Winter 2019 CMPS 101-01, Shel Finkelstein
Programming Assignment 2 (PA2)
Due Date: Sunday, February 10, 11:59pm (strict) on GitLab

The purpose of this assignment is to implement a Graph ADT (for an Undirected Graph) and associated operations in C. This project will utilize your List ADT from PA1, so and make sure your List is working properly. Begin by reading the handout Graphs.pdf file that's on Resources→General Resources, as well as appendices B.4 and B.5 from the text and class Lectures 8 and 9.

The Adjacency List representation of a Graph consists of an array of Lists. We've discussed this representation in Lectures 8 and 9. You will create a Graph ADT that represents a graph as an array of Lists. Each vertex will be associated with an integer label in the range 1 to *numVertices*, where *numVertices* is the number of vertices in the graph.

Your client program will use this Graph ADT to perform various operations on the Graph that are described below. The client program using your Graph ADT will be called `GraphProperties`, and will take two command line arguments (here `%` denotes the unix prompt):

```
% GraphProperties input_file output_file
```

For PA2, you are to write a makefile that creates the executable file, called GraphProperties, similar to the makefile that you used for PA1. Include a clean utility in your makefile.

Input and Output File Formats

The first line of the input file describes the graph, and the other lines describe operations to be performed on the graph. Most of these operations simply print values returned by functions implemented in a Graph ADT that you will implement in Graph.c and Graph.h, similar to the List.c and List.h files that you used in PA1.

The first line starts with an integer that we'll call numVertices that tells you the number of vertices in the graph. The rest of that line gives pair of distinct numbers in the range 1 to numVertices, separated by a space. These numbers are the vertices for an edge. There is a comma between numVertices and the first edge, and a comma between edges. We'll also put a space after each comma for readability. Here's an example that represents the undirected graph on Slide 12 of Lecture 8:

4, 1 3, 3 2, 3 4, 4 2

Note that the same graph could be represented by each of the following input lines:

4, 3 1, 3 2, 4 3, 4 2

4, 3 1, 3 2, 1 3, 4 3, 4 2, 4 3

even though the last graph has edge {1, 3} and edge {4, 3} appearing twice. (Edges {1, 3} and {3, 1} are the same edge.)

The rest of the lines in the input file that correspond to graph operations on operands. Each line begins with a keyword, which is followed by the operands. You should output the entire input line into the output file as a separate line. That line should be followed by another line that just has the result of the operation. If an input line does not follow one of the specified formats, the output on the second line should just say ERROR, and processing of the input file should continue. For example, it's an error if the operation isn't one of the legal operations (capitalization matters), or if the operation has the wrong number of operands, or if an operand that's supposed to be a vertex isn't a number between 1 and numVertices.

So the output file should always have twice as many lines as the input file (not counting the line that describes the graph). For example, if an input file has 7 lines, the first of which is the graph, then the output should have 12 lines in it, containing each graph operation lines, each followed immediately by the result of that operation (or ERROR).

Here are the operation tokens, their operands, and descriptions of the results. As we mentioned, most of these operations simply print values returned by functions implemented in the Graph ADT that you will implement in Graph.c and Graph.h. But *PathExists* is a little more interesting; we'll outline its implementation later in this file.

- **PrintGraph** takes no operands. It prints out the current graph, in the same format as an input line. But it should not print out an edge twice. Do this by only printing edge {u, v} only when u < v.

- **GetOrder** takes no operands. It returns the total number of vertices in the current graph.

- **GetSize** takes no operands. It returns the total number of edges in the current graph.

- **GetNeighborCount** takes a vertex u as operand. It returns the number of vertices that are neighbors of u in the current graph.

- **PathExists** takes a pair of vertices u and v as operands, and outputs YES if there is a path from u to v in the current graph, and NO otherwise. A description of how to perform PathExists using a recursive search appears at the end of this file. (Yes, if the operands u and v are the same vertex, there is a path from u to v, so the output will be YES.)

Example: For an input file that consists of the following lines:

4, 3 1, 3 2, 1 3, 4 3, 4 2, 4 3
PrintGraph
GetSize
PathExists 1 4
PathExists 3
GetNeighborCount 4
GetNeighborCount 5
GetNeighborCount 3

the output file should be:

PrintGraph
4, 1 3, 2 3, 3 4, 2 4
GetSize
4
PathExists 1 4
YES
PathExists 3
ERROR
GetNeighborCount 4
2
GetNeighborCount 5
ERROR
GetNeighborCount 3
3

Your Graph ADT will be implemented in files Graph.c and Graph.h. Graph.c defines a struct called GraphObj, and Graph.h will define a type called Graph that points to this struct. (You might want to re-read the C material in the two ADT handouts that are posted under Resources→General Resources, and review the Queue and Stack examples that are posted under Resources→QueueExample and Resources→StackExample.)

Your GraphObj struct is required to have the following fields:

- numVertices, the number of vertices in the graph, called the order of the graph.
- numEdges, the number of edges in the graph, called the size of the graph.
- An array of Lists, whose j^{th} element contains the neighbors of vertex j. This is the Adjacency List for vertex j. You'll use your List ADT from PA1 to represent the Adjacency List for each vertex.
- An array of int values whose j^{th} element indicates whether a vertex has been "visited" in the current search. You will define constants UNVISITED, INPROGRESS and ALLDONE in the file Graph.h. Usage of these constants in PathExists is explained below.

Since indexes of C arrays begin at 0, you might choose to use arrays that have length numVertices +1, but only use indices 1 through numVertices. That's so that array indices can be directly identified with vertex numbers (and index 0 is ignored). Of course, if you prefer to have arrays of length numVertices, rather than numVertices + 1,that works, as long as you always remember to look at position j-1 when you're accessing information about vertex j.

Your Graph ADT should export the following operations through the file Graph.h:

```
/*** Constructors-Destructors ***/
Graph newGraph(int numVertices);
        // Returns a Graph that points to a newly created GraphObj representing a graph which has
        // n vertices and no edges.

void freeGraph(Graph* pG);
        // Frees all dynamic memory associated with its Graph* argument, and sets
        // *pG to NULL.

/*** Access functions ***/
int getOrder(Graph G);
        // Returns the order of G, the number of vertices in G.

int getSize(Graph G);
        // Returns the size of G, the number of edges in G.

int getNeighborCount(Graph G, int v);
        // Returns the number of neighbors that vertex v has in G.  Returns -1 if v is not a legal vertex.

List getNeighbors(Graph G, int v);
        // Returns a list that has all the vertices that are neighbors of v.  There is no input operation
        // that corresponds to getNeighbors.

/*** Manipulation procedures ***/

int addEdge(Graph G, int u, int v);
        // Adds v to the adjacency list of u and u to the adjacency list of v, if that edge doesn't
        // already exist.  If the edge does already exist, does nothing.  Used when edges are entered.
        // Returns 0 if u and v are legal vertices, otherwise returns -1.

void unvisitAll(Graph G);
        // Marks all vertices of the graph as UNVISITED.

int getMark(Graph G, int u);
        // Returns the mark for vertex u, which will be UNVISITED, INPROGRESS or ALLDONE.

void setMark(Graph G, int u, int theMark);
        // Sets the mark for vertex u to be theMark.
        // theMark must be UNVISITED, INPROGRESS or ALLDONE.

int PathExistsRecursive(Graph G, int w, int v)
        // Described below; returns FOUND or NOTFOUND, which are (different) constants.

/*** Other operations ***/
void printGraph(FILE* out, Graph G);
        // Prints the Graph G in the same format as an input line, so all that a client need to do is a single
        // call to PrintGraph().  But it should not print out an edge twice.  Achieve this by only printing
        // the edge for {j, k} when j < k.
```

In addition to the above prototypes, Graph.h will define the type Graph, as well as #define constant macros UNVISITED, INPROGRESS and ALLDONE; these should be different integers, used as "marks". Graph.h will also have #define constants FOUND and NOTFOUND, which should also be different integers.

What does GraphProperties.c do?

GraphProperties reads in the first line of the input file, and uses newGraph() and addEdge() to create a graph that corresponds to that line, as described above. When it reads in a subsequent line whose operation token is one of *GetOrder, GetSize* or *GetNeighborCount,* it calls the corresponding function in Graph.c (using the graph that was read in as the graph argument), and outputs the value returned. Obvious requirements about number of operands should be checked in GraphProperties.c; obvious requirements about operand values in should be checked in Graph.c, as we already mentioned. (If G isn't a graph, your program has a bug, and should exit.) The PrintGraph function in Graph.c that is exported in Graph.h doesn't return a value; it just prints the Graph.

void PathExists(FILE* out, Graph G, int u, int v), which is in GraphProperties.c, has a file, a graph and a pair of vertices u and v as its arguments, and determines whether or not there is a path from u to v, printing YES if there is a path and NO if there isn't. Here's an informal description of how to write PathExists using the Graph ADT.

You'll need to have a function **int PathExistsRecursive(G,w,v)** in Graph.c in order to execute PathExists. PathExistsRecursive determines whether there is a path from w to v, taking advantage of "marks" that indicated which vertices are UNVISITED, so that we don't repeat work or get stuck in cycles.

```
int PathExistsRecursive(G,w,v)
    If w=v THEN RETURN(FOUND)
    setMark(G,w,INPROGRESS)
    FOR EACH vertex x on the getNeighbors(G,w) List
        theMark = getMark(G,x).
        IF theMark is UNVISITED
            theFoundValue = PathExistsRecursive(G,x,v)
        IF theFoundValue is FOUND
            return(FOUND)
        // Found a path to w, so no need to continue

    // Finished processing all of x's neighbors without finding v
    setMark(G,w,ALLDONE)
    RETURN(NOTFOUND)
```

Now that we have PathExistsRecursive(G,v), here's what PathExists(G,u,v) does:

```
    unvisitAll(G)           // No vertices have been visited yet
    theFoundValue = PathExistsRecursive(G,u,v) )
    IF theFoundValue is FOUND
        OUTPUT YES
    ELSE OUTPUT 'NO'
```

As usual, the result of PathExists (which will be YES or NO) should be output on a separate line, immediately after the input file line that requested graph operation PathExists.

Submitting your Solution

Since the Graph ADT includes uses a List to represent the neighbors of each vertex, and getNeighbors() returns a List, the file Graph.h should #include the header file List.h. (See the handout C Header File Guidelines, which is posted under Resources→General Resources, for commonly accepted policies on using .h files.) What files should Graph.c and GraphProperties.c include?

You will submit 7 files for this project in the directory PA2. The names for these files and the directory PA2 are not optional. Points will be deducted if you turn in wrongly named files, or extra files such as data files or binary files. Each file you write must begin with a comment block that has your name, cruzid, and the assignment name (PA2, in this case).

List.c, List.h, Graph.c, Graph.h, GraphProperties.c, makefile, README

Under Resources→PA2 on Piazza, we've provided List.c and List.h files that you may use for PA2, but you may also use the List.h and List.c files which you wrote for PA1, if you believe that they are correct.

Based on the makefile from PA1, you should know how to construct a makefile that will create an executable called `GraphProperties`, and will include a `clean` utility that removes all object files, including `GraphProperties`. (There is no `ListClient` for PA2.) We will post some public input files that you can use for testing, but we will also test your program on other private input files.

Remember that the compile operations mentioned in the Makefile **must** invoke the gcc compiler with the flags `-std=c99`. You may develop your programs on any system, using any editor or IDE. But it is a requirement of this and all other assignments in C that your program compile without warnings or errors under gcc, and run properly in the Linux computing environment on the UNIX Timeshare unix.ucsc.edu provided by ITS. In particular, you should not use the cc compiler. Your C programs must also run without memory leaks. Test them using `valgrind` on unix.ucsc.edu by doing:

```
% valgrind --leak-check=full program_name argument_list
```

You also must submit a `README` file for this (and every) assignment. The `README` file should list each file (other than the `README` file itself) that you submitted, together with a brief description of its role in the project, and any special notes to the people grading your assignment. The `README` is essentially a table of contents for the assignment.

You must submit your PA2 project on GitLab following the same directions provided on Piazza for PA2, but using the PA2 directory. **The due date for PA2 is Sunday, February 10, 11:59pm, and that deadline will be strictly enforced.**