) This is accomplished by simply popping vertices off the stack created in the first call. When this process is complete the trees in the resulting DFS forest constitute the strong components of *G*. (Note the strong components of *G* are identical to the strong components of G^T.) See the algorithm (Strongly-Connected-Components) and proof of correctness in section 22.5 of the text.

In this assignment you will create a graph module in C which represents a directed graph by an array of adjacency lists. Your graph module will, among other things, provide the capability of running DFS, and computing the transpose of a directed graph. DFS requires that vertices posses attributes for color (white, black, grey), discover time, finish time, and parent. Here is a catalog of required functions and their prototypes:

```
/* Constructors-Destructors */
GraphRef newGraph(int n);
void freeGraph(GraphRef *G);
```

```
/* Access functions */
int getOrder(GraphRef G);
int getParent(GraphRef G, int u);  /* Pre: 1<=u<=n=getOrder(G) */
int getDiscover(GraphRef G, int u);   /* Pre: 1<=u<=n=getOrder(G) */
int getFinish(GraphRef G, int u);   /* Pre: 1<=u<=n=getOrder(G) */

/* Manipulation procedures */
void addDirectedEdge(GraphRef G, int u, int v);  /* Pre: 1<=u<=n, 1<=v<=n */
void DFS(GraphRef G, ListRef S);    /* Pre: getLength(S)==getOrder(G) */

/* Other Functions */
GraphRef transpose(GraphRef G);
GraphRef copyGraph(GraphRef G);
void printGraph(FILE* out , GraphRef G);
```

Function newGraph will return a handle to a new graph object containing *n* vertices and no edges. freeGraph frees all heap memory associated with a graph object and sets it's GraphRef argument to NULL. Function getOrder returns the number of vertices in the graph *G*, while functions getParent, getDiscover, and getFinish return the appropriate field values for the given vertex. Note that the parent of a vertex may be NIL. Also the discover and finish times of vertices will be undefined before DFS is called. You must #define constant macros for NIL and UNDEF which represent those values, and place the definitions in the file Graph.h. The function addDirectedEdge will add vertex *v* to the adjacency list of vertex *u* (but not *u* to the adjacency list of *v*), thus establishing a directed edge from *u* to *v*.

The function DFS will perform the depth first search algorithm on *G*. The List *S* has two purposes in this function. First it defines the order in which vertices will be processed in the main loop (5-7) of DFS. Second, when DFS is complete, it will store the vertices in order of decreasing finish times (hence *S* can be considered to be a stack). The List *S* can therefore be classified as both an input and an output parameter to function DFS. Obviously you should utilize the List module you created in pa2 to implement *S* and the adjacency lists which represent *G*. DFS has two preconditions: (i) getLength(*S*) = *n*, and (ii) *S* contains some permutation of the integers {1, 2, ..., *n*} where *n* = getOrder(G). You are required to check the first precondition but not the second.

Recall that DFS calls the recursive algorithm Visit (DFS-Visit in the text), and uses a variable called time which is static over all recursive calls to Visit. There are three possible approaches to implementing Visit. You can define Visit as a top level function in your graph implementation file (which is private and therefore *not* exported), and let time be a static variable whose scope is the entire file. This option has the drawback that other functions in the same file would have access to the global time variable. The second approach is to let time be a local variable in DFS, then pass the address of time to Visit, making it an input-output variable to Visit. This is perhaps the simplest option, and is recommended. The third approach is to again let time be a local variable in DFS, then nest the definition of Visit *within* the definition of DFS. Since time is local to DFS, it's scope includes the defining block for Visit, and is therefore static throughout all recursive calls to Visit. This may be tricky if you're not used to nesting function definitions since there are issues of scope to deal with. If you pick this option, first experiment with a few simple examples to make sure you know how it works. Note that although nesting function definitions is not a standard feature of ANSI C, and is not supported by many compilers, it is supported by the gcc compiler (even with the –ansi flag for some reason). If you plan to develop your project on another platform, this approach may not be possible.

Function transpose returns a handle to a new graph object representing the transpose of *G*, and copyGraph returns a handle to a new graph which is a copy of *G*. Both transpose and copyGraph could be considered constructors since they create new graph objects. Function printGraph prints the adjacency list representation of *G* to the file pointed to by out.

The client of your Graph module will be called FindComponents. It will take two command line arguments giving the names of the input and output files respectively:

```
% FindComponents infile outfile
```

See the example FileIO.c on the webpage to learn how to read from and write to files in C. The main program FindComponents will do the following:

- Read the input file.
- Assemble a graph object *G* using newGraph and addDirectedEdge.
- Print the adjacency list representation of *G* to the output file.
- Run DFS on *G* and G^T, processing the vertices in the second call by decreasing finish times from the first call.
- Determine the strong components of *G*.
- Print the strong components of *G* to the output file.

After the second call to DFS, the List parameter *S* can be used to determine the strong components of *G*. I suggest you trace the algorithm Strongly-Connected-Components (p.554) on several small examples, keeping track of the List *S*, to see how this can be done. Input and output file formats are illustrated in the following example, which corresponds to the directed graph on the first page of this handout:

Input:

```
8
1 2
2 3
2 5
2 6
3 4
3 7
4 3
4 8
5 1
5 6
6 7
7 6
7 8
8 8
0 0
```

Output:

```
Adjacency list representation of G:
1: 2
2: 3 5 6
3: 4 7
4: 3 8
5: 1 6
6: 7
7: 6 8
8: 8

G contains 4 strongly connected components:
Component 1: 1 5 2
Component 2: 3 4
Component 3: 7 6
Component 4: 8
```

Observe that the input file format is very similar to that of pa3. The first line gives the number of vertices in the graph, subsequent lines specify directed edges, and input is terminated by the ‘dummy’ line 0 0. You have basically the same two design options for your Graph module that you did in pa3. On the one hand the struct Graph may contain array fields for the parent, discover, finish, and color attributes, as well as an array of ListRefs for the adjacency lists. In each array, index *i* pertains to the vertex with label *i*. (Again I recommend that arrays be built to begin indexing at 1, i.e. just waste the zeroth element.) The second option would be to define a private inner struct called Vertex (analogous to Node in List) which encapsulates several attributes, such as parent, discover, finish, and color. Then Graph would contain arrays of Vertex and ListRef variables. Just as before, either option is acceptable.

You are required to submit the following files: README, Makefile, List.c, List.h, ListClient.c, Graph.h, Graph.c, GraphClient.c, FindComponents.c. As usual README contains a catalog of submitted files and any special notes to the grader. Makefile should be capable of making the executables ListClient, GraphClient, FindComponents, and should contain a clean utility which removes all object files. The grader will be taking a second look at your List module so be sure to fix any errors he pointed out in pa2. Graph.c and Graph.h are the implementation and interface files respectively for your Graph module. GraphClient.c is used for testing of your Graph module. FindComponents.c implements the top level client and main program for this project. To get full credit, your project must implement all required files and functions, compile without errors or warnings, produce correct output on our test files, and produce no memory leaks under bcheck. By now everyone knows that points are deducted both for neglecting to include required files, and for submitting additional unwanted files, but let me say it anyway: do not submit binary files of any kind.

Note that FindComponents needs to pass a List to the function DFS, so it is also a client of the List module and must therefore #include the file List.h. Also note that the Graph module will be exporting a function which takes a ListRef argument (namely DFS). Therefore any file which #includes Graph.h (namely Graph.c, GraphClient.c, and FindComponents.c) must have the preprocessor directive #include for List.h appear *before* that for Graph.h, since the compiler must see the typedef which defines ListRef before it sees the prototype for function DFS(). Failure to do this will cause a syntax error. You should not #include List.h inside Graph.h to resolve this syntax error.

Below is a Makefile which you may alter as you see fit:

```
# Makefile for Graph ADT and related modules.
#     make                 makes FindComponents
#     make GraphClient     makes GraphClient
#     make ListClient      makes ListClient
#     make clean           removes all object and executable files

FindComponents : FindComponents.o Graph.o List.o
      gcc -o FindComponents FindComponents.o Graph.o List.o

GraphClient : GraphClient.o Graph.o List.o
      gcc -o GraphClient GraphClient.o Graph.o List.o

ListClient : ListClient.o List.o
      gcc -o ListClient ListClient.o List.o

FindComponents.o : FindComponents.c Graph.h List.h
      gcc -c -ansi -Wall FindComponents.c

GraphClient.o : GraphClient.c Graph.h List.h
      gcc -c -ansi -Wall GraphClient.c

ListClient.o : ListClient.c List.h
      gcc -c -ansi -Wall ListClient.c

Graph.o : Graph.c Graph.h List.h
      gcc -c -ansi -Wall Graph.c

List.o : List.c List.h
      gcc -c -ansi -Wall List.c

clean :
      rm -f FindComponents GraphClient ListClient FindComponents.o \
      GraphClient.o ListClient.o Graph.o List.o
```

Start Early and ask questions if anything is not completely clear.