Program 7 Spring 2018
30 Points

USU Walk Cleaning Problem

A sweeper must be used to remove the debris from all the sidewalks at USU. We want to find the cheapest way of traversing all walkways. We construct a graph of the sidewalks. If that graph has an Euler tour, all edges can be visited exactly once – which is guaranteed to find the most efficient way of cleaning the paths.

We know that we can easily determine if an undirected graph has an Euler tour by simply making sure it is connected and each node has even degree. However, knowing a tour exists does not give us any hints as to how to find the tour. The basic idea is this. Find cycles within the graph and then merge them.

For each of the input graphs (prog7A.txt, prog7B.txt, prog7C.txt) : (a) verify the graph has an Euler tour and (b) print the nodes of the tour. For your convenience, TestEuler.cpp and edge1.h have been provided. Feel free to change these routines as needed.

Notice how the edge class contains all useful methods an edge may need. Notice that parameters and local variables are explained via a comment and that all routines have a brief explanation. This is what is expected when you document a method.

The algorithm for finding an Euler tour is really quite simple.

SETUP

1. Input and store your graph using an adjacency matrix. Count the degree of each node. If any edge has a non-even degree, output “No Euler Tour Available”.
2. For ease in programming, you can assume the graph is connected.
3. Hint: I found it easiest to put all edges in an array of edges. Then, there was only one copy of an edge. Using this technique, the adjacency matrix can refer to edge number (which is just the subscript in the array).

MARK CYCLES

1. Preliminaries: We want to separate the graph into a set of cycles. Thus, we have to find the cycles. To keep track of cycles, with each edge, you will store a label indicating its “cycleID”. Initially all edges do not belong to any cycle. We will refer to an edge being *labeled* (belonging to a cycle) or *unlabeled*.

2. Mark Cycles
   a. Start at any node that has edges that are unlabeled. Let’s call the node B. Start at B, following unlabeled edges. Mark each edge you use as being in the same cycle.
   b. As you find a cycle, print it using node names. For example, edges B->C, C->A, A->B will create the cycle: B C A B
   c. The cycle ends when you return to B. If all edges of the graph have been visited, you are very lucky (and are done).

3. Otherwise, repeat this “Mark Cycles” process, finding additional cycles.

JOINING CYCLES

1. Preliminaries: After the Mark Cycles phase, you will have one or more cycles. If you have only one, you are done. Otherwise, you need to join the cycles. The basic idea is that you start following any cycle, but take "side trips" on other cycles as you go (eventually returning to the same spot).
2. Start traversing the cycles and joining them. It is like you get interrupted. Suppose you are making a cake and get interrupted with the doorbell. You go the doorbell and are asked to help a neighbor start his lawn mower. You help the neighbor. Before you return, you notice the mail has come. You get the mail. Then you return to the front door (and shut it). Then you return to cake making.

   As you use an edge, mark it as used (in the final tour). Keep a stack of cycles you are following. (This can be as simple as a vector.) The top of the stack of cycles always indicates the current cycle (which cycle you are trying to follow). So, if you have five total cycles and the following stack:

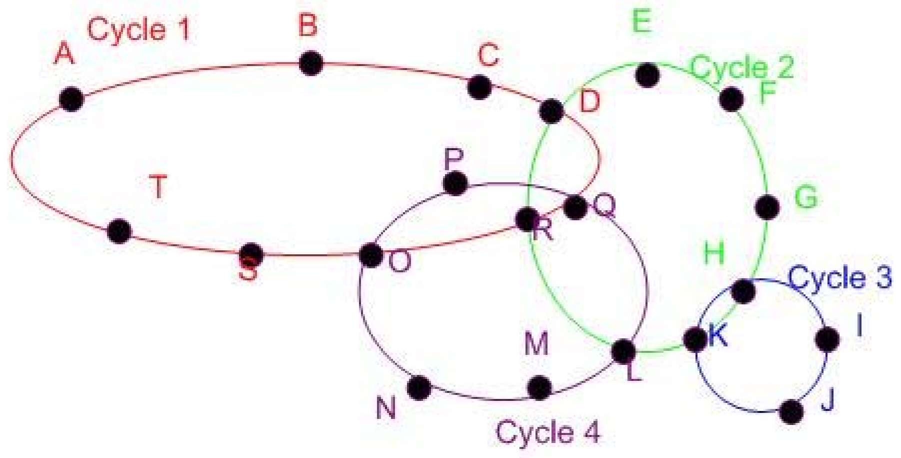
4 (top of stack)
1
2

   4 is the current cycle. 1 and 2 are "old" cycles. Cycle 3 and 0 are new cycles (as you have not yet started following them).

   For example,

   a. Begin with an edge. Mark it as "used".
   b. Suppose the edge is in cycle 2.
   c. Put 2 on the cycle stack
   d. Keep following (and labeling) edges of cycle 2, until you encounter a node with edges of a different cycle (which is not on your stack of cycles). [An edge from a *new cycle* is from a cycle which you have not encountered in this phase.]
   e. When you encounter an edge from a new cycle, put the new cycle name on your stack and start following (and labeling) the edges of the new cycle.
   f. If you encounter an edge of an "old cycle", **you do not follow it** until you complete the current cycle. Then you pop the current cycle off the stack, and start following edges of the top cycle of the stack. So, you always prefer new cycles to the current cycle or to old cycles – but you will only switch from the current cycle if the alternate is new (and not a cycle you have already begun to follow).
   g. Eventually you will complete all the cycles you have begun to follow.
   h. You are done when there are no unused edges.

The following diagram illustrates one possibility.



First Find the cycles (shown in colors) using any kind of random walk.

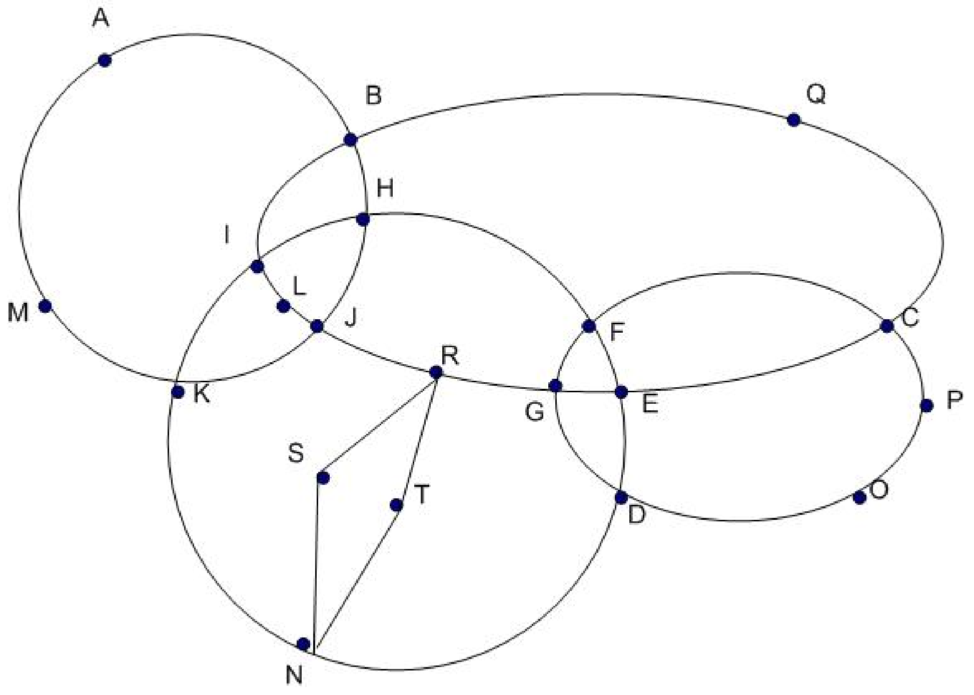
Then join the cycles.

Tour becomes: ABCDEFGHIJKHKLMNOPQ LRDQROSTA

Hints:

1. Be kind to yourself. Draw the graph so you can follow along with the algorithm.

For example, prog7C.txt is shown below. There is no way I would try to debug my code without having a copy of the graph to follow my progress.



2. Start early. This assignment is a bit involved. You will want to work on it when you are relaxed and thinking clearly.
3. If you find yourself saying, "I wish there was an easy way of doing X" - write the routine to do X. I had a routine getChar(int node) which converted the node number to a character. There are those who continue to do something in an awkward way because they refuse to take the time to create a method to do the task. They are too busy sawing to sharpen the saw.
4. Let the code talk to you. I had the printGraph routine show the cycle number of an edge so whenever I wanted to see the progress, I could printGraph and see the complete state.
5. In joining the cycles, I had a routine getNextEdge which took my stack of cycles I was tracing, my current node, and returned the edge I should follow. All of the complicated logic was in one routine that I could use without having to think of everything that was involved. Using that abstraction made it so much easier to think about the solution. My getNextEdge found three different edges as I traced through the edge list:
    a. An unused edge from a new cycle (not in my stack of cycles)
    b. The FIRST unused edge from my current cycle (in the cycle at the top of my cycle stack)
    c. The FIRST unused edge from the previous cycle (in the cycle one down from the top of the cycle stack)

Then, if an edge of type (a) exists, I just return it. If there are no type (a) edges, I return an edge of type (b). As a last resort, I return an edge of type (c). If that doesn't exist, I return a -1. Finding all three edge types at once made it so I didn't have to traverse the list multiple times.

This is more than you wanted to know, but trust me: Sometimes you may have a point that touches many edges from the same cycle. For example, perhaps all the edges at N (above) are in the same cycle. If your getNextEdge method returns the FIRST edge you encounter (in the adjacency matrix from N) you will be happy. This forces you to traverse the cycle in the same way as you created it. Not doing so can cause parts of the graph to be unreached. For example: Think of a figure 8. If you first traverse it as an 8 and then follow only the bottom circle, you will miss the top part of the figure 8.

SAMPLE OUTPUT

```
Cycle 0: A B F C D A

Cycle 1: G B E C G

Graph from prog7A.txt
  (print matrix to show edges and cycle numbers)

TOUR D A B G C E B F C D
```