Project 5 - Tour Guide System

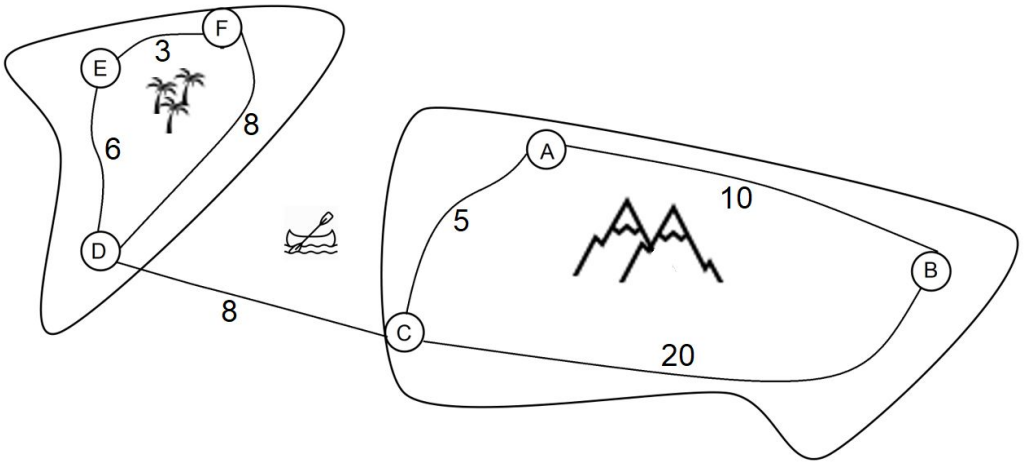
Out: March 25, 2019 @ 8:00 am
Due: April 22, 2019 @ 8:00 am

In project 5, you will be developing some tools for use by a Tourism company to help customers plan their trips and to identify which tourist spots are most important to the company. In the first part, you will implement a system that reads trip information and stores it. Next, you will implement a system to find the cheapest tickets for your customers. Finally, you will generate a list of the company's most visited tourist places.

Part 1: Identification of connected components and separation edges (bridges)

Part 1.1: Building the network (10 points)

In this part you will read in the tourist places and the routes between them. Input will start with a number 1, representing part 1 of the project. You will be given the number of tourist places (n) and the connections among them (m), followed by m pairs of tourist places and the travel ticket price associated with them. Create a network (graph) from the given data and identify sets of connected places. Two places are connected if they have a path between them. You can assume that each of the connections represent a two-way route. That is, if there is a path from A to B, there is also a path from B to A at the exact same price. Here is a small example of the input modelled with 6 tourist spots and routes between them.



Example 1:
1
6 7
A B 10
B C 20
C A 5
E D 6
D C 8
D F 8
F E 3

This information should be stored as a graph in the form of an **adjacency list** for your project; take a look at the class slides for suggestions on possible solutions.

Part 1.2: Identifying the critical routes (30 points)

In this part you will identify the critical routes and biconnected components in the network that you build in part 1.1. Critical routes are analogous to separation edges (bridges) in a graph. A separation edge is an edge whose removal increases the number of connected components.

Note: You must implement an iterative version of DFS for identifying critical routes and connected components -- so no recursion. (You need to convert the recursive implementation given in your textbook to an iterative version).

Your output to the file should be in the following format.

<number of connected components in the graph>

<number of separation edges>

<separation edges listed in lexicographical order>

i.e., in the last example D C is a separation edge. After removing it, we have two separate connected components: {A,B,C} and {D,E,F}. Therefore, the expected output is:
1
1
C D

Also refer to the sample test cases provided in the tests folder (Also make sure to output both the nodes of an edge using lexicographical order. For example in the above example your program should not output <D C>).

Part 1: Input:

The integer "1" followed by two integers n and m, representing the number of places and the number of routes respectively.

Part 1: Output:

<Number of connected components in the graph>

<Number of separation edges>

<Separation edges listed in alphabetical order>

Your program should work in O(m+n) time to not hit a time-out.

Hint:

You can modify DFS to find the separation edges. To do that for each vertex v, define the following evaluation function. It can be calculated in a constant time after visiting all the children of v and possible ancestors through the back edges.

evaFun(v) = **min {** disc(v),
min { evaFun(u) | (v , u) is a tree edge },
min { disc(w) | (v , w) is a back edge}
}

in which disc(v) is the *discovery time* of vertex v. After that vertex v can be popped from the stack.

A tree edge (s, t) is a separation edge if and only if evaFun(t) > disc(s).

Explanation: evaFun(v) represents the earliest discovery time of a node you can get to from v, by going forward along an arbitrary number of tree edges and at most one back edge. The formula can be explained as follows:

min { disc(v), // v can always reach itself, so evaFun(v) <= disc(v)

min { evaFun(u) | (v, u) is a tree edge }, // If one of v's children can get to x using this method, then so can v by taking the same path after going along (v, u)

min { disc(w) | (v, w) is a back edge } //the lowest disc(w) among all back edges (v,w).

}

If an edge (u, v) is a separating edge, then it is necessarily a tree edge, so we only need to test tree edges. What happens if we have a tree edge (u, v) where evaFun(v) > disc(u)? Well, this means that it is impossible to reach u or an ancestor of u through a back edge of one of v's descendants. This means that if (u, v) were removed, it would be impossible to reach u from a DFS started at v, so the number of connected components increases upon the removal, and therefore (u, v) is a separating edge. Each of these steps should be reversible, justifying the bidirectional statement:

A tree edge (u, v) is a separating edge <=> evaFun(v) > disc(u).

Here is the recursive procedure (You need to implement an iterative version of this**)**

```
Procedure DFS(v)

    For all u ∈ adj(v) // For each adjacent vertex of 'v'

        If(!visited[u])

            {

                Eval_func[v] = discovery_time[v];

                //Do DFS on u

                eval_func[v] = min (eval_func[v], eval_func[u])

            }

        else(back_edge)
```

{

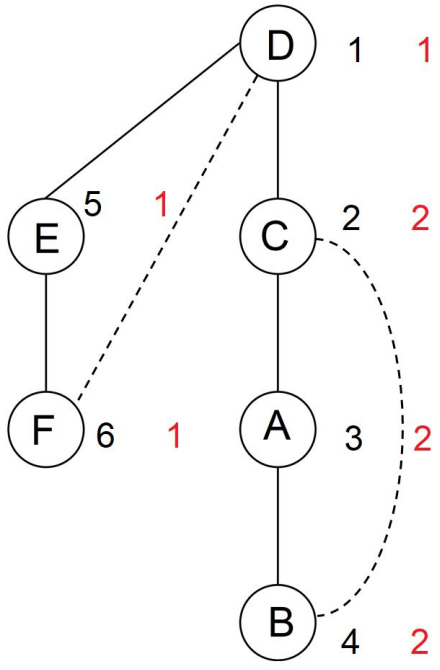
eval_func[v] = min(eval_func[v], discovery_time[u])

}

In the iterative version, whenever you encounter a vertex (i.e., **v**), it should be pushed into the stack. Update the eval_func for the node as described above. If all the children of **v** have been visited, then you should pop **v** from the stack.

Example:

In this example given on page 1, assume DFS started from vertex D. The following picture shows the DFS tree in solid lines and the back edges in dashed lines. There are two numbers next to each vertex, the black one is the discovery time and the red one is the result of the evaluation function for that vertex. Here, the only tree edge whose discovery time of the parent is less than the evaluation function of the child is edge C D, so it is a separation edge of the given graph.



Instructions for part1:

1) **You should store the graph in adjacency list.**
2) **Iteratively implement routines for Separation edges and Connected components (Recursive implementations will not be evaluated or considered). You can use library routines for stack operations.**
3) **There should not be any new line character at the end of the output.**

Computing shortest paths using Dijkstra's Algorithm

Part 2. Finding the cheapest ticket (35 Points)

Now that you have read in data, you will need to find tickets for your customers. In this project someone might want to travel between two places that requires a connection. That is, they may need to travel multiple cities to get to the destination as a direct route may not be available. You need to implement a Dijkstra's shortest path algorithm in a way that lets you search for the cheapest path between two different cities. For simplicity you can assume that there is only one cheapest route between two different cities.

Input/Output for this section will directly follow the network input and is formatted as follows:

<Source> <Destination>

e.g. IND DTW

You should continue reading in ticket queries until you read in the token "END".

Your output should be in the following format:

<source> <list of intermediate nodes> <destination> <cost up to 2 decimal places>.

Part 2: Input:

1. The integer "2" followed by two integers n and m, representing the number of places and the number of routes respectively.

2. m lines in the format <source> <destination> <cost>

3. 1 or more lines containing ticket queries in the form <source> <destination>

4. A line containing “END”.

Part 2: Output:

Each line is the optimal route between the source and destination in the format: <source > ... <intermediate nodes> ... <destination> <total cost to 2 decimal places> or the line “not possible” if it is not possible to reach the destination. Number of lines is equal to the number of queries, and in the same order.

Instructions for part 2:

1) **Use the adjacency list constructed in part 1 for part 2.**
2) **You are allowed to use the library routines for priority queues.**
3) **There should not be any new line character at the end of the output.**

Computing the Spanning Tree and Euler Tour

Part 3: Going on a tour (25 Points)

In part 3 you will be constructing a network in the same way as in part 1. However, this time our objective will be to find a way for a passenger to visit every tourist place offered by the tourist company. It turns out that this problem, a variation of a famous problem known as the travelling salesman problem, cannot currently be solved perfectly in a reasonable amount of time. Therefore, you will be implementing a simplified version of a solution to this problem, rather than an optimal one.

Your solution will be found in two steps. First, given the graph, you will construct a minimum spanning tree using either Prim-Jarnik or Kruskal’s algorithm. The root of this tree (start point of the tour) will be provided in the input. Once you have the minimum spanning tree, you should do an Eulerian tour of the tree. An Euler tour on tree is a walk around the tree where each edge is visited exactly once, and the tour starts and ends at the same vertex. Euler tour in which we visit the nodes on left produces preorder traversal and when we visit the nodes on the right produces

a postorder traversal of the tree. Eulerian tour of the tree can be implemented according to the following pseudocode:

```
Void eulerTour(Minimum_Spanning_Tree G):
      Print root node
      For each child of the root, in alphabetical order:
            eulerTour(child)
```

Part 3: Input:

1. The integer 3 followed by two integers ‘n’ and ‘m’ representing the number of tourist places and number of interconnections among them.
2. ‘m’ lines in the format of <source> <destination> <cost>.
3. One line containing the tourist place that will be the root of MST.

Output: The Eulerian traversal according to the given pseudocode, or the line “not possible” if it is not possible to visit all cities.

Instructions for part 3:

1) **Use the adjacency list constructed in part 1 for part 3.**
2) **You are allowed to use the library routines for priority queues.**
3) **You can implement a recursive program for euler tour.**
4) **While testing, the test cases will be provided such that the output of Prim’s/Kruskal’s algorithms are the same.**
5) **There should not be any new line character at the end of the output.**

Sample Input / Output Format (All input/output for project 5 will go through file IO)

1 3 3 A B 163.30 B C 238.30 C D 239.90	1 3 A B B C C D
2 3 3 A B 163.30 B C 238.30 A C 239.90 A B C A C B END	A B 163.30 C A 239.90 C B 238.30
2 4 2 A B 163.30 C D 150.00 A C END	not possible
3 5 4 IND ORD 163.30 ORD DTW 238.30 DCA DTW 150.00 DTW LAX 300.00 DTW	DTW DCA LAX ORD IND

NOTE:

Important:
1. If your program does not compile, your grade will be 0 - no exceptions.
2. If you don't follow the submissions instructions, your grade will be 0 - no exceptions.
3. Plagiarism and any other form of academic dishonesty will be graded with 0 points as definitive score for the project and will be reported at the correspondent office. We will use MOSS to check your solutions.

C++ Instructions:

Compilation will be done using g++ and makefiles. Create a folder with your username. Put all the source code (should not have any sub-directories in your submission folder) as well as the Makefile that compiles your provided source code into an executable named "program".

Ex: g++ -o program main.cpp stack.cpp circular_deque.cpp -std=c++11
./program <input file> <output_file>

Your project must compile using the standard g++ compiler on data.cs.purdue.edu. For convenience, you are provided with a template Makefile and C++ source file.

Java Instructions:

● Compiler we use: javac (v10.0.2)
● Files to submit: Create a folder with your username. Put all java files into the folder. There should not be any sub-directories. Your main program must be named Main.java. We will run that one only. Your project must compile using the javac compiler (v10.0.2) on data.cs.purdue.edu.

Compiling process:
● To compile Main:
1. Go to the project root.
2. Type command in console: javac src/*.java

● To run Main:
1. Go to the project root.
2. Type command in console: java -cp src/Main inputfiles/test1.txt outputfiles/output-test1.txt

Submission Instructions:
The homework must be turned in by the due date and time using the turnin command.
Follow the next steps:
1. Login to data.cs.purdue.edu (you can use the labs or a ssh remote connection).
2. Make a directory named with your username and copy your solution (makefile, all
your source code files including headers and any other required additional files) there. I.e. all your source files must be present in a single directory under your username (If you fail to do so, your grade will be 0).
3. Go to the upper level directory and execute the following command:
turnin -c cs251 -p project5 your_username
(Important: previous submissions are overwritten with the new ones. Your last submission will be the official and therefore graded).
4. Verify what you have turned in by typing turnin -v -c cs251 -p project5
(Important: Do not forget the -v flag, otherwise your submission would be replaced with an empty one)

We provide you with some skeleton code in this project. It's use is completely optional. You can edit it as you wish.