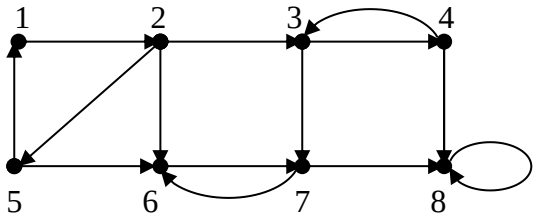
CMPS 101
Algorithms and Abstract Data Types
Programming Assignment 4

In this assignment you will build a Graph module in C that implements Depth First Search (DFS). You will use your Graph module to find the strongly connected components of a digraph. Begin by reviewing sections 22.3-22.5 of the text.

A digraph $G = (V, E)$ is said to be *strongly connected* iff for every pair of vertices $u, v \in V$, vertex u is reachable from v, and vertex v is reachable from u. Most directed graphs are not strongly connected. In general we say a subset $X \subseteq V$ is *strongly connected* iff every vertex in X is reachable from every other vertex in X. A strongly connected subset that 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 that is not strongly connected.

Example

G:



We can see that this digraph contains 4 strongly connected components: $C_1 = \{1, 2, 5\}$, $C_2 = \{3, 4\}$, $C_3 = \{6, 7\}$, and $C_4 = \{8\}$.

To find the strong components of a digraph G call DFS(G). As vertices are finished, place them on a stack. When DFS is complete the stack stores the vertices ordered by decreasing finish times. Next compute the transpose G^T of G. (The digraph G^T is obtained by reversing the directions on all edges of G.) Finally run DFS(G^T), but in the main loop (lines 5-7) of DFS, process vertices in order of decreasing finish times from the first call to DFS. This is accomplished by popping vertices off the stack. When the whole process is complete, the trees in the resulting DFS forest span the strong components of G. Note that 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 project you will again create a graph module in C using the adjacency list representation. 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 possess attributes for color (white, black, grey), discover time, finish time, and parent. Here is a catalog of required functions and their prototypes:

```
/* Constructors-Destructors */
Graph newGraph(int n);
void freeGraph(Graph* pG);
/* Access functions */
int getOrder(Graph G);
int getSize(Graph G);
int getParent(Graph G, int u);  /* Pre: 1<=u<=n=getOrder(G) */
int getDiscover(Graph G, int u);   /* Pre: 1<=u<=n=getOrder(G) */
int getFinish(Graph G, int u);   /* Pre: 1<=u<=n=getOrder(G) */
```

```
/* Manipulation procedures */
void addArc(Graph G, int u, int v);  /* Pre: 1<=u<=n, 1<=v<=n */
void addEdge(Graph G, int u, int v);  /* Pre: 1<=u<=n, 1<=v<=n */
void DFS(Graph G, List S);    /* Pre: length(S)==getOrder(G) */

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

Function newGraph() will return a reference to a new graph object containing *n* vertices and no edges. freeGraph() frees all heap memory associated with a graph and sets its Graph argument to NULL. Function getOrder() returns the number of vertices in *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. The discover and finish times of vertices will be undefined before DFS is called. You must #define constant macros for NIL and UNDEF representing those values, and place the definitions in the file Graph.h. The descriptions of functions addEdge() and addArc() are exactly as they were in pa4. Note that as in pa4, it is required that adjacency lists always be processed in increasing order by vertex label. It is the responsibility of functions addEdge() and addArc() to maintain adjacency lists in order by inserting new vertices into the correct locations.

Function DFS() will perform the depth first search algorithm on *G*. The argument 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(). You should utilize the List module you created in pa2 to implement *S* and the adjacency lists representing *G*. DFS() has two preconditions: (i) $\text{length}(S) == n$, and (ii) *S* contains some permutation of the integers $\{1, 2, \ldots, n\}$ where $n = \text{getOrder}(G)$. You are required to check the first precondition but not the second.

Recall DFS() calls the recursive algorithm Visit() (referred to as DFS-Visit in the text), and uses a variable called `time` that is static over all recursive calls to Visit(). Observe that this function is not mentioned in Graph.h and therefore is to be considered a private helper function in the Graph ADT. There are at least three possible approaches to implementing Visit(). You can define Visit() as a top level function in your graph implementation file and let `time` be a global variable whose scope is the entire file. This option has the drawback that other functions in the same file would have access to the `time` variable and would be able to alter its value. Global variables are generally considered to be a poor programming practice. 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 C, and is not supported by many compilers, it *is* supported by the gcc compiler (even with the `-std=c99` flag). If you plan to develop your project on a platform other than the UCSC ITS Unix timeshare, this approach might not be possible. There's actually a fourth possibility for implementing time that may be the easiest of all. Just give each call to Visit() its own local copy of time. It will then take the current value of time as input, and return the new value of time when it is done, no global variable, no passing by reference and no nested function definitions. These are design decisions left to you, and the type of thing that should be included in your README file.

Function transpose() returns a reference to a new graph object representing the transpose of *G*, and copyGraph() returns a reference 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. Obviously there is much in common between the graph module in this project and the one in pa4. If you wish, you may simply add functionality necessary for this project to the previous module, although it is not required that you do so. You should make note of choices such as this in your README file.

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
```

The main program FindComponents will do the following:

- Read the input file.
- Assemble a graph object *G* using newGraph() and addArc().
- 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 in topologically sorted order.

After the second call to DFS(), the List parameter *S* can be used to determine the strong components of *G*. You should 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 pa4. 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 are required to submit the following eight files:

README
Makefile

List.c
List.h
Graph.h
Graph.c
GraphTest.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 GraphTest and FindComponents, and should contain a clean utility that removes all object files. Graph.c and Graph.h are the implementation and interface files respectively for your Graph module. GraphTest.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 valgrind. By now everyone knows that points are deducted both for neglecting to include required files, for misspelling any filenames and for submitting additional unwanted files, but let me say it anyway: do not submit binary files of any kind.

Note that FindComponents.c needs to pass a List to the function DFS(), so FindComponents.c is also a client of the List module. In fact any client of Graph is also a client of List just by the presence of the List parameter to DFS(). Therefore Graph.h should itself #include the file List.h. (See the handout entitled *C Header File Guidelines* for more on this topic.)

A Makefile for this project will be posted on the course webpage which you may alter as you see fit. As usual start Early and ask questions if anything is not completely clear.