ESO207A: Data Structures and Algorithms

Homework *3* *Due Date: 23rd Oct in class*

Instructions.

1. Start each problem on a new sheet. For each problem, Write your name, Roll No., the problem number, the date and the names of any students with whom you collaborated.
2. For questions in which algorithms are asked for, your answer should take the form of a short write-up. The first paragraph should summarize the problem you are solving and what your results are (time/space complexity as appropriate). The body of the essay should provide the following:
   (a) A clear description of the algorithm in English and/or pseudo-code, where, helpful.
   (b) At least one worked example or diagram to show more precisely how your algorithm works.
   (c) A proof/argument of the correctness of the algorithm.
   (d) An analysis of the running time of the algorithm.

   Remember, your goal is to communicate. *Full marks will be given only to correct solutions which are described clearly.* Convoluted and unclear descriptions will receive *low marks.*

Problem 1. Bipartite Graphs. A *bipartite* graph is an undirected graph $G = (V, E)$ whose vertices can be partitioned into two sets $V = V_1 \cup V_2$, where, V_1 and V_2 are disjoint subsets, and there are no edges between vertices in the same set (i.e., for any $u, v \in V_1$, there is no edge between u and v.

(a) A graph G is said to be colorable with k-colors, if each vertex is given one of the k colors and no two adjacent vertices are given the same color. (Adjacent vertices are given different colors). Show that a graph is bipartite iff it is 2-colorable.

[only if]: Color all vertices in V_1 as blue and all vertices in V_2 as red. By definition, all edges are from V_1 to V_2 and hence adjacent vertices have different color. Hence G is 2-colorable.

[if]: Suppose G is 2-colorable. Let V_1 be the set of vertices with the first color and V_2 be the set of vertices with the second color. Since adjacent vertices are differently colored, all edges are between V_1 and V_2. Hence G is bipartite.

(b) Show that an undirected graph is bipartite if and only if it contains no cycles of odd length.

[only if]: Let $G = (V_1, V_2, E)$ be a bi-partite graph. Suppose there is a simple cycle $v_0, v_1, v_2, \ldots, v_k, v_0$. Let $v_0 \in V_1$. Since, edges are only between V_1 and V_2, it follows that $v_1 \in V_2$, $v_2 \in V_1$, and so on, that is, $v_i \in V_1$ iff i is even and $v_i \in V_2$ iff i is odd. Hence, k is odd. The cycle length is $k + 1$ which is even.

An easier way to see is that starting from the initial vertex v_0, every second vertex in the cycle returns to V_1, hence the only possible cycle length is some multiple of 2.

[if:] Suppose G is connected. Do a BFS on G starting from any vertex say s. Let V_1 be the set of vertices u with $u.d$ even and V_2 be the set of vertices u with $u.d$ odd. In any BFS of a directed graph, if there is an edge between u and v, then, either $u.d = v.d$ or $u.d$ and $v.d$ differ by at most 1. (Why?) Hence, the proposed partition of vertices into odd and even distances is bipartite iff there is no edge between vertices at the same distance from root.

Suppose to the contrary that there is an edge between u and v such that $u.d = v.d$. So, we have a path consisting of d BFS tree edges from s to u, say, $P_1 = s, v_1, v_2, \ldots, v_{d-1}, u$ and d tree edges from s to v, say $P_2 = s, w_1, w_2, \ldots, w_{d-1}, v$. Consider the cyclic path $P_1, \{u, v\}, P_2^R$, where, P_2^R is the reverse sequence of edges of P_2. From this path, remove the maximum common prefix of vertices P_1 and P_2, that is say $s, v_1, \ldots, v_k$ is the largest common prefix of both P_1 and P_2, for $k = 0, 1, \ldots, d-1$.

Now we obtain the simple cycle

$$v_k, \ldots, v_{d-1}, u, v, w_{d-1}, \ldots, w_k = v_k$$

The number of edges on this cycle is $2(d-k)+1$ which is an odd cycle. But G had no odd cycles, and hence this cannot happen. So there is no edge between two vertices at the same distance from the root.

(c) Give a linear time $O(|V|+|E|)$ algorithm to determine whether an undirected graph is bipartite.

1. Run BFS on G. If u has just been dequeued and the edge $\{u, v\}$ is being traversed and $v.d == u.d$, then the graph is not bipartite; return.
2. If BFS terminates gracefully, then the connected component of G is bi-partite.
3. Repeat BFS from all non-white vertices.

```
BFS(G)
1.  for each vertex $v \in V$
2.      $v.color = white$
3.      $v.d = \infty$
4.  for each vertex $v \in V$
5.      if $v.color == white$
6.          $BFS\_Explore(G, v)$

BFS_Explore(G, v)
1.  $v.color = gray$
2.  $v.d = 0$
3.  $MakeEmptyQueue(Q)$
4.  $Enqueue(Q, v)$
5.  while $Q$ is not empty
6.      $u = Dequeue(Q)$
7.      for each vertex $w$ in $AdjList[u]$
8.          if $w.color == white$
9.              $w.color = gray$
10.             $w.d = u.d + 1$
11.             $Enqueue(Q, w)$
```

To check for bipartiteness, add the following lines after line 11.

```
12            else // $w.color \neq white$
13.                if  $u.d == w.d$
14.                    return  FALSE    // not bipartite
```

Change the main *BFS* routine to something like the following.

```
$BFS(G)$
1.  for each vertex $v \in V$
2.      $v.color = white$
3.      $v.d = \infty$
4.  for each vertex $v \in V$
5.      if  $v.color == white$
6.          if not  $BFS\_Explore(G, v)$
7.              return FALSE
8.  return TRUE
```

(d) If an undirected graph has exactly one odd cycle, what is the minimum number of colors needed to color it.

Run a BFS on the graph, and consider the component of G that has the odd cycle. First color all vertices at even distance from the root as blue, and those at odd distance from the root by green. Now, by the above argument, since there is exactly one odd cycle, there is exactly one pair of vertices u, v that are equi-distant from the root such that $\{u, v\}$ is an edge. Make one of them red. Now, except for $\{u, v\}$ all edges $\{s, w\}$ are between vertices that whose distance from the root differs by exactly 1, and so no two end-points have the same color. Thus, 3 colors suffice.

Problem 2. Diameter of a tree A *tree* (also called a free tree in the Appendix of the CLRS book) is an undirected, acyclic and connected graph $T = (V, E)$. Design a linear time algorithm to compute the diameter of a given unweighted tree T, that is,

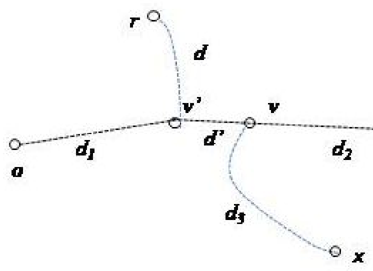
$$\text{diameter}(T) = \max_{u,v \in V} \delta(u, v)$$

where, $\delta(u, v)$ is the length of the shortest path from u to v. The algorithm should also maintain enough information to report a simple path whose length equals the diameter.

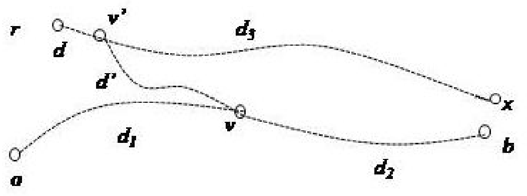
In a tree, all simple paths (paths that do not repeat any edges) are unique between any pair of vertices. Hence, we will only refer to lengths of simple paths between u and v as the length of the shortest path between them.

Let $L = a, u_1, u_2, \ldots, u_{k-1}, b$ be a longest simple path between any two vertices in T. Let r be an arbitrary vertex of the tree. Suppose we run BFS on T from r. Let x be the vertex with the highest distance from r as per BFS. Let the BFS shortest path from r to x be P.

Case 1: P and L have some common vertex. Along the path P starting from r, let v' be the first vertex that is common with L and v be the last vertex that is common with L. The path segment $v \ldots v'$ is the overlap between P and L and is of length d', which may be 0. See Figure. Without loss of generality assume that d_1, the distance from a to v', is at most d_2, the distance from b to v (see Figure, the other case is symmetric).



Case 1



Case 2

Hence, $d_3 = d_2$. Why? $d_3 \geq d_2$ since $d + d' + d_3$ is the longest distance from r which must be at least $d + d' + d_2$, the distance of r to b. Also, a to b is a longest simple path in T and has size $d_1 + d' + d_2$, whereas, the path from a to x has length $d_1 + d' + d_3$ and so $d_1 + d' + d_2 \geq d_1 + d' + d_3$ or that $d_2 \geq d_3$. So, the length of the simple path from a to x is the diameter (its length is $d_1 + d' + d_3 = d_1 + d' + d_2$, the diameter) which can be found by taking the largest distance of a BFS starting at x.

So the algorithm is as follows.

1. Run BFS from r. Let x be the vertex with the largest d value at the end of BFS.
2. Run BFS from x. Let a be the vertex with the largest d value at the end of this BFS. Then $a.d$ is the diameter.
3. The path from x to a can be obtained from the *pred* attribute of the second BFS.

Case 2: Suppose P and L have no common vertex. Let v be the vertex on the path L with the least BFS distance from r and let the path r to v share an initial segment of length d' from r to v' along P. Let the segment v to v' have length d'. As before, without loss of generality, let $d_1 \leq d_2$ (the other case is symmetric). The path from r to b via v' and v is of length $d + d' + d_2$. This is at most the length $d + d_3$ of the path from r to x, since x is the vertex with the largest distance from r. So

$$d + d' + d_2 \leq d + d_3 \ .$$

The length of the path from x to b via v' and v is $d_3 + d' + d_2$. Since, $d_1 \le d_2$, by assumption, we have,

$$d_3 + d' + d_2 \ge (d' + d_2) + d' + d_2 = 2d' + d_1 + d_2 \ .$$

This is a contradiction unless $d' = 0$ since the path from a to b is the longest path and has length $d_1 + d_2$, which should be no shorter than the path from x to b. So $d' = 0$, that is, paths P and L do intersect, or that, this case cannot occur.

Problem 3. DFS Basics.

1. Give a counterexample to the conjecture that if a directed graph G contains a path from u to v and $u.d < v.d$ in a DFS of G, then, v is a descendant of u in the depth-first tree produced.
2. Show by an example that the DFS of a directed graph with a vertex u can end up as a solitary vertex u in the DFS forest of G, even though u has incoming and outgoing edges from it.

Problem 4. [Problem 22-2 of CLRS: Articulation points, bridges and biconnected components] This question concerns an undirected, connected graph $G = (V, E)$. An **articulation point** or a *cut-vertex* of G is a vertex whose removal disconnects G. A **bridge** of G is an edge whose removal disconnects G. A **biconnected component** of G is a maximal set of edges such that any two edges in the set lie on a simple cycle. Design an algorithm to determine the articulation points, bridges and biconnected components using DFS. Let $G_T = (V, E_T)$ be a depth first tree of G.

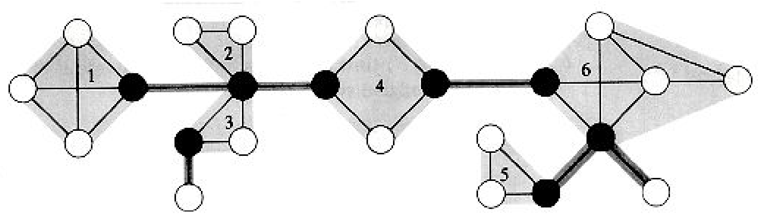


Figure 1: Shaded edge sets are biconnected components. Dark vertices are articulation points and darkened edges are bridges.

(a). Prove that the root of G_T is an articulation point of G iff it has two children in G_T.

Let r be the root vertex of some DFS tree G_T. The DFS of an undirected graph results in only tree edges and back edges; there are no forward or cross edges. [if:] Suppose r has two children. Then the two children subtrees have no edges between them and get disconnected from each other if r is removed. Hence, r is an articulation point. See Figure .

[only if:] If r has only one child s, then, r can be deleted and the graph remains connected with s now being the root of the DFS tree consisting of $V - \{s\}$ and all edges incident to s removed.

(b). Let v be a non-root vertex of G_T. Show that v is an articulation point of G iff v has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v.

Let v be a non-root vertex of G_T. Let $\pi(v)$ be the parent of v. [if:] Suppose v has a child s such that there is no back edge from s or any descendant w of s to any proper ancestor of

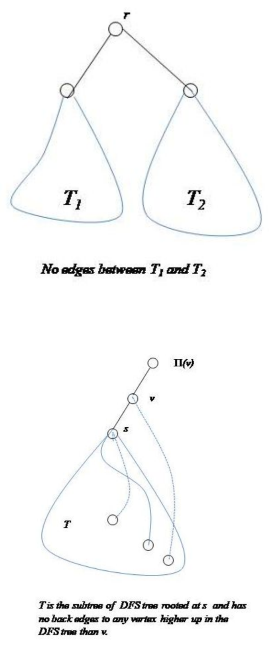

r
T_1
T_2
No edges between T_1 and T_2
$\Pi(v)$
v
s
T
T is the subtree of DFS tree rooted at s and has no back edges to any vertex higher up in the DFS tree than v.

Figure 2: A non-root articulation point in DFS tree

v. Then, by deleting v, there is no connectivity between s and $\pi(s)$. So v is an articulation point. See Figure 2

[only if:] Suppose every child s of v is such that there is a backedge from s or some descendant of s to some ancestor of v. Let v have k children $T_1, T_2, \ldots T_k$. As we know there are no cross edges between T_i and T_j. Each T_i is obviously connected (by tree edges), and there is at least one vertex say $v_i \in T_i$ that has a back edge to a *proper* ancestor of v. Let $V_1 =$ the set of vertices in $T_1 \cup T_2 \ldots \cup T_k$ and $V_2 =$ the set of vertices in the remaining part of the DFS tree, except $\{v\}$. v and its incident edges are now removed from G. The subgraph $G[V_2]$, that is, the graph induced by V_2 is a tree, a subtree of the original DFS tree and is therefor connected. For any vertex $u \in T_i$, u has a path via the tree edges to v_i and then via the back edge reaches some vertex w in V_2. Same is true for any vertex $u' \in T_j$– it reaches some vertex $w' \in V_2$. Since $G[V_2]$ is connected, therefore, there is a path between w and w'. Thus, there is a path between u and u'. Hence, any vertex in $V_1 \cup V_2$ is connected with any vertex in $V_1 \cup V_2$. Thus the graph $G[V - \{v\}]$ is connected and so v is not an articulation point.

(c). Let

$$v.low = \min \begin{cases} v.d \ , \\ w.d : \ (u, w) \text{ is a back edge for some descendant } u \text{ of } v. \end{cases}$$

Show how to compute $v.low$ for all vertices v.

Note that the graph is connected, so our top-level DFS can start from any one vertex.

```
$DFS(G)$
1.  for each vertex $v \in V$
2.        $v.color = white$
3.  $time = 0$
    // Given that $G$ is connected
4.  Choose any vertex $s$ of $V$ and call $DFS\_Explore(G, s)$
```

```
$DFS\_Explore(G, u)$
1.  $u.color = gray$
2.  $time = time + 1$
3.  $u.d = time$
4.  $u.low = u.d$
5.  for each vertex $v \in AdjList[u]$
6.        if $v.color == white$
7.              $v.\pi = u$
8.              $DFS\_Explore(G, v)$
9.              $u.low = \min(u.low, v.low)$
10.       elseif $v.color == gray$
11.             $u.low = \min(u.low, v.d)$
12. $time = time + 1$
13. $u.f = time$
14. $u.color = black$
```

(d). Give an algorithm to compute all articulation points in time $O(|E|)$. A vertex u is an articulation point iff either it is the root vertex and has multiple children, OR, it is a non-leaf vertex and $u.low = u.d$.

u is root iff $u.d$ is 1. Checking whether u has multiple children can be done by maintaining a children count for each node $u.nchildren$ which is initialized to 0, and is incremented by 1 whenever a white neighbor is encountered in *DFS_Explore* in the pass over the adjacency list.

u is a leaf vertex if $u.d = u.f + 1$.

```
$DFS\_Artp(G)$
1.  for each vertex $v \in V$
2.        $v.color = white$
3.        $v.nc = 0$        //$nc$ is number of children in DFS tree
4.  $time = 0$
5.  $art\_pt\_list = NIL$        // list of articulation points
6.  $s$ = Choose a vertex of $V$
7.  $DFS\_Explore\_Artp(G, s)$        // Given that $G$ is connected, DFS from one vertex suffices
8.  if $s.nc > 1$
9.        $art\_pt\_list.Insert(s)$      //insert $s$ at head of list
```

```
$DFS\_Explore\_Artp(G, u)$
1.  $u.color = gray$
2.  $time = time + 1$
```

```
3.    $u.d = time$
4.    $u.low = u.d$
5.    $is\_art\_pt$ = FALSE
6.    for each vertex $v \in AdjList[u]$
7.        if $v.color == white$
8.            $v.\pi = u$
9.            $u.nc = u.nc + 1$
10.           $DFS\_Explore(G, v)$
11.           $u.low = \min(u.low, v.low)$
12.           if $v.low >= u.d$ and $u.d > 1$
13.               $is\_art\_pt$ = TRUE
14.       elseif $v.color == gray$
15.           $u.low = \min(u.low, v.d)$
16.   $time = time + 1$
17.   $u.f = time$
18.   $u.color = black$
19.   if $is\_art\_pt$
20.       $art\_pt\_list.Insert(u)$
```

(e). Show that an edge of G is a bridge iff it does not lie on any simple cycle of G.

[only if:] Suppose an edge $e = \{u, v\}$ lies in a simple cycle and G is a connected graph. Then deleting e still leaves the graph connected and so e is not a bridge. [if:] If $e = \{u, v\}$ does not lie on any simple cycle of G, then it is the unique path from u to v (if there was another path not involving e, then, there would be a simple cycle involving e). So removing the edge disconnects u from v.

(f). Give an algorithm to compute all the bridges of G in time $O(|E|)$.

If $\{u, v\}$ is a bridge, then it cannot be a back edge in the DFS tree. Why? Because a backedge lies on some simple cycle. So a bridge edge $\{u, v\}$ is tree edge. Suppose $u.d < v.d$. Then, $v.low$ must equal $v.d$. Conversely, if $v.low \geq u.low$ then $\{u, v\}$ lies on a cycle and $\{u, v\}$ is not a bridge. An inline modification to the *DFS_Explore* procedure is shown below.

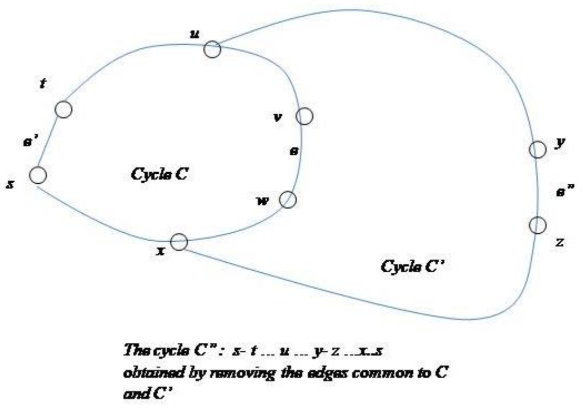
```
$DFS\_Explore\_Bridges(G, u)$
1.    $u.color = gray$
2.    $time = time + 1$
3.    $u.d = time$
4.    $u.low = u.d$
5.    for each vertex $v \in AdjList[u]$
6.        if $v.color == white$
7.            $v.\pi = u$
8.            $DFS\_Explore(G, v)$
9.            if $v.low == v.d$
10.               print " {u,v} is a bridge edge "
11.           $u.low = \min(u.low, v.low)$
12.       elseif $v.color == gray$
13.           $u.low = \min(u.low, v.d)$
```

```
14. $time = time + 1$
15. $u.f = time$
16. $u.color = black$
```

(g). Prove that the biconnected components of G *partition* the non-bridge edges of G.

Let E_b be the set of non-bridge edges of G. Define a binary relation B (biconnection) between edges as follows: Given edges $e, e' \in E_b$, let eBe' hold if $e = e'$ or e and e' lie on a simple cycle in G. Clearly B is a reflexive and symmetric relation. To see that it is transitive, let $e'Be$ hold and eBe'' hold. Then, there is some simple cycle C_1 on which both e' and e lie. Also, there is some (other perhaps) simple cycle C' on which e and e'' lies. We have to show that there is a simple cycle denoted C'' on which e and e'' lies. See Figure 3.

Firstly, if $C = C'$, then e, e' and e'' lie on C and we are done. Otherwise, C and C' have some common edges including e. Remove all the common edges between C and C' and the resulting set of edges is a simple cycle. See Figure 3.



The cycle C" : s- t ... u ... y- z ...x..s
obtained by removing the edges common to C
and C'

Figure 3: Obtaining a simple cycle from two simple cycles with a non-empty intersection of edges

(h). Give an $O(|E|)$ algorithm to label each non-bridge edge e of G with a positive integer $e.bcc$ such that $e.bcc = e'.bcc$ iff e and e' lie in the same biconnected component.

We will design a 2-pass algorithm. First, we run the *DFS* procedure starting with some source vertex s to compute $u.d$ and $u.low$ for every vertex $u \in V$. Next, we run another DFS starting from the same source vertex and using the same order of the adjacency lists as used by the first one. Only, this time, when we encounter a vertex v, we will have information of $v.d$ and $v.low$ already with us.

Corresponding to each edge (corresponding entry in the adjacency list structure) e, we keep a field $e.bcc$, which is initialized to *NIL* say. Finally, the *NIL* values will correspond to bridges. Also for bookkeeping purposes, with every vertex w, keep a field $w.pbcc$, which is the number of the biconnected component *bcc* to which the edge $\{\pi(w), w\}$ belongs. If this edge is a bridge, then $w.pbcc$ is set to *NIL*.

Suppose during the second DFS, the algorithm is at vertex u (currently on the top of the stack) and suppose we traverse the *tree* edge $\{u, v\}$.

Case 1: If $v.low > u.d$, then $\{u, v\}$ is a bridge edge.

Case 2: If $v.low = u.d$, then, $\{u, v\}$ is the first edge seen of a new biconnected component. Accordingly, a counter *bccnum* (initialized to 0) is incremented, and corresponding to the edge $e = \{u, v\}$, the field $e.bcc$ is set to *bccnum*. Also, $v.pbcc$ is set to *bccnum*.

Case 3: if $v.low < u.d$, then, the edge $e = \{u, v\}$ belongs to the same biconnected component as its parent tree edge, that is $\{\pi(u), u\}$. In this case, set $e.bcc$ is set to $u.pbcc$ and $v.pbcc$ is also set to the same.

Now suppose while visiting u, we traverse a back edge $e = (u, v)$. Every back edge lies in a biconnected component, since it creates a simple cycle with the tree edges leading from v to u. So, we set $e.bcc$ to be $u.pbcc$.

These actions can be incorporated into pseudo-code.