Programming Project Three: Graph Algorithms

Out: Nov. 29, 2012; Due: Dec. 16, 2012.

I. Motivation

1. To give you experience in implementing a graph data structure using the adjacency list representation.
2. To give you experience in implementing a few graph algorithms.

II. Programming Assignment

You will read from the **standard input** a description of a graph and then report two things on that graph:

1. The shortest path between a source node and a destination node specified in the input.

2. Whether the graph is a directed acyclic graph (DAG).

1. Input Format

The first line in the input specifies the number of nodes in the graph, N. The nodes in the graph are named from 0 to $N-1$. The second line specifies a source node $0 \leq s \leq N-1$ and the third line specifies the destination node $0 \leq d \leq N-1$. You should report the shortest path from s to d. Each subsequent line represents a **directed** edge by 3 numbers in the form:

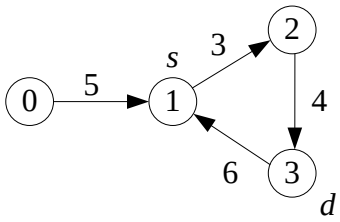
```
<start_node> <end_node> <weight>
```

where both `<start_node>` and `<end_node>` are integers in the range $[0, N-1]$, representing the start node and the end node of the edge, respectively, and `<weight>` is a **non-negative integer** representing the edge weight. Thus, the graph specified in this format is a directed graph with non-negative edge weights.

An example of input is

```
4
1
3
0 1 5
1 2 3
2 3 4
3 1 6
```

It represents the following directed graph with source node as Node 1 and destination node as Node 3:



Typically, we will describe the graph in a file. However, since your program takes input from the standard input, you need to use the Linux input redirection "<" on the command line to read the graph from the file.

2. Output Specification

Your program writes to the standard output. It will first show the shortest path. Then, it will tell whether the graph is DAG or not.

a) Showing the shortest path

If there exists a path from the source node to the destination node, your program should print the length of the shortest path followed by the path. Specifically, it prints the following two lines:

```
Shortest path length is <length>
Shortest path is <s> <v> … <d>
```

The <length> specifies the length of the shortest path, and <s> <v> … <d> are the sequence of the nodes on the shortest path. Note that there is a space after <d>. In the special case that the source node is the same as the destination node, the shortest path is degraded to <s> alone.

If there exists no path from the source node to the destination node, your program should print:

```
No path exists!
```

b) Telling whether the graph is a DAG or not

If the graph is a DAG, your program should print:

```
The graph is a DAG
```

Otherwise, print:

```
The graph is not a DAG
```

For the example graph shown above, the output should be:

```
Shortest path length is 7
Shortest path is 1 2 3
The graph is not a DAG
```

III. Hints on Data Structure

You are required to implement the graph data structure using the adjacency list representation. To simplify your task, you are allowed to use the standard template library. You may include the following three: <vector>, <deque>, and <list>. If you have not used them before, you can search online for their usage. Using them will greatly simplify your task. You are also allowed to include <climits>, which contains some limits such INT_MAX, INT_MIN, etc.

As a hint, you can declare several classes as follows (However, this is not the **required** implementation. You can define your own classes or modify these class definitions.):

1. Node class in node . h

```
#ifndef NODE_H
#define NODE_H

#include <vector>
#include <iostream>

using namespace std;

class Edge;

class Node
{
    int index;               // The index of the node
    vector<Edge*> outEdges;  // The outgoing edges
    vector<Edge*> inEdges;   // The incoming edges
    bool explored;
    int value;
    Node *prev;    // The previous node on the shortest path

public:
    Node();
    Node(int ind);

    friend class Edge;
    friend class Graph;
    friend ostream& operator<<(ostream &os, const Node &v);
    // You can define other methods or data members.
};

#endif
```

2. Edge class in edge.h

```
#ifndef EDGE_H
#define EDGE_H

#include <vector>
#include <iostream>

using namespace std;

class Node;

class Edge
{
    Node *start;  // The start node of the edge
    Node *end;    // The end node of the edge
    int weight;   // The weight of the edge

public:
    Edge();
    Edge(Node *s, Node *d, int w);

    friend class Node;
    friend class Graph;
    friend ostream& operator<<(ostream & os, const Edge &e);
    // You can define other methods or data members.
};

#endif
```

3. `Graph` class in `graph.h`

```
#ifndef GRAPH_H
#define GRAPH_H

#include <vector>
#include <iostream>
#include "node.h"
#include "edge.h"

using namespace std;

class Graph
{
    vector<Node*> nodes;
    vector<Edge*> edges;
    Node* source;
    Node* dest;

public:
    Graph();
    ~Graph();
    // You can define other methods or data members.
};

#endif
```

IV. Program Arguments and Error Checking

Your program takes no arguments. You do not need to do any error checking. You can assume that all the inputs are syntactically correct.

V. Implementation Requirements and Restrictions

- You must make sure that your code compiles successfully on a Linux operating system. You are required to write your own `Makefile` and submit it together with your source code files. **Your compiled program should be named as `p3` exactly.**
- You should name C++ source code files with the extension ".C" (**CAPITAL C**) instead of ".cpp".
- You may not leak memory in any way. To help you check if you are leaking memory, you may wish to call `valgrind`, which can tell whether you have any memory leaks. (You need to install `valgrind` first if your system does not have this program.) Put the following rule in your `Makefile`:

  ```
  memcheck: $(TARGETS)
     valgrind --leak-check=full ./$(TARGETS) $(MEMCHECKARGS)
  ```

  You can check memory leak by typing "`make memcheck`" in your Linux terminal. However, you should set the macro `MEMCHECKARGS` as the arguments of the program you are testing. For example, if you want to check whether running program

  ```
  ./p3 < graph.in
  ```

  causes memory leak, then you should set

  ```
  MEMCHECKARGS = < graph.in
  ```

- You may `#include <iostream>`, `<fstream>`, `<sstream>`, `<string>`, `<cstdlib>`, `<climits>`, `<vector>`, `<deque>`, `<list>`, and `<cassert>`. No other system header files may be included, and you may not make any call to any function in any other library.
- Output should only be done where it is specified.

VI. Testing

We have supplied three input files `g1.in`, `g2.in`, and `g3.in` for you to test your program. The outputs of our program for these three inputs are also provided. They are `g1.out`, `g2.out`, and `g3.out`. All these files are located in the Programming-Project-Three-Related-Files.zip. To do the test, type a similar command to the following into the Linux terminal once your program has been compiled:

```
./p3 < g1.in > test.out
diff test.out g1.out
```

If the `diff` program reports any differences at all, you have a bug.

These are the minimal amount of tests you should run to check your program. Those programs that do not pass these tests are not likely to receive much credit. You should also write other different test cases yourself to test your program extensively.

You should also check whether there is any memory leak using `valgrind` as we discussed above.

VII. Submitting and Due Date

You should submit all the related `.h` files, `.C` files, and the `Makefile`. The `Makefile` compiles a program named `p3`. The due date is 11:59 pm on Dec. 16th, 2012.

VIII. Grading

Your program will be graded along four criteria:

1. Functional Correctness
2. Implementation Constraints
3. General Style
4. Performance

Functional Correctness is determined by running a variety of test cases against your program, checking your solution using our automatic testing program. We will grade Implementation Constraints to see if you have met all of the implementation requirements and restrictions. In this project, we will also check whether your program has memory leak. General Style refers to the ease with which TAs can read and understand your program, and the cleanliness and elegance of your code. For example, significant code duplication will lead to General Style deductions. Part of your grade will also be determined by the performance of your algorithm. Algorithms that run slowly may receive only a portion of the performance points.