Homework Assignment #4

Due: Noon, 27th Nov, 2017
Submit **printed** solutions via eClass
Submit **handwritten** solutions at dropbox on CSC level 1

CMPUT 204
Department of Computing Science
University of Alberta

Note: All logs are base-2 unless stated otherwise.
You should answer **at least 3 out of Problems 1-4**, and **must answer Problem 5**. Should you answer all of the first 4 problems, your grade will be composed of the best 3 out of 4.

Exercise I. Let G be a graph whose vertices are the integers 1 through 8, and let the adjacent vertices of each vertex be given by the table below:

vertex	adjacent vertices
1	$\{2, 3, 4\}$
2	$\{1, 3, 4\}$
3	$\{1, 2, 4\}$
4	$\{1, 2, 3, 6\}$
5	$\{6, 7, 8\}$
6	$\{4, 5, 7\}$
7	$\{5, 6, 8\}$
8	$\{5, 7\}$

(a) Draw G.

(b) Write G in the adjacency matrix model.

(c) Order the vertices as they are visited in a BFS traversal starting at vertex 1. Draw the BFS tree.

(d) Order the vertices as they are visited in a DFS traversal starting at vertex 1. Draw the DFS tree.

Suggestion. If you are having issues with understanding the way BFS/DFS operate, repeat this question with other graphs!

Problem 1. (20 pts) Given a graph G (directed or undirected) it's 2-closure is the graph H such that $V(H) = V(G)$ and for any two nodes $u, v \in V(H)$ we put an edge if in G we have that v is reachable from u by a path of length 1 or 2. (Therefore, it is clear that any $e \in E(G)$ will remain an edge also in $E(H)$.)

(a) (10 pts) Give a *deterministic* algorithm that takes as an input a graph G with max-degree of Δ in the *adjacency list* model and outputs its 2-closure in $O(n\Delta^2)$-time.

Note: our construction of a Hash-Table, for example, is inherently random.

Answer. We basically invoke BFS-visit from every node, where we only insert into the queue the initial node s and its neighbors, and every node that becomes gray we insert into the list of neighbors of s in the 2-closure. Note that we just need to re-traverse this list to re-color all of the nodes `WHITE` so that when we iterate over a neighbor of s we will insert s into its new (in H) adjacency list.

```
procedure Build2Closure(G)      ** G = (V, E)
foreach v ∈ V do
   v.color ← WHITE              **unknown yet
   v.dist ← ∞                   **distance from s
foreach v ∈ V do
      v.L ← new list of neighboring nodes
      Find2Neighbors(G, v, L)


procedure Find2Neighbors(G, s, L)
Initialize a queue Q            **waiting vertex queue
s.color ← GRAY                  **in queue Q
s.dist ← 0
enqueue(Q, s)
while (Q ≠ ∅) do
   u ← dequeue(Q)
   foreach neighbor v of u do
      if (v.color = WHITE) then
         Insert(L, v)     ** add v as a neighbor of s in the 2-closure
         v.color ← GRAY           **discovered v
         v.dist ← u.dist + 1
         if (u.dist ≤ 1) then
            enqueue(Q, v)
   u.color ← BLACK              **done with u
end while
s.color ←WHITE
s.dist ← ∞
foreach (u ∈ L) do
   u.color ←WHITE
   u.dist ← ∞
```

Note how in our algorithm we only insert s and its immediate neighbors (those nodes with $dist = 1$) to Q, so per a call of the latter algorithm, the queue will have no more than $1 + \Delta$ nodes. For each node u in the queue we do at $O(1 + deg(u)) = O(1 + \Delta)$ work, hence per each call of `Find2Neighbors(s)` we do $O((1 + \Delta) \cdot (1 + \Delta)) = O(\Delta^2)$ work. All in all, we do $O(n\Delta^2)$ work.

Correctness follows from the correctness of BFS: since we traverse exactly the top 2-layers from every node s (L_1 = nodes of distance 1 from s, L_2 =nodes of distance 2 from s) then we do add to L all of those nodes, namely, in new adjacency list there will be all of the nodes in $L_1 \cup L_2$. Finally, note that at the end of `Find2Neighbors()` we reset all the nodes to their original state, and thus we actually do follow BFS for 2-layers from every node.

(b) (10 pts) Give an algorithm that takes as an input a graph G in the *adjacency matrix* model and outputs its 2-closure in $o(n^3)$-time.

Hint: We call it an adjacency *matrix* for a reason...

Answer. Let A be the adjacency matrix of G. Since it is a matrix, we can multiply it with itself, so let's look at $A^2 = A \cdot A$.

For every pair of disjoint nodes u, v, we have $(A^2)_{u,v} = \sum_w A_{u,w} \cdot A_{w,v}$. As $A_{u,w}, A_{w,v} \in \{0, 1\}$ we have that $(A^2)_{u,v} = 0$ iff for every node w either $A_{u,w} = 0$ or $A_{w,v} = 0$. Namely, for every possible path (u, w, v) one of the two edges is missing, and therefore there is no path of length 2 connecting u to v.

Therefore, we construct the adjacency matrix of H, denoted M, using the following procedure:

1. We multiply $A \cdot A$ using any fast matrix-multiplication algorithm (best one: $O(n^\omega)$ for $\omega = 2.37...$, but you could use Strassen as all we needed is a $o(n^3)$-time algorithm).

2. For every u, v we set $M_{u,v} = 1$ if either $A_{u,v} = 1$ or $(A^2)_{u,v} > 0$. Since we only check two matrix entry per pair of nodes we do $O(1)$-work per pair of nodes and $O(n^2)$ work overall.

The total runtime of our algorithm is thus $O(n^\omega + n^2) = O(n^\omega)$ and therefore $o(n^3)$.

Problem 2. (20 pts) A vertex s of a *directed* graph $G(V, E)$ is called a sink if for every vertex $v \neq s$, it holds that $(v, s) \in E$ and $(s, v) \notin E$. In other words, every vertex has an edge to s and no edges leave s. The graph is given by an adjacency matrix A.

Give an algorithm takes as an input a directed graph G and either finds a sink or returns "no sinks" in only $O(n)$ time. Argue that your code is correct.
Notice that a running time of $O(n)$ is remarkable given that the input can have potentially $O(n^2)$ edges.
Hint: It is enough for your algorithm to find a single candidate for being a sink, then verify whether it is indeed a sink or not. Make sure your algorithm runs in $O(1)$ time per candidate elimination.
Answer. To verifty whether a node s is a sink, we traverse all other nodes in G and verify that indeed the edge (v, s) exists and the edge (s, v) does not exists. This takes $O(1)$ time per node v, so traversing all nodes requires $O(n)$ time.

```
procedure VerifySink(s)
foreach (u ∈ V) do
   if (u ≠ s and ((s,u) ∈ E or (u,s) ∉ E) ) then
      return FALSE
return TRUE
```

Here is an implementation of finding a sink that uses a queue. It is definitely not the only implementation, it is just convenient to use queue, as it make sure we only do $O(1)$ work per node.

```
Procedure FindSink(V, E)
candidate ← ⊥
Insert all nodes into a queue Q
while (Q isn't empty) do
   v ←Dequeue(Q)
   if (cadidate = ⊥) then     ** There's no candidate for a sink currently
      candidate ← v
   else
      if ((candidate, v) ∈ E and (v, candidate) ∈ E) then
         candidate ← ⊥    ** Both edges exist, so neither can be a sink
      if ((candidate, v) ∉ E and (v, candidate) ∉ E) then
         candidate ← ⊥    ** Both edges don't exist, so neither can be a sink
      if ((candidate, v) ∈ E and (v, candidate) ∉ E) then
         candidate ← v    ** There's only an edge from the candidate to v, so v might be a sink
      ** If there's only an edge from v to candidate exists we don't do anything — keep current candidate
end while
if (candidate = ⊥ or VeriftySink(candidate) = FALSE) then
   return ''no sink''
else
   return candidate
```

Obviously, the runtime of the algorithm is $O(n)$ as for any node in Q we only do $O(1)$ work (verifying whether an edge exists or not in the adjacency matrix model takes only constant time); and at the end we run `VerifySink()` which takes $O(n)$.

Correctness:
If there's no sink in the graph, regardless of what our queue-based vertex traversal does, the candidate cannot be a sink, so by verifying it we find out that it is not a sink and we output no-sink.
If there's a sink in the graph, denoted s, then when s is dequeued from Q — regardless of whether we have a current candidate or not — s then becomes the candidate. From that point on, $candidate = s$ and none

of the cases of the `if` hold as for any node v we have only the edge (v, s). So once we are done traversing the nodes, $candidate = s$. By verifying it, we make sure we output s.

Problem 3. (20 pts) Give an algorithm that solves mazes. The maze is a rectangular maze with r rows and c columns, given in the form of a function `IsWall(`i, j, D`)` that tells you, in $O(1)$-time, whether you can or cannot move from square (i, j) in direction D (where $D \in \{\texttt{Up}, \texttt{Down}, \texttt{Left}, \texttt{Right}\}$). You are also given a start position $s = (i, j)$ and a finish position $f = (i', j')$. Your goal is to find the shortest path taking you from s to f (if such a path exists). Your algorithm must run in $O(r \cdot c)$ time.
Explain why your algorithm is correct and why its runtime is $O(r \cdot c)$.

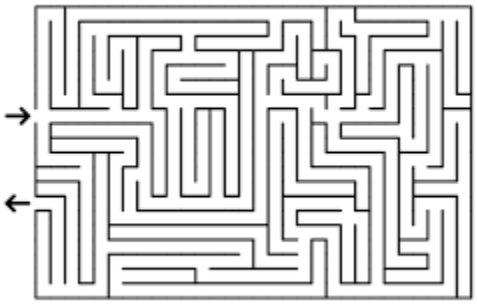

Figure 1: A rectangular maze.

Answer. We create a graph representing this maze and run BFS on this graph from the starting node (or end node).

1. Create the graph:

   (a) Per square (i, j) in the graph, we put a (i, j)-node — so all in all $|V| = rc$.

   (b) For each (i, j) we check all 4 directions and if we can move at a certain direction (there isn't a wall and we don't leave the limits of the maze) then we put an edge connecting the two respective nodes.

   Since we check only 4 possible directions per node, then we can only have $4rc$ edges at most, and so $|E| \leq 4rc/2 = 2rc$. Moreover, building this graph's adjacency lists takes $O(1)$ work per vertex, so it takes $O(rc)$-times overall.

2. Invoke BFS from the finish-vertex f until we hit the start-vertex (or until BFS stops and we figure out we cannot reach the start vertex).

   This takes $O(|V| + |E|) = O(rc)$.

3. Start at the start node s, and print the path from it to the root of the BFS tree (which should be the finish-vertex).

   This takes $O(|V|)$ as the BFS-tree has at most $|V| - 1$ edges.

By the properties of the BFS, this gives a shorts $s - f$ path, so we have found the shortest path connecting s and f.

Problem 4. (20 pts) Recall how we increase the capacity of a stack (or a queue, or a hash-table) once a `Push()` call is invoked on a full stack — using `IncreaseCapacity()`:

1. Allocate a new array with $2 \times$ *stack.capacity* cells
2. Copy all items from the original array to the new array, delete the old array

Analogously to this, we can decrease capacity and clear some memory once *stack.size* is small in comparison to its *stack.capacity* — using `DecreaseCapacity()`:

1. Allocate a new array with *stack.capacity*$/2$ cells
2. Copy all items from the original array to the new array, delete the old array

Note that both `IncreaseCapacity()` and `DecreaseCapacity()` run in $\Theta(n)$-time.

(a) (5 pts) Show that invoking `DecreaseCapacity()` whenever the number of elements in the stack (i.e., *stack.size*) is *stack.capacity*$/2$ is unwise. That is, show that there exists a sequence of n `Push()` / `Pop()` instructions whose amortized cost is $\Omega(n)$.

Answer. Consider a sequence that fills a stack with $\frac{n}{2}$ calls of `Push()`, each takes just $O(1)$ time. The next `Push()` doubles the capacity of the stack to n, but sets the size as $\frac{n}{2}+1$. The next `Pop()` decreases the size of the stack to $\frac{n}{2}$, which invokes the capacity-decrease. Repeating these two instructions for a total of $n/2$ instruction results in overall cost of $n/2 \cdot n = \frac{1}{2}n^2 = \Omega(n^2)$ on a sequence of n instructions. Thus, the amortized cost of this sequence is $\geq \frac{n+\frac{1}{2}n^2}{n} = \Omega(n)$.

(b) (15 pts) Prove that invoking `DecreaseCapacity()` whenever *stack.size* $=$ *stack.capacity*$/3$ maintains constant amortized cost. That is, show that any sequence of n `Push()` / `Pop()` instructions has amortized cost of $O(1)$. You may assume the initial Stack capacity is 1.

Here is a suggested outline:

Call an instruction *heavy* if it invokes either `IncreaseCapacity()` or `DecreaseCapacity()`; and call it *light* otherwise. Light instructions always take $O(1)$. Your argument should probably begin by showing that between any two heavy instructions there have to be many light instructions (by considering all 4 possible cases), where m denotes the capacity of the stack between the two heavy instructions. Then use this claim to prove, by induction, that $T(n)$ — defined as the worst-case cost of a sequence n `Push()`/`PoP()` instructions — is in $O(n)$.

Answer. We claim that between any two heavy instructions we must have at least $\frac{m}{6}$ light instructions. We prove this claim using case analysis. We always assume that the first of the two heavy calls created a stack of capacity m.

- Both calls increase capacity: In this case, the first call created a stack with capacity m, so prior to the first call the stack capacity was $m/2$. This means the first call creates a stack where $m/2$ cells are full. Therefore, to invoke capacity increasing call again, we had to do at least $m/2$ `Push()` instructions between the two heavy instructions.
- Both calls decrease capacity: In this case, the first call created a stack with capacity m, so before it the capacity was $2m$ and the decrease-capacity call was invoked because only $2m/3$ of cells were full. So in our m-capacity stack we have $2m/3$ entries. Therefore, to decrease the capacity again we had to take out additional $m/3$ elements of the stack, so there had to be a least $m/3$ light `Pop()` instructions in between the two heavy calls.
- First call increases capacity, second call decreases capacity: The first call created a stack with capacity m and just $m/2$ elements in it, so in order to invoke a capacity decrease we had to take out additional $m/6$ elements, so there had to be $m/6$ light `Pop()` instructions.

- First call decreases capacity, second call increases capacity: The first call creates a stack with capacity m with $2m/3$ elements in it. To increase capacity we need to fill the stack, so there had to be addition $m/3$ light `Push()` instructions between the two calls.

With this claim, we can prove by induction that $T(n) \in O(n)$. Let C be a constant large enough such that any light instruction takes less than C time, and any heavy instruction on a stack with capacity m takes $\leq Cm$ time. We argue that $T(n) \leq 7Cn$ for any n. This will prove the claim as the amortized cost of each instruction would be $\frac{T(n)}{n} \leq 7C = O(1)$.

Base case: $n = 1$, the first instruction is light. Moreover, also consider as a base case the first heavy instruction — since the initial capacity is 1, then the first time we alter the capacity of the stack takes $\leq C \cdot 1 = C \leq 7C$ time steps.

Induction step: Assume $T(n) \leq 7Cn$, we show $T(n) \leq 7C(n+1)$. Consider the last instruction. If it is light, then it takes only C time, so $T(n+1) = T(n) + C \leq 7Cn + C \leq 7Cn + 7C$. If it is heavy, then let m be the capacity of the stack once we begin the instruction, so the cost of the $n+1$ step is $\leq Cm$. However, all $m/6$ operations before it were light, so they took altogether no more than $Cm/6$. Thus $T(n+1) \leq T(n - m/6) + C \cdot \frac{m}{6} + Cm \leq 7C(n - m/6) + 7Cm/6 \leq 7Cn \leq 7C(n+1)$. ■

Problem 5. (40 pts) Given a connected undirected graph $G = (V, E)$ on n nodes and $m \geq n - 1$ edges, we introduce the following definitions.

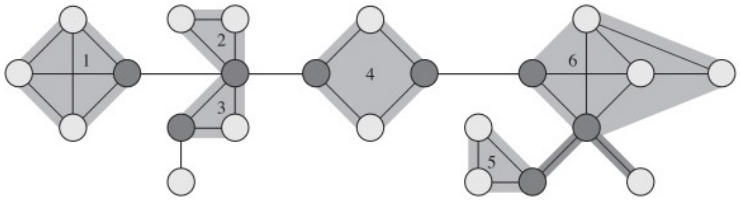

Figure 2: The articulation points (heavily shaded vertices), and biconnected components (shaded regions with numberings) in a toy example.

An articulartion point is a node v such that the removal of v and its adjacent edges leaves G unconnected.

A bridge is an edge $e = (u, v)$ such that the removal of e leaves G unconnected.

Biconnected pair of vertices $u, v \in V(G)$ is a pair of vertices with *two* vertex-disjoint paths from u to v. In other words, there's a simple cycle containing both u and v.

A biconnected component is a maximal set of **edges** such that each pair of edges lie on some simple cycle.

In this question, we show how DFS finds all vertices / edges which satisfy the above definitions.

(a) (3 pts) In the DFS-tree of G, prove that the root is an articulation point if and only if its has more than one child.

Answer. Let v be the root of the DFS tree. Assume v has only one child. Therefore, by removing v the remainder of the tree remains connected and thus spans all remaining $n - 1$ vertices, hence v is not an articulation point.

Assume v has two or more children $u_1, u_2, ...u_k$ each is the root of its respective subtree $T_1, T_2, ..., T_k$. Since the graph is undirected, the DFS-tree is such that there are no cross edges. Therefore, all edges that touch a node in T_1 must either remain within T_1 or must be a back-edge connected to an ancestor of u_1. Since v is the only ancestor of u_1, it means that any edge that connects a node in T_1 with a node in $V(G) \setminus T_1$ must connect it to v. Hence, by removing v and all of the edges that are adjacent to v, we disconnect T_1 from the rest of the graph. Thus, v must be an articulation point. ■

(b) (9 pts) In the DFS-tree of G, fix a non-root node v and its child u. We say u is *v-bypassing* if there exists a back edge connecting either u or a descendant of u with an ancestor of v (an ancestor that cannot be v itself). We say v is a *fully-passed node* is all of its children are v-bypassing.

Prove that a non-root v is an articulation point if and only if it is not a fully-passed node.

Answer. Let v be a non-root node on the DFS tree. We argue that if v isn't fully passed, then it is an articulation point, and that if it is fully passed then it is not an articulation point.

Suppose v is not fully passed. Therefore, there exist a child u such that there are no back-edge connecting u or any of its descendants with a proper ancestor of v. We argue that removing v and all of the edges that are adjacent to v leave the subtree rooted at u disconnected from the rest of the graph. To see that, observe that all edges that touch this subtree are either tree edges or back-edges. As u don't v-pass no edge connect a descendant of u with an ancestor of v. Thus, any path connecting u with an ancestor of v must go through v. By removing v from the graph, we therefore disconnect u and its descendants from the rest of the graph.

Suppose v is fully passed. Let u be any child of v and w the parent of v (as v is a non-root). We argue that there exists a graph path connecting u with w. Thus, if v is removed, there still exists a path connecting any descendant of v with any other node (use the suitable $u \to w$ path, and the remaining tree edges to connect to the rest of the graph). Since u is a v-passing node, there exists a back-edge (a, b) from a descendant of u to an ancestor of v. Thus, the tree-path $u \to a$, along with the edge (a, b) and the (maybe 0-length) tree-path $b \to w$ remain in the graph even after removing v. This concludes the claim.

(c) (1 pts) Given the DFS tree rooted at a node s, we define for each node v the attribute $v.dist$ =distance between s and v on the DFS-tree. Argue that A BFS traversal on the DFS-tree assigns each node its $v.dist$ attribute in $O(n)$ time. (Note the runtime is independent of the number of edges in G.)

Answer. Given the DFS-tree, we create a new graph G in the adjacency list model. For each node $v \in V(G)$ we put a node v into the new graph, and we then add the edge connecting v with $v.predec$ by adding $v.predec$ to the adjacency list of v and adding v to the adjacency list of $v.predec$ (both take $O(1)$ time). Thus we have created a new graph with n nodes and $n-1$ edges. Running BFS on this graph (starting from the roof of the DFS-tree) produces the desired $v.dist$ attribute, and this takes $O(n+(n-1)) = O(n)$-time on this new graph with only $n-1$ edges.

(d) (7 pts) We also define

$$v.\ell = \min\{v.dist;\ \ w.dist \text{ for some back-edge connecting a descendant } u \text{ of } v \text{ with some } w\ \}$$

Show how to find $v.\ell$ for all vertices in G in $O(n+m)$ time. Argue the correctness and runtime of your algorithm.

Answer. First, we compute the DFS-tree using DFS. This takes $O(n+m)$ time. Another pass on the tree finds the children of each node v using:

```
foreach x ∈ V(G) do
   if (x.predec ≠ NIL) then
      add x to the list of children of x.predec
```

Then we compute ℓ recursively, starting with the leafs of the tree.

```
Procedure FindL(v)
v.ℓ ← v.dist
foreach u, child of v do
   FindL(u)
   if (u.ℓ < v.ℓ) then
      v.ℓ ← u.ℓ
foreach w, neighbor of v, do
   if (v.predec ≠ w and w.dist < v.ℓ) then     ** we MUST ignore the tree edge between v and v.predec
      v.ℓ ← w.dist
```

First, it is clear that `FindL(root)` invokes a single `FindL()` call for each node, and that therefore, we traverse all edges of the graph. For each node v we do $O(1) + O(\#children(v)) + O(\deg(v))$ work in addition to the recursive calls. Thus, overall we do $O(n + 2\sum_v \deg(v)) = O(n+m)$ work.

Secondly, as for correctness, we prove that `FindL(v)` sets the right value for $v.\ell$, by induction on the height of the node v.

In the base case, v is of height 0, so it is a leaf. Hence `FindL(v)` sets $v.\ell$ as the minimum of $v.dist$ and $u.dist$ for any neighbor of v. Note that all edges leaving v are backedges except for the edge connecting v with its predecessor, which we ignore. In the induction case, fix any node v of height h and assume we set $u.\ell$ correctly for all children of v which are of height $h-1$. Thus, before traversing the back-edge leaving v, we set $v.\ell$ as the minimum between $v.dist$ and $w.dist$ for any back-edge leaving a proper-descendant of v (any descendant which is not v itself). Thus, it remains to check the back-edges leaving v itself. Node

that v now has tree-edges that we do check — any edge connected v with its child is considered by the code. However, if u is a child of v then $u.dist > v.dist$ (by definition), so we will not set $v.\ell$ as $u.dist$ for a child u of v.

(e) (5 pts) Give an algorithm that finds all articulation points of G in $O(n+m)$ time. Argue your algorithm correctness and runtime.

Answer. The algorithm first does DFS and finds the DFS tree. Then it finds the children of each node and finds ℓ for each node using the previous article. Now, it finds the set of articulation point.
For the root, it checks whether or not it has two or more children, and if so, marks it as an articulation point.
For any leaf v, it marks the leaf v as a non-articulation point.
For any non-root and non-leaf node v, the algorithm checks to see whether $u.\ell < v.dist$ for all children u of v. If v has a child u for which $u.\ell \geq v.dist$ then v is an articulation point, otherwise, it is not an articulation point.

The algorithm is correct for the root due to article (a). It is correct for a leaf because removing a leaf leaves the rest of the DFS tree connected and thus leaves the graph connected. The reason why it is correct for a non-root non-leaf node is that we have for any parent v and a child v that $u.\ell < v.dist$ iff u is v-bypassing. If u is v-bypassing, then the backedge from a descendant of u to a proper ascendant w of v sets $u.\ell$ at least as small as $w.dist$; If u isn't v-bypassing, then all back-edges from the descendants of u lead to either v or children of v, so $u.\ell$ must be at least $v.dist$.

Clearly, the algorithm runs in $O(n+m)$ as this is the runtime of DFS and `FindL()`. The marking of articulation point takes $O(1)$-time per node, and can be done in $O(n)$ overall. Thus the runtime is $O(n+m)$.

(f) (5 pts) Prove that an edge $e = (u, v)$ is a bridge iff it doesn't lie on any simple cycle in G.

Answer. If $e = (u, v)$ is on a simple cycle, then its removal leaves u and v connected by the remaining edges along the simple cycle. This leaves the graph connected as it was connected before the removal of e and so any path that previously used the edge e can now use the alternative $u \to v$ path that is still there in the graph. Hence, e is not a bridge.

If e is not a bridge, then its removal leave the graph connected. In particular, u and v remain connected by some path $u \to v$ that doesn't use e. Iteratively remove any cycles along this path to derives a simple $u \to v$ path. Then this path together with e closes a simple cycle: $u = v_0, v_1, ..., v_k = v$.

(g) (5 pts) Give a $O(n+m)$ algorithm for finding all bridges in G.

Answer. Again, we run DFS. Again, in an undirected connected graph, we have a single DFS tree and only tree- and back-edges (no cross-edges). So it is evident that any non-tree edge is a back-edge and thus lie on some cycle, and therefore cannot be a bridge. It remains to check for each tree edge between a parent v and a child u whether (u, v) lies on a cycle or not. To that end, we again use $u.\ell$. If $u.\ell \leq v.dist$ then there has to be an back-edge from a descendant of u connecting it to an ancestor of v, and this back edge closes a cycle on which (u, v) must lie. So we traverse all DFS-tree edges, with parent v and child u and check whether $u.\ell > v.dist$, and only for such edge we mark them as bridge edges. All other edges are non-bridge edges.

Clearly, once we compute $u.\ell$ for all nodes on the tree, the set of non-bridge edges can be found in $O(n)$. Thus, the runtime is dominated by the DFS runtime — $O(n+m)$.

(h) (5 pts) Give a $O(n+m)$ algorithm that finds the biconnected components of G. I.e., your algorithm gives each **edge** a label such that $e.label = e'.label$ iff e and e' belong to the same biconnected component of G. Explain why your algorithm is correct.

Answer. Fix any two vertices u and v. We argue that if u and v are in the same biconnected component, than between u and v there must be a path where all the edges are non-bridges. To see that, note that there's a cycle on which both u and v lie. So that gives a path $u \to v$ where all edges lie on a simple cycle and thus cannot be bridges. Moreover, if u and v are in different biconnected component, then any path

$u \to v$ must go through at least one edge which is a bridge. To see that, let P be a path between u and v. Since u and v belong to two different biconnected component, there has to be an edge $e \in P$ such that both of its endpoints belong to two different biconnected components. This edge $e = (x, y)$ has to be a bridge, because if it is not a bridge than x remains connected to y even without e, so x and y are connected by a cycle and so they belong to the same biconnected component.

This suggests an algorithm for finding the biconnected components: do BFS / DFS only use just non-bridges (i.e., before looking at a neighbor of a node, check that the edge connecting them isn't a bridge). Once we do so, the different trees in the BFS/DFS forest must be the vertices in the different biconnected components. Label each vertex with the root of the tree on which is lies G.

However, this does not completely find all the bi-connected components of G. See, for example, components 2 and 3 in the Figure. So, once we find the different components in the BFS forest (after removing the bridges), we need to make sure there are no articulation points *even withouth bridges* — since each articulation point breaks the component into several components. So now, by removing an articulation point from the BFS forest, we potentially break some component into several new components – so we make multiple copies of this articulation point, adding one copy to each component it touches.

Now we label the edges by the label of its endpoints — if both reside in the same component. Otherwise, both endpoints lie in different components and so the label of the edge e is the e itself — as no other edge e' lie on the same cycle as e.