CS515 Lab 10

The purpose of this lab is to implement a graph data structure, a graph traversal algorithm, and the Disjoint Set ADT. Download all files from `~cs515/public/10L`

Task 1 (70%): Write a program that determines what the shortest path is between two nodes.

Write a program that reads from standard input a list of inputs which collectively represents a graph, and then answers a list of queries (also from stdin) to give the shortest path between graph node pairs. Graph nodes are represented as ASCII characters. Initial graph input line stores a **directed** edge, represented by a pair of nodes, in which the first node is the starting node of the edge. Later graph query lines start with a question mark and are followed by a pair of nodes, representing a query of path connectivity between the nodes. A sample input file **`data0`** with sample output is shown below.

```
a b
b c
b d
c e
c f
e a
e f
? a e
? b f
? d a
? d f
```

In **`data0`**, the first seven lines represent a directed un-weighted graph of seven edges and six nodes. For example, the line "**`a b`**" indicates there is a directed edge starting at node **`a`** and ending at node **`b`**.

```
$ ./connectChecker < data0
from a to e : path found.
a b c e
from b to f : path found.
b c f
from d to a : path not found.
from d to f : path not found.
```

The next four lines are connectivity queries. The line "**`? a e`**" queries for a path starting node **`a`** to node **`e`**. You can assume that the graph-query lines always follow after the all the graph input lines.

Your task is to write a program that answers the queries based on a given graph, and prints out the shortest path if two nodes are connected. Follow the steps below to write your program:

1. Determine how to represent and store the graph. You are allowed to use STL containers in this lab. For example, you may use the adjacency list and represent the graph using a `Map <char, list<char> >`. In this case, the list might look as follows:
   ```
   a: b
   b: c d
   c: e f
   e: a f
   ```
2. Once the graph is built, you can work on connectivity queries. Choose a graph traversal algorithm to solve this problem, answering whether a pair of nodes are connected or not, and then print out the **shortest** path if one exists. Would you use depth- or breadth-first traversal?

Task 2 (30%): Finish and test a Disjoint Set implementation and convert it to a template class.

The provided starter file contains the header file of a simple **non-templated** implementation of a disjoint set **`DisjointSet0.cpp`**. The disjoint set can only contain integer elements. Your tasks are:

1. Finish the implementation of the data structure Disjoint Set in **`DisjointSet0.cpp`**. You need to implement both the Union by Rank and Path Compression techniques.
2. Test your Disjoint Set by completing the starter program **`DStest0.cpp`**. This testing program should solve the connected-components problem on a graph with nodes labeled 1 through 9, with the edges 3-5, 4-2, 1-6, 5-7, 4-8, 3-7, and 8-1. The graph queries are already present in the program, expressed using assertions. Note that, since the test uses the non-templated disjoint set class, the implementation file should be compiled with separate compilation

   ```
   %g++ DStest0.cpp DisjointSet0.cpp -o DStest0
   ```
3. Write a `DisjointSet` class template. You should include the template declaration and implementation in a single header file named **`DisjointSet.h`**. (this means you don't need to have a separate implementation file `DisjointSet.cpp`.)

4. Test your disjoint set class template. Make a copy of the test file `DStest0.cpp` and name it **`DStest.cpp`**. Then, change the includes in `DStest.cpp` from
   `#include "DisjointSet0.h"`
   to
   `#include "DisjointSet.h"`
   and declare the disjoint set using the template declaration so
   `DisjointSet s;`
   is now replaced with
   `DisjointSet<int> s;`
   finally, compile the test file directly as

```
%g++ DStest.cpp
```

Submission:

- Submit the files **`connectChecker.cpp`**, **`DisjointSet.h`**, **`DStest.cpp`**, and **`README`**.
- The code you submit should compile properly with a `makefile` having the following lines:

```
CXX=g++

all: connectChecker DStest
connectChecker: connectChecker.cpp
	$(CXX)  connectChecker.cpp -o connectChecker

DStest: DisjointSet.h  DStest.cpp
	$(CXX)  DStest.cpp -o DStest
```

- To submit the files in Mimir, follow the link for this lab in MyCourses. Here is a list of files you are expected to submit:
  `connectChecker.cpp DisjointSet.h DStest.cpp README`
- As always, do not turn in executables or object code, and make sure your submission compiles successfully on Agate. Programs that produce compile time errors or warnings will receive a zero mark (even if it might work perfectly on your home computer.) To check this, use the make command with the provided makefile, and check that your submission passes tests on Mimir.

Important:
You must include the standard comment block in each of your source files, including your name, section, date and collaboration details. You must also finish the README file along with your programs.

You should also include detailed collaboration declaration information in your comments. If you worked in pairs in this lab, each of you must include the partner's name in your program comment. Both you and your partner must complete an individual submission in order to earn a grade. If you work by yourself, you must indicate in the program comments that you have worked on your own independently.

You can resubmit as needed—your last submission will be graded. Here is a tentative grading scheme:

Graph connectivity test 1	15
Graph connectivity test 2	15
Graph connectivity test 3	20
Graph connectivity test 4	20
Disjoint Set test run	10
Disjoint Set test run 1	10
Disjoint Set test run 2	10