Homework 6
Fundamental Algorithms, Fall 2002, Professor Yap

Due: Wed Dec 11 (last recitation) or Thu Dec 12 (to one of us).

1. Biconnected Component (20 Points)
   Do parts (a) to (d) in Exercise 22-2, page 558 of CLR Text.
   SOLUTION:
   (a) Let \( u_0 \) be the root of \( G_c \). We prove this in two directions. First, suppose \( u_0 \) has more than two children. Let two of the children be \( v \) and \( v' \). There are no edges of the form \( (w,w') \) where \( w \) is any node in the subtree rooted at \( v \) and \( w' \) is any node in the subtree rooted at \( v' \). This is because the DFS tree for a bigraph has no cross edges. Hence any path from \( w \) to \( w' \) must go through \( u_0 \). Therefore, if we delete \( u_0 \) (and all the edges incident on \( u_0 \)) we would have disconnected \( w \) from \( w' \). By definition, this means \( u_0 \) is an articulation point. Conversely, suppose \( u_0 \) has one child. Then clearly, deleting \( u_0 \) will not disconnect the graph \( G \). So \( u_0 \) is not an articulation point.
   (b) Let \( v \) be a nonroot vertex. Again we show the two directions separately. First suppose \( v \) has a child \( s \) as described in the problem. Then removing \( v \) would disconnect \( s \) from the parent of \( v \), and so \( v \) is an articulation point. Conversely, if there is no such \( s \), we claim that \( v \) is not an articulation point. To see this, suppose \( u_0 \) is the parent of \( v \) and \( u_1, \ldots, u_m \) are all the children of \( v \). Then by our assumption, there exists path from \( u_i \) to \( v \) for each \( i = 1, \ldots, m \). This means that the set \( S = \{ v_0, v_1, \ldots, v_m \} \) are all in the same connected component. Hence the removal of \( v \) did not disconnect these vertices. Clearly, any path that goes through \( v \) must go through two nodes in the set \( S \). It follows that if there is a path from any node \( u \) and \( u' \) in the original graph \( G \), then there is still a path from \( u \) to \( u' \) in the graph after we remove \( v \).
   (c) We can modify the DFS-VISIT algorithm in the text (p.54) to maintain the values \( low[u] \) for each \( u \). First, we initialize \( low[u] = d[u] \) (in line 3.5). Subsequent, we update \( low[u] = \min\{low[u], low[v]\} \) in line 7.5 (inside the “then” clause).
   (d) We simply run the DFS-VISIT algorithm from any starting node \( v_0 \). We add line 8.5 to DFS-VISIT to output \( u \) as an articulation point if \( low[u] = d[u] \). Finally, we also output \( v_0 \) as an articulation point if it has more than one child.

2. Graph Diameter (20 Points)
   Let \( G = (V,E) \) be a bigraph, assumed to be connected. The diameter of \( G \) is \( \max_{u,v \in V} \delta(u,v) \), where \( \delta(u,v) \) is the length of the shortest path between \( u \) and \( v \) (what we also called “link distance” in class). Give an efficient algorithm to estimate the diameter within a factor of 2, i.e., your
algorithm must return a number \( D \) such that the diameter of \( G \) lies in the interval \([D, 2D]\)? Bound the running time.

SOLUTION:

Do BFS at any node \( v \), and return \( D \) where the depth of the BFS tree is \( D \).

Why is this correct? Clearly, the diameter is at least \( D \) since there is a shortest path of length \( D \) in the graph. Furthermore, the diameter must be at most \( 2D \) since any two nodes \( u, v \) in the graph can be connected by a path from \( u \) to the root of the BFS tree, and from the root to \( v \). The running time is \( O(m) \) to do BFS.

3. Dijkstra's Algorithm (5+20 Points)

Consider running Dijkstra's algorithm on the graph in Figure 24.6 (page 596, CLR Text). However, instead of the weights there, you must add a positive integer \( \Delta > 0 \) to each weight. We want you to choose the smallest \( \Delta \) such that the order in which nodes that becomes "known" is different than the original order, which is \((s, y, z, t, x)\). You should try \( \Delta = 1, 2, 3 \), etc until you see a different order emerging.

What to hand in: tell us what \( \Delta \) is, and submit a table showing your simulation of Dijkstra. The table is rather similar to Prim's algorithm in the previous homework.

CONVENTIONS: the data for each row of your table should correspond to this order: \((s, t, x, y, z)\). The first row is \((0, 10 + \Delta, \infty, 5 + \Delta, \infty)\). To fill in the \( i \)th row, you first copy the SMALLEST weight in the \((i - 1)\)st row that is still "unknown" and underline it. Then you proceed to fill in the rest of the rows (use double quotes (") to indicate a repeated value, and leave blank those entries corresponding to "known" nodes).

SOLUTION:

(a) What is the minimum \( \Delta \) to cause a different order? We check that if \( \Delta = 1, 2 \), then the order does not change. However, if \( \Delta = 3 \), then we will have a tie for a minimum. By breaking the tie one way or another, we get different orders. Of course, one of them will be different from the original order. Hence \( \Delta = 3 \) is the minimum we seek. If we want to ensure the order is different regardless of how the ties are broken, then we will need \( \Delta = 4 \). So we will accept either answer.

(b) Here is the simulation of Dijkstra for \( \Delta = 3 \).

<table>
<thead>
<tr>
<th>Vertices:</th>
<th>s</th>
<th>t</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1:</td>
<td>0</td>
<td>13</td>
<td>( \infty )</td>
<td>8</td>
<td>( \infty )</td>
</tr>
<tr>
<td>re Stage 2:</td>
<td>&quot;</td>
<td>20</td>
<td>8</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Stage 3:</td>
<td>13</td>
<td>17</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 4:</td>
<td>&quot;</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 5:</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4. **Bellman-Form Algorithm (5+20 Points)**

The Bellman-Ford algorithm detects negative cycles (p.588, CLR). Suppose you also want to know all those vertices that are in a negative cycle. How do you modify the algorithm? HINT: keep track of the shortest paths using the $\pi[u]$ array (cf.p.584).

**SOLUTION:**

First, let us solve a slightly simpler problem than is posed in this question. Assume we just want to detect all vertices $u$ such that the length of shortest path from $s$ to $u$ is $-\infty$. This can be done as follows: in line 7, instead of returning FALSE, we simply set $d[v] = -\infty$.

We also replace line 8 by another call to DFS from each node $v$ where $d[v] = -\infty$. Every node that is reaches by these DFS’s will have their $d$-value set to $-\infty$.

Unfortunately, there seems to be no simple way to detect only those $v$ such that $v$ is contained in a negative cycle. One way to do what we want is to first partition the vertices into strong components. Then for each strong component $C$ either every vertex in $C$ is contained in a negative cycle, or none of them are. To decide which is the case, we note that $C$ contains a negative cycle iff some vertex in $C$ has its $d$-value equal to $-\infty$ in the modified line 7 above.

5. **Shortest Path (20 Points)**

Consider the min-cost path problem in which you are given a digraph $G = (V,E;C_1,\Delta)$ where $C_1$ is a positive cost function on the edges and $\Delta$ is a positive cost function on the vertices. Intuitively, $C_1(i,j)$ represents the time to fly from city $i$ to city $j$ and $\Delta(i)$ represents the time delay to stop over at city $i$. A jet-set business executive wants to construct matrix $M$ where the $(i,j)$th entry $M_{i,j}$ represents the “fastest” way to fly from $i$ to $j$. This is defined as follows. If $\pi = (v_0,v_1,\ldots,v_k)$ is a path, define

$$C(\pi) = C_1(\pi) + \sum_{j=1}^{k-1} \Delta(v_j)$$

and let $M_{i,j}$ be the minimum of $C(\pi)$ as $\pi$ ranges over all paths from $i$ to $j$. Please modify the Floyd-Warshall Algorithm, to compute $M$ for our executive.

**SOLUTION:**

Define $C^{[k]}(i,j)$ to be the minimum cost path from $i$ to $j$ in which the intermediate vertices must come from the vertices $1,\ldots,k$. Then we have

$$C^{[k]}(i,j) = \min \{C^{[k-1]}(i,j), C^{[k-1]}(i,k) + \Delta(k) + C^{[k-1]}(k,j) \}$$

Of course, $M_{i,j} = C^{[n]}(i,j)$.
Now, place this update instruction for $C^{[k]}(i,j)$ inside the usual Floyd-Warshall algorithm.

The algorithm will take $O(n^3)$ time.