CS211 – Programming Practicum Fall 2018

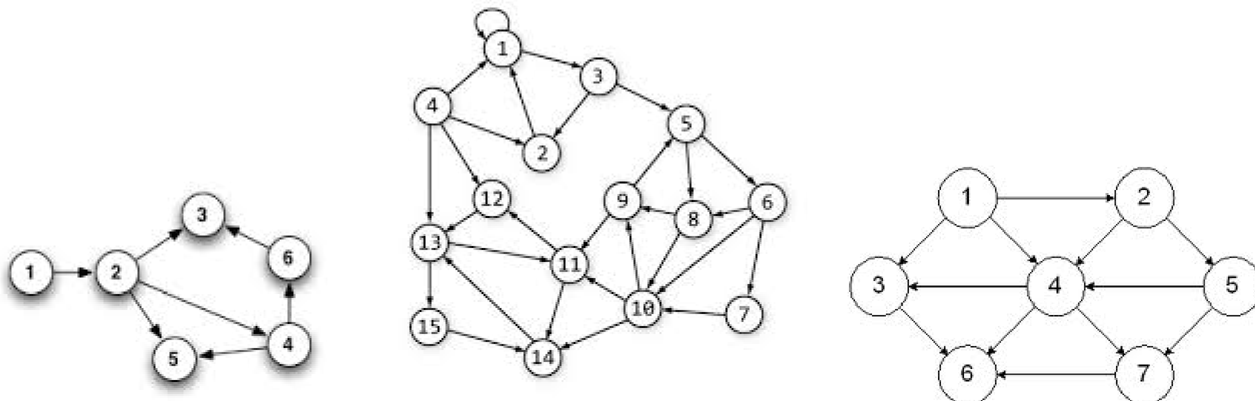
Programming Project 7

Due: Wednesday, 12/5/18 at 11:59 pm

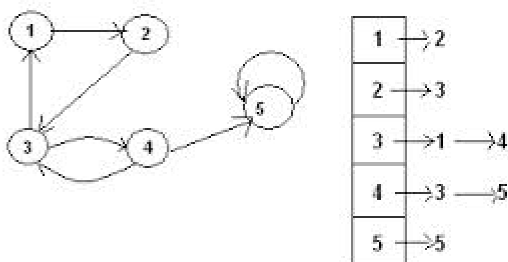
What is the Fastest Way to Get from Here to There?

For this program, you will write a C++ Program to represent a travel network as was done with Project 6. This travel network will use an array of linked lists as its primary storage structure. This type of storage structure is typically called an “adjacency list”. The main difference is here we want to do a Breadth First Search to find the short path between locations in the travel network.

Assume you have a small airline that flies planes between a small number of airports. Each airport will be given a number. If a plane flies from airport X to airport Y, the network will have an “edge” from X to Y. Below are a number of drawings that could represent this idea. The airports are represented by the circled numbers, the edges are represented by the arrows. Consider the first drawing. It has 6 airports. It has a plane that flies from airport 1 to airport 2. Three planes fly from airport 2, one to airport 3, one to airport 4 and one to airport 5. No planes leave from airport 3 or from airport 5 (yes, it would be stupid to strand planes at those airports, but ignore that fact for now). Planes fly from airport 4 to airports 5 and 6. Finally, planes fly from airport 6 to airport 3.



In an adjacency list, each location/airport needs a list of those locations/airports that one can get to in one move/flight. In this program, we need a list for each airport. If the travel network has N airports, the array will have N linked lists, one for each airport. If airport X has planes flying to 3 different airports, the linked list for airport X would have 3 nodes. Consider the following image showing a travel network and an adjacency list:



There are 5 airports, so we have an array of 5 linked lists. Since Airport 3 can fly planes to two Airports, namely Airport 1 and Airport 4, the linked list for Airport 3 has two nodes: one node containing the value 1; another node containing the value 4.

Program Input and Commands

The input for the operations will come from standard input and from files. The input will initially come from standard input. If the user specified the f command, your program will then read input from a file. See the description below for more details. The commands are to follow the descriptions given below. Note: that the form <int> could be any integer number and it will NOT be enclosed in angle brackets. <int> is just a notation to specify and integer value. The integer value is to be input on the same line as the command character. If the first character on the line is not one of the following characters, print an error message and ignore the rest of the information on that line.

q - quit the program immediately.

? - display a list of the commands the user can enter for the program.

\# **-** ignore this line of input. Treat the line of input as a comment

s <int1> <int2> - display shortest path from airport **<int1>** to airport **<int2>** in one or more flights.

t <int1> <int2> - display a message stating whether a person can travel from airport **<int1>** to airport **<int2>** in one or more flights.

r <int> - remove all values from the traffic network and resize the array to contain the number of airports as indicated by the given integer value. The value of the integer must be greater than zero. **The airports will be numbered from 1 to the given integer value.** Be sure to deallocate all unused memory as part of this command.

i <int1> <int2> - insert the edge to indicate a plane flies from airport **<int1>** to airport **<int2>**.

d <int1> <int2> - delete the edge that indicates a plane flying from airport **<int1>** to airport **<int2>**.

l - list all the items contained in the travel network. First display all of the airports (if any) that can be reached from the airport #1 in one flight (that have an edge in the network), followed by all the airports (if any) that can be reached from airport #2 in one flight, etc.

f <filename> - open the file indicated by the **<filename>** (assume it is in the current directory) and read commands from this file. When the end of the file is reached, continue reading commands from previous input source. This must be handled using recursion. Beware of a possible case of an infinite recursive loop, the **f** command is may not call a file that is currently in use.

Initially your program should have the array to hold 10 airports (numbered 1 to 10). If a command specifies an airport outside of the current valid range, print an error message and ignore the command.

The code given in proj6Base.cpp should provide the basics on reading input. You will need to do some cutting and pasting of code to read in input for all commands. The input is properly read in for the t and f commands. It is assumed that you can determine the code for the rest of the commands by looking at the code those these two commands.

Travel Algorithm and the Airport Object

To determine the path from airport X to airport Y in one or more flights, a breadth-first-search algorithm must be used. For this algorithm to work, we will need to be able to store which was the previous airport when each airport when visited. Setting up an Airport class is required. This object will contain all of the data that one airport knows:

- the head of the linked list for the airport's adjacency list,

- a value to determine if an airport has been visited or not (see end of paragraph below), and
- the methods to use that data.

The travel network **MUST** be a dynamic array of these Airport objects. The adjacency list will also need a Node class/object to store the linked list information. Note that for the Breadth First Search the "visited" information must include not just a Boolean values as was done for the Depth First Search. It needs to know from which Airport the first "visiting flight" originated. For example, assume Airport 4 is first visited with a flight from Airport 2. The visited information would need to store the value of 2 as the "visited value" for Airport 4. Initially when the Airport are being marked as "unvisited", store a unique value that is cannot be used an Airport value, such as -1 (or -999 or 0 or something less than 1).

The pseudo code for Breadth First Search algorithm is as shown below. Note it is valid to ask, can I to go from airport Z to airport Z in one or more flights. It really asks, "If I leave airport Z, how do I return to it?"

```
list breadthFirstSearch (int x, int y)
{
  mark all airports as unvisited (set all previousLocation to -1);
  set the list AirportQueue to be empty
  add x to the end of the AirportQueue
  if ( bfs ( y, AirportQueue ) = = FALSE )
     print ("You can NOT get from airport " + x + " to airport " + y + " in one or more flights");
     return an empty list
  else
     print ("You can get from airport " + x + " to airport " + y + " in one or more flights");
     pathList is set to an empty list
     set currentAirport to y
     add currentAirport to front of pathList
     do
          currentAirport = previousLocation for airport nodeValue
          add currentAirport to front of pathList
     while ( currentAirport != x )
     return pathList
}

boolean bfs ( int b, AirportQueue )
{
  while ( the AirportQueue is not empty )
    {
      Set a to be the Airport at the front of the AirportQueue
      Remove Airport at the front of the AirportQueue

      for (each airport c that can be reached from a in one flight)
          if ( airport c is unvisited (previousLocation is still -1 )  )
               mark airport c as visited (set previousLocation to a);
               if ( c == b )
                  return TRUE
               add c to the end of the AirportQueue
     }
  return FALSE
}
```

The doShortestPath ( ) Method

The doShortestPath( ) method of the TravelNetwork class is similar to the doTravel( ) method except:

1. It calls breadthFirstSearch ( ) instead of depthFirstSearchHelper ( ).
2. If the list returned by breathFirstSearch ( ) is not empty, it needs to print out the path (from start to finish) contained in that list

This method is left for you to add to your program.

Everyone is STRONGLY encourage to copy all of your files used in Project 6 to a different directory/project/folder/file for Project 7. Thus if there is a question in the grading of your Project 6, you still have an original/unmodified copy of Project 6 to refer to.

The FILE Command: f

The f command may seem difficult to implement at first, but it has a creative solution that you are to use. The code in the file Proj6Base.cpp is intended to give you an idea on how this solution is to be implemented.

First note that main( ), is extremely short. It just creates an instance of the TravelNetwork class and calls the processCommandLoop( ) method with an value of FILE* that reads from standard input.

The method processCommandLoop( ) reads from the input source specified by the parameter and determines the which command is being invoked.

When the f command is invoked, it is to open the file specified by the command, create a new instance of FILE* that reads from this file. Then make a recursive call to processCommandLoop( ) with this new instance of the FILE* so the next line of input comes from the specified file instead of where the previous command came from. When the end of a file is reached, the program is to revert back to the previous input source that contained the f command. This previous input source could be standard input or a file. By making these calls recursively, reverting back to the previous input source is a complete no-brainer.

However, this can cause an infinite loop if you try to access a file that your program is already reading from. Consider this scenario. Assume the user enters a command from standard input to start reading from file A. However; file A tells you to read from file B, file B tells you to read from file C, and file C tells you to read from file A. Since you always start reading from the top of the file, when file C eventually tells the program to read from file A, the program will reprocess the command to read from file B, which will reprocess the command to read from file C, which will reprocess the command to read from file A, which will reprocess the command to read from file B, which will reprocess the command to read from file C, which will reprocess the command to read from file A, which will reprocess…

In order to stop this, you are required to maintain a linked list of file names. Before the f command attempts to create a new instance of FILE* that read from file X, the f command is to check if the linked list of file names already contains the name of X.

- If the name X already exists in the linked list, the f command will NOT create a new instance of FILE* and it will NOT make the recursive call to processCommandLoop( ).

- If the name X does not exist in the linked list, the f command will add the name X to the linked list before making the recursive call to processCommandLoop( ) and it must remove the name X from the linked list after the call to processCommandLoop( ) returns.

You are responsible to write the code for this linked list yourself. Note that this will most likely be a linked list of Strings, while each airport's adjacency list will most likely be a linked list of integers.

Classes You Must Write

First you are to write a class called Airport. This class is to contain everything that is known about a specific Airport. This MUST include the airport's adjacency list and the "visited" status for the Depth First Search.

Next, you are to write a linked list class. You are encouraged to name the class MyList as the name "List" is already used in the C++ Standard Template Libraries. Please note that your list class will need another class for the nodes in the list. Thus you must write a class called MyNode also. For your list class, in addition to the normal insertValue( ) and deleteValue( ) operations, you may want to write operations such as getNumberOfCurrentsValues( ) and getNthValue( ).

You also need to have lists for the AirportQueue and for the pathList as mentioned in the required algorithm. The AiportQueue is actually a linked list queue (add to the end of the list, get item at front of the list, and remove from the front of the list). The pathList is actually a linked list stack (add to the front of the list, get item at the front of the list, and remove from the front of the list). You may wish to us inheritance to help build the queue and stack.

The TravelNetwork class has been started for you. This class must contain the array of Airports and the linked list of filenames for the "in-use" files needed for the f command.

All data members (or "instance variables") must be made private. Failure to do this will result is a severe reductions of points for the project.

You are not allowed to use any of the classes from the C++ Standard Template Libraries in this program. These classes include ArrayList, Vector, LinkedList, List, Set, Stack, HashMap, etc. **If you need such a class, you are to write it yourself.** These are sometimes called the C++ Standard Container Library. A full listing of the C++ Standard Template Libraries can be found at:
http://www.cplusplus.com/reference/stl/

Multiple Source Code Files

Your program is to be written using at least two source code files. One of the source file files is to contain the main function of the program named in a file using your NetId and Program name, like:
Ptroy1Proj7.cpp
The other source code file must contain your Airport class in a file named:
Airport7.cpp
You may use additional source code files if you wish (i.e. MyList.cpp), but these two are required. The above implies that you will need to write any appropriate .h file(s) and a makefile.

Note the name change to Airport7.cpp. This is so still have your original program for Project 6 in case there is a grading issue.

Coding Style

Don't forget to use good coding style when writing your program. Good coding style makes your program easier to be read by other people as the compiler ignores these parts/differences in your code. Elements of good code style include (but may not be limited to):

- Meaningful variable names
- Use of functions/methods
- Proper indentation
- Use of blank lines between code sections
- In-line comments
- Function/method header comments
- File header comments

The Code Review Checklist also hints at other elements of good coding style.

Program Submission

You are to submit the files for this project as a single zip file via the Assignments Page in Blackboard.

To help the TA, zip your files together and name your zip file with your net-id and the assignment name, like: Ptroy1Proj7.zip