Winter 2019 CMPS 101-01, Shel Finkelstein
Programming Assignment 3 (PA3)
Due Date: Wednesday, February 27, 11:59pm (strict) on GitLab

The purpose of this assignment is to implement an ADT for a **Directed Graph** (**digraph**) and associated operations in C. PA3 will utilize your List ADT once again, and some parts of this project will look very familiar if you completed PA2. You should read the Graphs.pdf file handout that's on Resources→General Resources, as well as appendices B.4 and B.5 from CLRS, Chapter 22 of CLRS, and class Lectures 8, 9, 15, etc.

The Adjacency List representation of a digraph also consists of an array of Lists; this representation was discussed in Lectures 8 and 9. However, the edges in a digraph are directed (like one-way streets); there may be an edge (3, 1) between vertex 3 and vertex 1 although there is no edge (1, 3) between vertex 1 and vertex 3. You will create a **Digraph ADT** that represents a directed graph as an array of Lists. As in PA2, each vertex is associated with an integer label in the range 1 to *numVertices*, where *numVertices* is the number of vertices in the digraph.

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

```
% DigraphProperties input_file output_file
```

For PA3, you are to write a makefile that creates the executable file, called DigraphProperties, similar to the makefiles that you used for PA1 and PA2. Include a clean utility in your makefile.

Input and Output File Formats

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

The first line starts with an integer that we'll call numVertices that tells you the number of vertices in the digraph. 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. All edges are directed. Here's an example that represents digraph on Slide 14 of Lecture 8:

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

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

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

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

even though the edge (1, 3) appears in the last digraph twice.

However, the following digraph is different:

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

because edges {1, 3} and {3, 1} are different edges.

The rest of the lines in the input file that correspond to digraph 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 digraph). For example, if an input file has 7 lines, the first of which is the digraph, then the output should have 12 lines in it, containing each digraph operation lines, each followed immediately by the result of that operation (or ERROR).

As in PA2: If the digraph in the first line of the input file is illegal, then you should just print out that first line, followed by another line with the word ERROR on it. Do not process any subsequent lines in the input file. What's an illegal digraph? A digraph is illegal if the number of vertices isn't positive, or if an edge specifies a vertex that doesn't exist in the digraph. For example, if there are 6 vertices, both vertex 0 and vertex 7 are illegal. Also, there shouldn't be any "self-loop" edges between a vertex and itself, so an edge (3, 3) would also make the digraph illegal.

Here are the operation tokens, their operands, and descriptions of the results. **All** of these operations simply print values returned by functions implemented in the Digraph ADT that you will implement in Digraph.c and Digraph.h.

- **PrintDigraph** takes no operands. It prints out the current digraph, in the same format as an input line. Each edge should be printed out exactly once, following the same format as the input line for the digraph, starting with the number of vertices, then a comma, then the edges.

  For PA3, you must print the edges sorted by source and then by destination. If u < v, print all the edges whose source is vertex u before all the edges whose source is vertex v. Also, if (u, v1) and (u, v2) are edges, and v1 < v2, print (u, v1) before (u, v2).

  Strong suggestion: Keep the destinations for each vertex as a sorted list. That will make it easy to print out all the edges in the current digraph sorted as described.

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

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

- **GetOutDegree** takes a vertex u as operand. It returns the number of vertices that are outgoing neighbors of u in the current digraph. The out degree of a vertex u is the number of vertices v such that (u, v) is an edge in the digraph.

- **AddEdge** takes two vertices u and v as operands. It adds the edge (u, v) to the current digraph. If that edge didn't exist in the current digraph, it returns 0. If that edge already existed in the current digraph, it returns 1, since no action was taken. (This is not treated as an error.)

- **DeleteEdge** takes two vertices u and v as operands. It deletes the edge (u, v) from the current digraph. If that edge already existed in the current digraph, it returns 0. If that edge didn't exist in the current digraph, it returns 1, since no action was taken. (This is not treated as an error.)

- **Distance** takes a pair of vertices u and v as operands, and outputs the distance from u to v in the current digraph, and INF if there is no path from u to v. Of course, this must be a directed path. Yes, if the operands u and v are the same vertex, then there is a path from u to v that has no edges in it, so the output will be 0. The distance from u to v is the length of the shortest path from u to v.

- **Acyclic** takes no operands. It outputs YES if the current digraph is acyclic, and NO otherwise.

- **TopoSort** takes no operands. It outputs a topological sort of the current digraph, if the current digraph is acyclic. (There may be more than one correct topological sort.) However, if the digraph is not acyclic, it outputs IMPOSSIBLE.

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

```
4, 1 3, 2 4, 3 4, 4 3, 3 2
PrintDigraph
GetSize
Distance 1 4
Distance 3 1
Acyclic
TopoSort
GetOutDegree 4
GetOutDegree 5
DeleteEdge 2 4
Acyclic
DeleteEdge 1 2
DeleteEdge 4 3
AddEdge 1 2
Acyclic
TopoSort
```

the output file should be:

```
PrintDigraph
4, 1 3, 2 4, 3 2, 3 4, 4 3
GetSize
5
Distance 1 4
2
Distance 3 1
INF
Acyclic
NO
TopoSort
IMPOSSIBLE
GetOutDegree 4
1
GetOutDegree 5
ERROR
DeleteEdge 2 4
0
Acyclic
NO
DeleteEdge 1 2
1
DeleteEdge 4 3
0
AddEdge 1 2
0
Acyclic
YES
TopoSort
1 3 4 2
```

Your Digraph ADT will be implemented in files Digraph.c and Digraph.h. Digraph.c defines a struct called DigraphObj, and Digraph.h should define a type called Digraph that points to this struct.

Your DigraphObj struct is required to have the following fields. *It is okay for you to have other fields, but you should document them in your README file*.

- numVertices, the number of vertices in the digraph, called the order of the digraph.
- numEdges, the number of edges in the digraph, called the size of the digraph.
- An array of Lists, whose j^{th} element contains the outgoing neighbors (destinations) from source vertex j, the vertices k such that (j, k) is an edge in the digraph.. This is the outgoing Adjacency List for vertex j. You'll use your List ADT from PA1 and PA2 to represent the outgoing Adjacency List for each vertex. We strongly suggest that you keep the outgoing neighbors in sorted order, to make printing the digraph simpler.
- Other arrays similar to the marks that you used in PA2. You'll have to determine yourselves what arrays (if any) you'll need for PA3.

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 Digraph ADT should export the following operations through the file Digraph.h:

*/*** Constructors-Destructors ***/*
Digraph newDigraph(int numVertices);
// Returns a Digraph that points to a newly created DigraphObj representing a digraph which has
// n vertices and no edges.

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

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

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

int getOutDegree(Digraph G, int u);
// Returns the number of outgoing neighbors that vertex u has in G, the number of vertices v such
// that (u, v) is an edge in G. Returns -1 if v is not a legal vertex.

List getNeighbors(Digraph G, int u);
// Returns a list that has all the vertices that are outgoing neighbors of vertex u, i.e.,
// a list that has all the vertices v such that (u, v) is an edge in G.
// There is no input operation that corresponds to getNeighbors.

*/*** Manipulation procedures ***/*

int addEdge(Digraph G, int u, int v);
// Adds v to the adjacency list of u, if that edge doesn't already exist.
// If the edge does already exist, does nothing. Used when edges are entered or added.
// Returns 0 if (u, v) is a legal edge, and the edge <u>didn't</u> already exist.
// Returns 1 if (u, v) is a legal edge and the edge <u>did</u> already exist.
// Returns -1 if (u, v) is not a legal edge.

int deleteEdge(Digraph G, int u, int v);
// Deletes v from the adjacency list of u, if that edge exists.
// If the edge doesn't exist, does nothing. Used when edges are deleted.
// Returns 0 if (u, v) is a legal edge, and the edge <u>did</u> already exist.
// Returns 1 if (u, v) is a legal edge and the edge <u>didn't</u> already exist.
// Returns -1 if (u, v) is not a legal edge.

You probably will want to have other functions in Digraph.c/Digraph.h that are similar to the unvisitAll, getMark and setMark functions in PA). But you'll have to figure out what those functions are yourselves.

```
/*** Other operations ***/
void printDigraph(FILE* out, Digraph G);
	// Outputs the digraph G in the same format as an input line, including the number of vertices
	// and the edges.  The edges should be in sorted order, as described above.

void distance(FILE* out, Digraph G, int u, int v);
	// Outputs the distance between vertices u and v if there is a directed path between them in the
	// digraph.  Otherwise, outputs INF

void acyclic(FILE* out, Digraph G);
	// Outputs YES if the digraph is acyclic.  Otherwise, outputs NO.

void topoSort(FILE* out, Digraph G);
	// Outputs a topological sort of the digraph.  If the digraph is not acyclic, outputs IMPOSSIBLE.
```

In addition to the above prototypes, Digraph.h will define the type Digraph, as well as #define constant macros for constants that you decide that you need to implement the functions described above.

What does DigraphProperties.c do?

DigraphProperties reads in the first line of the input file, and uses newDigraph() and addEdge() to create a digraph that corresponds to that line, as described above. When it reads in a subsequent line whose operation token is one of *GetOrder, GetSize* or *GetOutDegree,* it calls the corresponding function in Digraph.c (using the digraph that was read in as the digraph argument), and outputs the value returned. For other operations tokens (*Distance, Acyclic, TopoSort*)*,* it calls the corresponding function in Digraph.c which performs its own output. Obvious requirements about number of operands should be checked in DigraphProperties.c; obvious requirements about operand values in should be checked in Digraph.c. (If G isn't a digraph, your program has a bug, and should exit.)

About **distance**: Find the distance between two vertices of a digraph using BFS. See Section 22.2 of CLRS.

About **acyclic**: A digraph is acyclic if and only if a DFS of the digraph produces no back edges, as described in Section 22.3 of CLRS. You'll have to figure out how to implement this.

About **topological sort**: Topological sort using DFS was discussed in Lecture 16 (February 15). See Section 22.4 of CLRS.

Submitting your Solution

Since the Digraph ADT includes uses a List to represent the outgoing neighbors of each vertex, and getNeighbors() returns a List, the file Digraph.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 Digraph.c and DigraphProperties.c include?

You will submit 7 files for this project in the directory PA3. The names for these files and the directory PA3 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 (PA3, in this case).

List.c, List.h, Digraph.c, Digraph.h, DigraphProperties.c, makefile, README

Under Resources→PA2 on Piazza (yes, PA2), we've provided List.c and List.h files that you may use for PA3, 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 `DigraphProperties`, and will include a `clean` utility that removes all object files, including `DigraphProperties`. (There is no `ListClient` for PA3.) 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, <u>you should not use the cc compiler</u>. 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. For PA3, you must also describe all the fields that you have in your DigraphObj struct besides the ones provided to you (numVertices, numEdges and the array of Lists) earlier in this document. <u>Also, your README must briefly explain your algorithms for distance, acyclic and topoSort, and provide complexity of your algorithms (e.g., O(n+m)). We'll deduct credit for inefficient algorithms.</u>

You must submit your PA3 project on GitLab following the same directions provided on Piazza for previous Programming Assignments, but using the PA3 directory. **The due date for PA3 is <u>Wednesday, February 27</u>, 11:59pm, and that deadline will be strictly enforced.**