"CS 374" Spring 2015 — Homework 4

Due Tuesday, March 3, 2015 at 10am

••• Some important course policies •••

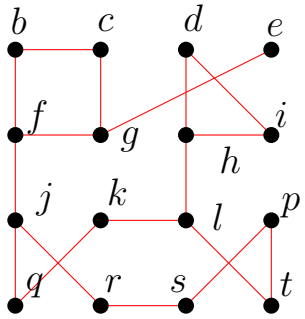
- **You may work in groups of up to three people.** However, each problem should be submitted by exactly one person, and the beginning of the homework should clearly state the names and NetIDs of each person contributing.
- **You may use any source at your disposal**—paper, electronic, or human—but you ***must*** cite ***every*** source that you use. See the academic integrity policies on the course web site for more details.
- **Submit your pdf solutions in Moodle.** See instructions on the course website and submit a separate pdf for each problem. Ideally, your solutions should be typeset in LaTex. If you hand write your homework make sure that the pdf scan is easy to read. Illegible scans will receive no points.
- **Avoid the Three Deadly Sins!** There are a few dangerous writing (and thinking) habits that will trigger an automatic zero on any homework or exam problem. Yes, we are completely serious.
  - Give complete solutions, not just examples.
  - Declare all your variables.
  - Never use weak induction.
- Unlike previous editions of this and other theory courses we are not using the "I don't know" policy.

See the course web site for more information.

If you have any questions about these policies,
please don't hesitate to ask in class, in office hours, or on Piazza.

---

1. 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) In the graph shown above, identify all the cut-vertices.

   (b) For each node u define:

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

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

   (d) 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) 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 on the example graph and compute the pre values and the low values for each node.)

2. 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 it 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 component 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.

3. *Racetrack* (also known as *Graph Racers* and *Vector Rally*) is a two-player paper-and-pencil racing game that Jeff Erickson apparently played on the bus in 5th grade.[1] The game is played with a track drawn on a sheet of graph paper. The players alternately choose a sequence of grid points that represent the motion of a car around the track, subject to certain constraints explained below.

   Each car has a *position* and a *velocity*, both with integer x- and y-coordinates. A subset of grid squares is marked as the *starting area*, and another subset is marked as the *finishing area*. The initial position of each car is chosen by the player somewhere in the starting area; the initial velocity of each car is always $(0, 0)$. At each step, the player optionally increments or decrements either or both coordinates of the car's velocity; in other words, each component of the velocity can change by at most 1 in a single step. The car's new position is then determined by adding the new velocity to the car's previous position. The new position must be inside the track; otherwise, the car crashes and that player loses the race. The race ends when the first car reaches a position inside the finishing area.

   Suppose the racetrack is represented by an $n \times n$ array of bits, where each 0 bit represents a grid point inside the track, each 1 bit represents a grid point outside the track, the 'starting area' is the first column, and the 'finishing area' is the last column.

   Describe and analyze an algorithm to find the minimum number of steps required to move a car from the starting line to the finish line of a given racetrack. No proof of correctness required.

   A 16-step Racetrack run, on a 25×25 track. This is *not* the shortest run on this track.

[1] The actual game is a bit more complicated than the version described here. See http://harmmade.com/vectorracer/ for an excellent online version.

velocity	position
$(0,0)$	$(1,5)$
$(1,0)$	$(2,5)$
$(2,-1)$	$(4,4)$
$(3,0)$	$(7,4)$
$(2,1)$	$(9,5)$
$(1,2)$	$(10,7)$
$(0,3)$	$(10,10)$
$(-1,4)$	$(9,14)$
$(0,3)$	$(9,17)$
$(1,2)$	$(10,19)$
$(2,2)$	$(12,21)$
$(2,1)$	$(14,22)$
$(2,0)$	$(16,22)$
$(1,-1)$	$(17,21)$
$(2,-1)$	$(19,20)$
$(3,0)$	$(22,20)$
$(3,1)$	$(25,21)$