CS 170 Algorithms HW 4
Spring 2014 E. Mossel

1. (15 pts.) Reachability from single vertex

Consider any graph G. In the metagraph of G, if there are two (or more) source strongly connected components, then there can never be a vertex which can reach both of the source SCCs. So, if G is one-way connected, then there must be only one source SCC, and the special vertex v must be in this source SCC.

Moreover, any vertex u in this source SCC can reach v, and so all vertices in G are also reachable from u. So, if G is one-way connected, then all vertices can be reached from any vertex in the source SCC of G.

This gives the following algorithm - run a DFS on G from any starting node, and find the vertex v with the highest post number (which must be in a source SCC). Then run a DFS from v and check that all vertices are reachable from v. There are 2 DFS's, so this takes $O(|V|+|E|)$ time.

Clearly, if the algorithm returns true, then G is one-way connected (it actually finds the special vertex v). If the algorithm returns false, we know that there is a vertex in a source SCC of G which cannot reach all vertices, and by the preceding discussion this means that G is not one-way connected.

2. (20 pts.) DFS edge types in BFS

(a) Suppose we have a forward edge (u,v) from vertex u with depth k in the BFS tree. Since (u,v) is a forward edge, v is a non-child descendant of u and so has depth $\geq k+2$.

BFS has the property that the shortest path from the starting vertex s to a vertex v with depth k in the BFS tree has k edges. So, the shortest path to v has at least $k+2$ edges, and the shortest path to u has k edges. However, if we take the shortest path to u and add the edge (u,v), we get a path to v with $k+1$ edges. Contradiction.

(b) In the corresponding algorithm for DFS, we use the pre and post numbers to classify edges. The key insight is that if we remove the non-tree edges from the graph and run DFS again, we will get *exactly the same pre and post numbers*. This is because in normal DFS, any non-tree edge is ignored, and so it has no effect on the pre and post numbers. So, another valid way of classifying edges for DFS is:

1. Run DFS on the graph and construct the DFS tree.
2. Run DFS on the DFS tree to generate pre and post numbers.

3. Classify edges using these pre and post numbers.

Now if we look at steps 2 and 3, we can see that it does not depend at all on the fact that we are using a DFS tree - it would work for any tree. In particular, it would also work for the BFS tree, which gives the following algorithm:

1. Run BFS on the graph and construct the BFS tree.
2. Run DFS on the BFS tree to generate pre and post numbers.
3. Classify edges using these pre and post numbers.

The correctness of this algorithm follows immediately from the correctness of the algorithm for classifying edges in DFS.

Construction of the BFS tree takes $O(|V|+|E|)$ time. Running DFS on the BFS tree takes $O(|V|)$ time (since a tree has $O(|V|)$ edges). Classifying each edge takes $O(|E|)$ time. So, the algorithm takes $O(|V|+|E|)$ time in total.

3. (15 pts.) Odd Cycle

Consider the case of a strongly connected graph first. The case of a general graph can be handled by breaking it into its strongly connected components, since a cycle can only be present in a single SCC. We proceed by coloring alternate levels of the DFS tree as red and blue. We claim that the graph has an odd cycle if and only if there is an edge between two vertices of the same color (which can be checked in linear time).
If there is an odd cycle, it cannot be two colored and hence there must be a monochromatic edge. For the other direction, let u and v be two vertices having the same color and let (u,v) be an edge. Also, let w be their lowest common ancestor in the tree. Since u and v have the same color, the distances from w to u and v are either both odd or both even. This gives two paths p_1 and p_2 from w to v, one through u and one not passing through u, one of which is odd and the other is even.
Since the graph is strongly connected, there must also be a path q from v to w. Since the length of this path is either odd or even, q along with one of p_1 and p_2 will give an odd length tour (a cycle which might visit a vertex multiple times) passing through both v and w. Staring from v, we progressively break the tour into cycles whenever it intersects itself. Since the length of the tour is odd, one of these cycles must have odd length (as the sum of their lengths is the length of the tour).

4. (15 pts.) Multiple Shortest Paths

We perform a BFS on the graph starting from u, and create a variable $\texttt{num\_paths}(x)$ for the number of paths from u to x, for all vertices x. If $x_1, x_2, \ldots x_k$ are vertices at depth l in the BFS tree and x is a vertex at depth $l+1$ such that $(x_1,x), \ldots, (x_k,x) \in E$ then we want to set $\texttt{num\_paths}(x) = \texttt{num\_paths}(x_1) + \ldots + \texttt{num\_paths}(x_k)$. The easiest way to do this is to start with $\texttt{num}_\texttt{p}\texttt{aths}(x) = 0$ for all vertices $x \neq u$ and $\texttt{num\_paths}(u) = 1$. We then update $\texttt{num\_paths}(y) = \texttt{num\_paths}(y) + \texttt{num\_paths}(x)$, for each edge (x,y) that goes down one level in the tree. Since, we only modify BFS to do one extra operation per edge, this takes linear time. The pseudocode is as follows

```
function count_paths($G,u,v$)
    for all $x \in V$:
        dist($x$) = $\infty$
        num_paths($x$) = 0

    dist($u$) = 0
    num_paths($u$) = 1
    $Q$ = [$u$]
    while $Q$ is not empty:
        $x$ = eject($Q$)
        for all edges $(x,y) \in E$
            if dist($y$) = dist($x$) + 1:
                num_paths($y$) = num_paths($y$) + num_paths($x$)
            if dist($y$) = $\infty$:
                inject($Q,y$)
                dist($y$) = dist($x$) + 1
                num_paths($y$) = num_paths($x$)
```

5. (20 pts.) Shortest Cycle

Define matrix D so that D_{ij} is the length of the shortest path from vertex i to vertex j in the input graph. Row i of the matrix can be computed by a run of Dijkstra's algorithm in time $O(|V|^2)$. So we can calculate all of D in time $O(|V|^3)$. For any pair of vertices u,v we know that there is a cycle of length $D_{uv}+D_{vu}$ consisting of the two shortest paths between u and v and that this cycle is the shortest among cycles containing u and v. This shows that it suffices to compute the minimum $D_{uv}+D_{vu}$ over all pairs of vertices u,v to find the length of the shortest cycle. This last operation takes time $O(|V|^2)$, so the overall running time is $O(|V|^3)$.

6. (15 pts.) Graph Construction

We can construct a graph where the nodes are the states, and there is a directed edge from state a to state b if by one crossing we get state b from a. We name the the states using four-bit binary strings, where the four bits correspond to F, W, S, and C respectively, 0 means East, and 1 means West. For example, the state 0101 is the state where F, S are at the east bank, and W, C are on the west bank. There are $2^4 = 16$ states in total, among them 10 states are safe (i.e. nothing will be eaten).

$$\{0000, 1010, 0010, 1110, 1011, 0100, 0001, 1101, 0101, 1111\}$$

There are 10 edges in total

$$0000 \to 1010, 1010 \to 0010, 0010 \to 1110, 1110 \to 0100$$
$$0100 \to 1101, 1101 \to 0101, 0101 \to 1111, 0010 \to 1011$$
$$1011 \to 0001, 0001 \to 1101$$

(a) By examining the graph, it is easy to identify two paths from 0000 to 1111 involving 7 river crossings, and they are the shortest ones.

$$0000 \to 1010 \to 0010 \to 1110 \to 0100 \to 1101 \to 0101 \to 1111$$

$$0000 \to 1010 \to 0010 \to 1011 \to 0001 \to 1101 \to 0101 \to 1111$$

(b) From above, we know there are 2 such solutions.