CS 473: Fundamental Algorithms, Spring 2014
HW 1 (due Tuesday, at noon, February 4, 2014)

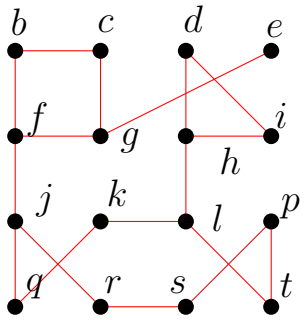
This homework contains three problems. **Read the instructions for submitting homework on the course webpage**.

Collaboration Policy: For this homework, Problems 1–3 can be worked in groups of up to three students.

Each student individually have to also do **quiz 1** online.

---

1. (30 PTS.) Cut vertices



Given a connected *undirected* graph $G = (V, E)$, vertex u is called a ***separating vertex***, or ***cut-vertex***, if removing u leaves the graph into two or more disconnected pieces; note that u does not count as one of the pieces in this definition. Your goal in this problem is to develop a linear time algorithm to find *all* the cut-vertices of a given graph using **DFS**. Let T be a **DFS** tree of G (note that it is rooted at the first node from which **DFS** is called). For a node v we will use the notation T_v to denote the sub-tree of T hanging at v (T_v includes v).

(A) (3 PTS.) In the graph shown above, identify all the cut-vertices.

(B) (7 PTS.) For each node u define:

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

Give a linear time algorithm that computes the low value for all nodes by adapting **DFS**(G). Give the altered pseudo-code of **DFS**(G) to do this. There is no need to prove that your code is correct.

(C) (5 PTS.) Prove that the root of the DFS tree is a cut-vertex if and only if it has two or more children.

(D) (5 PTS.) Prove that a non-root vertex u of the DFS tree T is a cut-vertex if and only if it has a child v such that no node in T_v has a backedge to a *proper* ancestor of u (that is, an ancestor of u which is not u itself).

(E) (10 PTS.) The above two properties can be used to find all the cut-vertices in linear time. Give the pseudo-code for a linear time algorithm to do so. There is no need to prove that your code is correct.

(It is instructive to run **DFS**(G) on the example graph and compute the pre values and the lowvalues for each node.)

2. (35 PTS.) Strange towns.

Let $G = (V, E)$ be a directed graph that represents the road network of a town. We say G is strange if there is a pair of nodes $u, v \in V$ such that neither u can reach v nor v can reach u. Describe a linear time algorithm to determine if G is strange. By linear time we mean an algorithm that runs in time $O(m + n)$, where $m = |E|$ and $n = |V|$. *Hint:* First solve the problem when G is a DAG. It is instructive to understand examples of strange directed graphs

and normal(?) ones.

3. (35 PTS.) Risky Walks.

In hardware and software verification one considers *infinite* walks over (finite) directed graphs where the nodes represent states and edges represent transitions from one state to another. Certain properties about the system can be expressed as properties about infinite walks and sometimes these can be checked efficiently. Here is a problem in this vein. Let $G = (V, E)$ be a directed graph with n vertices and m edges. A finite walk in G is a sequence of nodes $v_0, v_1, \ldots, v_h$ for some integer h where for each $0 \leq i < h$, (v_i, v_{i+1}) is an edge of G (note that nodes can repeat). An infinite walk in G, as the name suggests, is simply an infinite sequence $v_0, v_1, \ldots,$ where for each $i \geq 0$, $v_i \in V$ and $(v_i, v_{i+1}) \in E$. An infinite walk P in G is said to visit a node v infinitely often if v occurs infinitely often in P. Notice that since V is finite, every infinite walk P in G contains at least one node that occurs infinitely often.

Let $s \in V$ be a start vertex and let $A \subset V$ be a set of *dangerous* vertices and $B \subset V$ be a set of *safe* vertices; we will assume that $A \cap B = \emptyset$. An infinite walk P starting at s is said to be risky if contains some node from A infinitely often but no node from B infinitely often. Describe an efficient algorithm that given $G = (V, E)$, $s \in V$, $A \subset V$ and $B \subset V$ (such that $A \cap B = \emptyset$), decides whether there is a risky infinite walk P in G that starts at s. You may want to think about the following easier problems first; these are not to be submitted.

- If P is an infinite walk show that the set of nodes that occur infinitely often in P is a subset of a strongly connected components of G.
- Describe an algorithm that determines if G has an infinite walk starting at s.
- Describe an algorithm that determines if G has an infinite walk that vists some specific vertex v infinitely often.