CSE 373 Homework 7 Project Description

Assigned:	Wednesday, May 29, 2002
Due:	Wednesday, June 5, 2002
	At the start of class

Introduction

For this homework project, you will develop a code module that implements an unweighted shortest path algorithm and answer some questions.

mainGraph. The mainGraph program calls readGraph (provided) to read in a graph description and turn it into a set of adjacency lists. It calls your procedure UnweightedPathLength to build a path table like figure 9.19, and then prints a description of the graph that is suitable for processing with dot to build a graphic display of the graph.

The main program for the homework is supplied in ADT/Graph/mainGraph.c, and you are to write one new procedure, UnweightedPathLength, as described in figure 9.18 in the book. Note that the pseudocode in the book is close to what is needed, but is not complete.

Grading

The 7 homework assignments of the quarter will count for a total of 50% of your class grade, and each individual homework assignment will count for about 7% of the total class grade. The grading for this project is as follows.

Questions:	10 points
Implementation:	10 points
Total	20 points

NOTE: You need to turn in several things:

1. the paper copy of your answer sheet
2. printouts of fig9p4.dot and fig9p10.dot showing the output from your program
3. printouts of the two plots generated by dot from the files in item 2
4. a copy of the receipt you got when you did the web turn in
5. and do a web turnin of your implementation of graph.c

Staple the answer sheet, the dot files, the plots, and the receipt together and turn them in on Wednesday.

Directory Structure

All of the homework projects are implementations of one of the Abstract Data Types that we discuss in class. The directory structure of the files you receive is as follows:

ADT/	top level directory
ADT/include	header files
ADT/nets	example graph descriptor files
ADT/Graph	directory for the Graph project

Program: mainGraph

The purpose of this program is to read the graph description file and create a graph adjacency list representation. It then does a topological sort and a shortest path run over the graph, and finally creates a drawing showing the structure of the graph.

Program: dot

This program converts dot files to postscript, suitable for plotting.

If you want to add printf statements to your code, you should write out your information surrounded by /* and */, because then dot can still read the output file and it will ignore these lines (it considers them to be comments). See plotgraph.c for examples of how to do this.

To do:

1. Create the Graph project just as you have done for the previous projects. For the Program arguments entry, use "..\nets\fig9p4.txt".
2. Note that although there is a file "graph.c" provided to you it is missing the routine that you need to implement. Also note that the List, Queue, and Hash ADT implementations are all provided. They are the same functions we have been implementing over the entire quarter.
3. Change the name that is included in the function getGraphAuthor. As delivered to you, it says "Anonymous Author." You should change that to be your own name.
4. Write the UnweightedPathLength procedure described below, rebuild the project, and run it. Debug until done.
5. If you run the project from Visual C, the output gets displayed in a console window. I have supplied a batch file called winrunit.bat that runs your program (Debug\graph.exe) from a command line and stores the output in dot input files. It then runs dot, and stores the results in postscript files. To use this batch file, get a command prompt (Start->Programs->Accessories->Command Prompt) and use the cd command to move to the ADT\Graph directory. Then type "winrunit".

The dot files contain whatever information your program wrote out. If you have been careful to surround any output you have added with /* and */ then dot should run okay.

The postscript files contain an actual drawing of the graph that was created. You can use Ghostview (available from our web site) or any other postscript viewer to display and print the drawings.

6. Review the code as needed in order to answer the questions in the homework.

The definitions of struct GraphHeader, struct Node, struct Edge, and struct PathEntry are in include/privategraph.h. They are not in graph.c because the plot functions need to be able to see the structure definitions also.

The functions are as follows.

char *getGraphAuthor(void)

Returns a text string to the caller naming the person who wrote the program. Change “Anonymous Author” to your name.

void UnweightedPathLength(Graph G, struct Node *S);

Implement the unweighted shortest path algorithm described in section 9.3.1 and figure 9.18. You need to understand how the adjacency lists are organized and use the functions in the List ADT to access them. The adjacency lists are created in readGraph using CreateListArray, and then creating one adjacency list for each node. You need to understand how to create and use a Queue, using the functions in the Queue ADT.

I suggest that you study the Topsort routine to see how the various parts of the pseudocode are implemented, and then apply that to your implementation of UnweightedPathLength. In particular you need to know that there is an index value stored in each Node structure, so when the book says something like T[W], you can translate the W into an array index by getting the index value out of the Node structure.

You can use the constant value DBL_MAX to represent infinity.

A brief piece of pseudocode for this procedure is:

```
allocate the uPathTable array with one entry per node
initialize the uPathTable entries
create a queue for the nodes
save the pointer to the start Node in G->uPathStart
initialize the uPathTable entry for the start node.  Use S->index
to get the correct subscript
enqueue S
while the queue is not empty
      dequeue a node and process all the nodes adjacent to it per
      fig 9.18.  The adjacency lists are stored in the array
      G->adj, and the list for a particular node S is at
      G->adj[S->index].
destroy the queue
```

void Topsort(Graph G);

This is the topological sort procedure described in section 9.2 and figure 9.7. You can use it to help you understand how the various pointers, arrays, and indexes are stored in the actual implementation.

include/privategraph.h

struct Node – a description of a single node. Pointers to these objects are enqueued and dequeued in the operation of your procedure. The “index” value tells you the array offset of this node, for use when storing and retrieving data in the uPathTable. The “index” value is also used to select the correct list header from the array of list headers in the GraphHeader G->adj[idx].

struct Edge – a description of a single edge. Each entry in an adjacency list contains a pointer to an edge. The “aP” entry in the Edge is a pointer to the initial node of the edge, the “bP” entry is the terminal node of the edge.

struct PathEntry – a single entry in the state table maintained during the operation of the shortest path algorithm. Contains “known”, “dist”, and previous node information. Your procedure needs to allocate an array of these structures, and store the location of the array in the GraphHeader at offset uPathTable. Your procedure then uses the table to maintain state information as it processes the nodes.

struct GraphHeader – contains various pieces of information describing the graph as follows:

“node” and “edge” are pointers to arrays of pointers to Node and Edge objects. You don’t need to use these in your procedure. “nodeCount” and “edgeCount” are simple integer values containing the obvious values. “hash” is a pointer to a hash table which you don’t need to use.

“adj” is a pointer to an array of Lists of adjacency entries. There is one list for each node, and the list contains pointers to edges that start at that node and go to an adjacent node. If you have a pointer to an Edge object in variable edgeP, then the Node object that describes the terminal node of the edge is pointed to by edgeP->bP.

“topNum” is a pointer to an array of topological sort numbers. It is created by Topsort, and you only need to look at it to help you understand how Topsort works.

“uPathStart” is a pointer to the initial node in the shortest path analysis. You need to set this value to the value that is passed into your procedure. “uPathTable” is the table built by your procedure, and is the implementation of figure 9.19.

fig9p4.dot

```
/* Graph has a topological sort. */
/* Graph has shortest path table.  Start node is v1. */
digraph G {
label="Figure 9.4 An acyclic graph\nTue May 28 14:45:33 2002 - Anonymous Author";
size="7.5,10";
center=true;
n0x8056238 [label = "v1" ];   /* idx: 0, indegree: 0, tsort: 1, dist: 0 (start node) */
n0x8056258 [label = "v2" ];   /* idx: 1, indegree: 1, tsort: 2, dist: 1 via v1 */
n0x8056278 [label = "v3" ];   /* idx: 2, indegree: 2, tsort: 6, dist: 1 via v1 */
n0x8056298 [label = "v4" ];   /* idx: 3, indegree: 3, tsort: 4, dist: 1 via v1 */
n0x80562b8 [label = "v5" ];   /* idx: 4, indegree: 1, tsort: 3, dist: 2 via v2 */
n0x80562d8 [label = "v6" ];   /* idx: 5, indegree: 3, tsort: 7, dist: 2 via v3 */
n0x80562f8 [label = "v7" ];   /* idx: 6, indegree: 2, tsort: 5, dist: 2 via v4 */
"n0x8056238" -> "n0x8056278" [ style = bold ];
"n0x8056238" -> "n0x8056298" [ style = bold ];
"n0x8056238" -> "n0x8056258" [ style = bold ];
"n0x8056258" -> "n0x80562b8" [ style = bold ];
"n0x8056258" -> "n0x8056298" [ style = dotted ];
"n0x8056278" -> "n0x80562d8" [ style = bold ];
"n0x8056298" -> "n0x80562f8" [ style = bold ];
"n0x8056298" -> "n0x80562d8" [ style = dotted ];
"n0x8056298" -> "n0x8056278" [ style = dotted ];
"n0x80562b8" -> "n0x8056298" [ style = dotted ];
"n0x80562b8" -> "n0x80562f8" [ style = dotted ];
"n0x80562f8" -> "n0x80562d8" [ style = dotted ];
} /* 7 nodes, 12 edges */
```

fig9p10.dot

```
/* Graph has no topological sort (or it has a cycle). */
/* Graph has shortest path table.  Start node is v1. */
digraph G {
label="Figure 9.10 An unweighted directed graph\nTue May 28 14:45:34 2002 - Anonymous
Author";
size="7.5,10";
center=true;
n0x8056248 [label = "v1" ];   /* idx: 0, indegree: 1, dist: 0 (start node) */
n0x8056268 [label = "v2" ];   /* idx: 1, indegree: 1, dist: 1 via v1 */
n0x8056288 [label = "v3" ];   /* idx: 2, indegree: 1, dist: 2 via v4 */
n0x80562a8 [label = "v4" ];   /* idx: 3, indegree: 2, dist: 1 via v1 */
n0x80562c8 [label = "v5" ];   /* idx: 4, indegree: 2, dist: 2 via v4 */
n0x80562e8 [label = "v6" ];   /* idx: 5, indegree: 3, dist: 2 via v4 */
n0x8056308 [label = "v7" ];   /* idx: 6, indegree: 2, dist: 2 via v4 */
"n0x8056248" -> "n0x80562a8" [ style = bold ];
"n0x8056248" -> "n0x8056268" [ style = bold ];
"n0x8056268" -> "n0x80562c8" [ style = dotted ];
"n0x8056268" -> "n0x80562a8" [ style = dotted ];
"n0x8056288" -> "n0x80562e8" [ style = dotted ];
"n0x8056288" -> "n0x8056248" [ style = dotted ];
"n0x80562a8" -> "n0x80562c8" [ style = bold ];
"n0x80562a8" -> "n0x8056308" [ style = bold ];
"n0x80562a8" -> "n0x80562e8" [ style = bold ];
"n0x80562a8" -> "n0x8056288" [ style = bold ];
"n0x80562c8" -> "n0x8056308" [ style = dotted ];
"n0x8056308" -> "n0x80562e8" [ style = dotted ];
} /* 7 nodes, 12 edges */
```