Computer Science III (22C:30, 22C:115)
Project 2, Due: 11/13/02, 5 pm

1. Introduction.
In this programming project you will develop a new implementation of the `graph` class with additional functionality for graph traversal. You will test the new implementation is various ways and compare the efficiency of the new and old implementations.

2. Adjacency List Representation.
Implement a class called `graphAL` using the *adjacency list* representation discussed in class. Make sure that your new implementation provides all the functionality provided by the `graph` class implemented as part of Project 1. Specifically, the `+=` operator and the `-=` operator should continue to behave as they did in the `graph` class. The function `get_Neighbors` should be modified slightly, in that instead of returning a vector of neighboring nodes, the function should now return a linked list of neighboring nodes.
While most details of the implementation should be clear to you from working on Project 1 and our in-class discussion of adjacency lists, there is one issue that needs to be raised. Recall that the adjacency list representation consists of a vector of nodes, each node in the vector pointing to a list of neighboring nodes. The question is how large should this vector of nodes be and what should be done when it fills up. You should essentially do what you did in Project 1. That is, start with an empty vector and increase its size to 2 times its original size plus one, whenever an extra slot is needed for a new node.

3. Graph Traversals
Now add three functions to implementation of `graphAL`: `bfs`, `dfs`, and `shortestPath`. The functions should have the following headers:

```
bool bfs(const node & source, apvector<int> & distances, apvector<node> & bfsTree);
bool dfs(const node & source, apvector<int> & distances, apvector<node> & bfsTree);
bool shortestPath(const node & source, const node & dest, apvector<node> & path);
```

I will now describe each of these functions briefly.

- The function `bfs` takes a source node and performs *breadth-first search* to compute a breadth-first search tree, along with associated distances in the graph of nodes from the source. These two pieces of information are returned as reference parameters by the function. The length of both the vectors, `distances` as well as `bfsTree`, should be identical and should be equal to the number of nodes in the graph. Information in both vectors should be ordered identically; meaning that if the ith slot in `bfsTree` contains the parent information for a certain node, then the ith slot in `distances` should contain the distance information for the same node. Besides this, there are no other constraints on how this information is organized. You should note that not all nodes may be reachable from the source node. In particular, the graph may consist of several pieces such that there are no edges between nodes in one piece and nodes in a different piece. Such pieces are called the *connected components* of a graph. Only the nodes in the same connected component as the source node are reachable from it. For nodes that are not reachable from the source node, the entry in the `distances` vector should be -1 and the parent pointer in `bfsTree` should be `NULL`. This will make it easy to compute the set of nodes, unreachable from the source node, by examining either `distances` or `bfsTree`. Finally, `bfs` returns `false` if the source node does not exist in the graph; `true` otherwise.
- The `dfs` function provides functionality that is very similar to that of `bfs`. It performs a *depth-first search* of the graph and computes a depth-first search tree along with associated distances. Note

that, while the distances computed by `bfs` are shortest distances in the graph, this is not true of the distances computed by `dfs`.

- The function `shortestPath` takes a source node and a destination node and returns a shortest path between the two nodes. This information is returned as a vector of nodes, such that the first node in this vector is the source node and the last node is the destination node. If either the source node or the destination node is missing from the graph the function returns `false`; otherwise the function returns `true`. It is possible that there is no path between the source node and the destination node (this happens if they belong to different connected components). In this case, the length of the returned path vector should be 0.

For `bfs` use the `queue` class defined in the Standard Template Library. It is a little easier to implement `dfs` recursively, so that is what you should do rather than use a stack-based implementation.

4. Testing the Graph Traversal Functions.
You should test the newly implemented graph traversal functions in two ways. First, enhance the driver program from Project 1, so that it can process an input line of the form:

```
S str1 str2
```

It should respond to such a line of input by printing out a *shortest path* from the node with name `str1` to the node with name `str2`. In other words, it should print in order the names of the nodes in a shortest path starting with `str1` and ending with `str2`. If `str1` or `str2` are not valid node names, it should produce an "error message". If there is no path from `str1` to `str2`, it should produce a message saying so.

Second, enhance the ladders program from Project 1 so that it can also read an input line of the form:

```
L w1 w2
```

and respond by printing out the shortest "ladder" from word `w1` to word `w2`.
Both of the above tasks can be achieved by just making an appropriate call to the `shortestPath` function and printing the vector it returns.

5. BFS versus DFS.
Write a program called that performs an experiment to compare the performance of breadth-first search and depth-first search on the ladders graph as follows.

1. Start by reading input from `words.dat` and building the ladders graph.
2. Then, pick a pair of 5-letter words randomly, say w_1 and w_2. Compute the length of the path between w_1 and w_2 returned by `bfs` and the length of the path between w_1 and w_2 returned by `dfs`.
3. Repeat the above step 1000 times and after all 1000 trials of the experiment are completed, report the average length of a path returned by `bfs` and report the average length of a path returned by `dfs`.

So your program reads input from `words.dat` and produces as output two numbers.

6. Comparison of the Two Implementations.
Now I want you to perform an experiment to compare the two graph implementations: `graph` and `graphAL`. Specifically, start by adding the function `bfs` to the `graph` class. This is easy because the function `bfs` that you implemented as part of the `graphAL` class can be used with little or no modification in the `graph` class.

Then implement two versions of a function `generateRandomGraph` that generates a random graph. These functions are not part of any class and have the following function header:

```
graph<int> generateRandomGraph(int n, float p);
graphAL<int> generateRandomGraph(int n, float p);
```

The two functions are identical in all ways except that the first function returns a `graph` object, while the second function returns a `graphAL` object. Both functions, take a positive integer n and a real number p in the range 0 through 1, and return a graph with n nodes and edges generated with probability p. To

generate edges of a graph with probability p, consider each pair of nodes and with probability p connect them by an edge. This can be done easily using the `RandReal` function from the `RandGen` class. Notice that when $p = 0$ we get a graph with no edges and as we increase p the number of edges in the graph increases until, when $p = 1$, the graph has all possible edges. Also notice that `generateRandomGraph` returns graph objects whose nodes are identified by integers. Since it does not really matter how the nodes are identified, they can be numbered $1, 2, \ldots, n$.

For the experiment fix $n = 1000$ and use values of $p = 1/20, 2/20, 3/20, \ldots, 19/20, 20/20$. For each n and p, generate 10 random `graph` objects and 10 random `graphAL` objects. Time the execution of `bfs` on each of the 10 `graph` objects and report the average time of execution. It does not matter what source you use for `bfs`, so you might as well use node 1. Similarly, time the execution of `bfs` on each of the 10 `graphAL` objects and report the average time of execution. So your program reads no input and produces as output two sets of 20 running times, one set of twenty numbers for `bfs` implemented as part of the `graph` class and another set of twenty numbers for `bfs` implemented as part of the `graphAL` class.

9. Some advice.

This project requires that you integrate several concepts that we have explored in class and it is my expectation that it will take you all of 3 weeks to complete it. So please start early and do not hesitate to ask me questions as you make progress.

Here is my advice on how to break up your work on the project into stages.

Stage 1 Implement the `graphAL` class one function at a time. Use the driver program from Project 1 to test the functions in this class. Specifically, after implementing each function, compile your code, and test the new function using the driver program.

Stage 2 Implement `bfs` for the `graphAL` class; compile and test your code. Implement the `shortestPath` function; compile and test your code. Copy the implementation of `bfs` from the `graphAL` class into the `graph` class. Make any modifications that are necessary and compile and test the `bfs` in the `graph` class.

Stage 3 Implement `dfs` for the `graphAL` class; compile and test your code.

Stage 4 Perform the various experiments. Give yourself enough time just to run the experiments.

You should break up work within each stage into pieces and make sure that you implement a piece and test it before moving on to the next piece. You will maximize your grade on this project by using this approach. This is because, it is better to turn in a working program that does a few of the tasks well, rather than a large chunk of code that attempts to do everything, but does nothing.

10. Overall organization.

As in Project 1 submit the following files: `graph.h`, `graph.cxx`, `node.h`, `node.cxx`, `edge.h`, `edge.cxx`, `driver.cxx`, and `ladders.cxx`. The new files, specific to this project are: `graphAL.h`, `graphAL.cxx` which contain the `graphAL` class; `bfsVersusDfs.cxx` which contains the experiment comparing `bfs` to `dfs`; `newBfsVersusOldBfs.cxx` which contains the experiment comparing `bfs` in the `graph` class to `bfs` in the `graphAL` class.

In addition to these files that contain your code, you should also have a `README` file that clearly tells us about any known errors your programs have and also lists all required features not implemented.

All of these files should be in a directory that you will have to submit by 5 pm on 11/13/02.
