CS 170 Algorithms
Spring 2015 Christos Papadimitriou and Prasad Raghavendra **HW 4**

1. (10 pts.) Depth-First Search Warm-up

(a) Ordering: ABGDEFCHI

nodes	pre-visit	post-visit
A	1	12
B	2	11
G	3	10
D	4	7
E	5	6
F	8	9
C	13	18
H	14	15
I	16	17

Tree: AB, BG, GD, DE, GF, CH, CI

Back: ED, DB

Forward: AE

Cross: IH

(b) $\{B,G,D,E\},\{A\},\{C\},\{F\},\{H\},\{I\}$ are the strongly connected components.

2. (20 pts.) Minimal Graphs

Solution:

(a) *Main Idea:* Let k be the number of connected components of G. Then, the answer is $n-k$.

Pseudocode:

Algorithm MinimalGraphUndirected(G):
1. Find connected components of G (Using a DFS).
2. Return n - number of connected components found.

Correctness: Each edge at the very best would merge two different connected components. Thus, we start with n components initially (isolated vertices), and when we add edges in one by one, we will decrease the number of components by at most 1 at each step. At the end, we are left with k components, thus we need at least $n-k$ edges. We can check that this is indeed achievable. Doing a DFS from one particular node will fine all edges in that component. G' can just include all these edges in the DFS tree. A tree on c vertices has $c-1$ edges. Thus, let c_i be the number of nodes in component i. Then, the total number of edges is $\sum -i = 1^k(c_i-1) = (\sum_{i=1}^k c_i) - k = n-k$.

Running time: A DFS takes linear time. The return value takes constant time to evaluate. Thus, this algorithm runs in linear time.

(b) *Main Idea:* Let's define the notion of a "weakly connected component" (abbreviated as wcc) as the connected components in the graph when we think of the edges as undirected. Let a be the number of wcc's in G, and let b the number of wcc's in G that have a cycle. Then, the answer is $n-a+b$.

Pseudocode:

Algorithm MinimialGraphDirected(G):
1. Let G' be undirected version of G (i.e. add all edges from G, but also add reverse edges into G').
2. Find the number of connected components in G' (using a DFS). Record this number as a
3. For each connected component in G', run a DFS in G to check if there's a cycle (there exists a back edge). Record the number of connected components with a cycle number as b.
4. Return $n-a+b$

Correctness: Let's suppose we only had a single weakly connected component. Suppose that this wcc had c nodes. There are two cases

- Suppose that this wcc did not have a cycle. Then, we can topologically sort the vertices, and connect them in that order with $c-1$ edges and satisfy all the conditions. We definitely need at least this many edges (since if we had fewer, the resulting graph wouldn't be weakly connected).
- Suppose that this wcc had at least one cycle. Here, we can't do a topological sort, $c-1$ edges (since if we could, we would have a tree, which can be topologically sorted). However, we can definitely do c edges, since we can connect all the vertices in a cycle.

Now, we've solved this problem for each wcc. Now, it's clear that we don't need any edges between any two different weakly connected component. Thus, we've solved the problem overall on any general graph. The quantity $a-b$ is precisely the number of wcc's with no cycles, thus we have shown that $n-(a-b)=n-a+b$ is indeed the minimum number of edges required.

Note: A common idea is to decompose the input graph into a DAG on SCCs, then to put a cycle on each SCC and then take a spanning forest from the DAG structure. This algorithms uses too many edges.

3. (10 pts.) DAG Revisited

Linearize the DAG. The answer to the question is YES iff the DAG has an edge $(i,i+1)$ for every pair of consecutive vertices labeled i and $i+1$ in the linearized order.

Reason: The edges in the linearized order can only go in the increasing direction in the linearized order, and the required path must touch all of the vertices.

Complexity: Both linearization and checking outgoing edges from every vertex takes linear time.

4. (20 pts.) Unique Shortest Path

This can be done by slightly modifying Dijkstra's algorithm. The array `usp[·]` is initialized to `true` in the initialization loop. The main loop is modified as follows:

while H is not empty **do**
  u = deletemin(H)
  for all $(u,v) \in E$ **do**
    if dist(v) > dist(u) + $l(u,v)$ **then**
      dist(v) = dist(u) + $l(u,v)$
      usp[v] = usp[u]
      decreasekey(H, v)
    end if
    if dist(v) = dist(u) + $l(u,v)$ **then**
      usp[v] = `false`
    end if

end for
end while

This will run in the required time when the heap is implemented as a binary heap.

Correctness: By Djikstra's proof of correctness, this algorithm will identify the shortest paths from the source u to the other vertices. For uniqueness, we consider some vertex v. If there are multiple shortest paths, then they either all share the same final edge (w,v) (for some vertex w), or they have different final edges. In the former case, there must be multiple shortest paths from u to w. This will be detected in the first conditional statement when the algorithm explores from w and updates the distance to v by taking edge (w,v). In the latter case, the algorithm will detect the existence of multiple shortest paths with the second conditional statement by testing for equality of two distances.

5. (30 pts.) Biconnected Components

(a) Let B and C be two distinct biconnected components defined by $B = \{e' | e_1 \sim e'\}$ and $C = \{e' | e_2 \sim e'\}$, where e_1 and e_2 are distinct edges. Suppose B and C share edge f in common. Then $f \sim e_1$ and $f \sim e_2$. But this implies $e_1 \sim e_2$.

To see why, note that is clearly true if $f = e_1$ or $f = e_2$. Otherwise, there is a simple cycle containing f and e_1 and a simple cycle containing f and e_2. The union of these two simple cycles contains a simple cycle that has both e_1 and e_2 as edges. (Note: For the purposes of grading, we will not require more detail than this)

By a similar argument, if $e' \sim e_1$ then $e' \sim e_2$ because $e_1 \sim e_2$. Likewise, if $e' \sim e_2$ then $e' \sim e_1$. Thus, $B = C$, contradicting the fact that they are distinct sets.

(b) We argue that two biconnected components must share at most one vertex. For the sake of contradiction, assume that two components C_1 and C_2 share two vertices u and v. Note that there must be a path from u to v in both C_1 and C_2, since in each component, there is a simple cycle containing one edge incident on u and one edge incident on v. The union of these two paths gives a cycle containing some edges from C_1 and some from C_2. However, this is contradiction as this would imply that an edge in C_1 is related to an edge in C_2.

(Note: The name *separating vertex* hints that if we delete such a vertex, then we will disconnected the biconnected components it is common to. Indeed, let u be a shared vertex. Let (u,v_1) and (u,v_2) be the edges corresponding to u in the two components. Then we claim that removing u disconnects v_1 and v_2. If not, then there must be a path between v_1 and v_2, which does not pass through u. However, this path, together with (u,v_1) and (u,v_2), gives a simple cycle containing one edge from each component which is a contradiction.)

(c) Let v be the root of the tree. If u is the only child of v in the tree and (v,w) is some edge in the graph then $(v,u) \sim (v,w)$ because either $u = w$ or (v,u), P, (w,v) forms a simple cycle, where P is the path from u to w in the DFS tree. Thus, v is not contained in more than one biconnected component. If v has two children u_1 and u_2, then if $(v,u_1) \sim (v,u_2)$ there would be a simple path from u_1 to u_2 which cannot happen because there cannot be cross edges in an undirected graph. Thus, v is part of two distinct biconnected components.

We can also cite the disconnecting property of separating vertices. If the root has only one child, then it is effectively a leaf and removing the root still leaves the tree connected. The DFS from the first child explores every vertex reachable through a path not passing through the root. Also, since the graph is undirected, there can be no edges from the subtree of the first child to that of any other child. Hence, removing the root disconnects the tree if it has more than one children.

(d) Let v be a non-root vertex with predecessor u. Suppose v' is a child of v that does not have any descendants with backedges to ancestors of v. If $(u,v) \sim (v,v')$, then we have a simple cycle with

ancestors of v, followed by v, followed by descendants of v'. Since this cycle closes, we would have a descendant of v' connected to an ancestor of v.

If no such v' existed, then any child v' of v would have some descendant w that had a backedge to an ancestor of v. We could take (v,v'), followed by a path from v' to w, followed by the backedge, and then back to (u,v) to get a simple cycle showing $(u,v) \sim (v,v')$. Thus, v is only part of one biconnected component.

Again, another way of looking at this is with the disconnecting view of separating vertices. If every child v' has some descendant with a backedge to an ancestor of v, each child can reach the entire tree above v so the graph is still connected after removing v. If there is a child v' such that none of its descendants have a backedge to an ancestor of v, then in the graph after removing v, there is no path between an ancestor of v and v' (note that there cannot be any cross edges since the graph is undirected).

(e) We first show how to compute the entire array of `low` values in linear time. While exploring each vertex u, we look at all the edges of the form (u,v) and can store at u, the lowest $\texttt{pre}(v)$ value for all neighbors of u. Then, $\texttt{low}(u)$ is given by the minimum of this value, $\texttt{pre}(u)$, and the `low` values of all the children of u. Since each child can pass its `low` value to the parent when its popped off the stack, the entire array can be computed in a single pass of DFS.

A non-root node u is a separating vertex iff $\texttt{pre}(u) \leq \texttt{low}(v)$ for any child v of u. This can be checked while computing the array. Also, if u is a separating vertex and v is a child such that $\texttt{pre}(u) \leq \texttt{low}(v)$, then the entire subtree with v as the root must be in different biconnected components than the ancestors or other children of u. However, this subtree itself may have many biconnected components as it might have other separating vertices.

Hence, we perform a DFS pushing all the edges we see on a stack. Also, when we explore a child v of a separating vertex u such that the above condition is met, we push an extra mark on the stack (to mark the subtree rooted at v). When DFS returns to v, i.e. when v is popped off the stack (of vertices), we can also pop the subtree of v from the stack of edges (pop everything till the mark). If the subtree had multiple biconnected components, they would be already popped off before the DFS returned to v.

6. (1 pts.) Extra Credit: Road Network Design

Construct a directed graph $\vec{G} = (\vec{V}, \vec{E})$ from the undirected graph $G = (V,E)$ and the potential edges E'. For every vertex $v \in V$, create $k+1$ copies $v_0, \cdots v_k \in \vec{V}$ and edges (v_i, v_{i+1}) (having 0 weight) for $0 \leq i \leq k-1$. For every (undirected) edge $(u,v) \in E$, create corresponding (directed) edges (u_0, v_0), $(v_0, u_0), \cdots, (u_k, v_k), (v_k, u_k) \in \vec{E}$ having same weight.

Finally, for every (undirected) potential edge $(u,v) \in E'$, create (directed) edges $(u_i, v_{i+1}), (v_i, u_{i+1}) \in \vec{E}$ having same weight to point from the i-copy to the $(i+1)$-copy for $0 \leq i \leq k-1$. Now the shortest path from s_0 to t_1 in $\vec{G}$ corresponds to the shortest path from s to t in G, using a potential edge at most k times.

Now Dijkstra's algorithm solves the problem in $O(k(|V| + |E| + |E'|) \log k|V|)$ time.

Rubric:

For any part of a problem with 10 points or more, use the following guideline for partial credit:

- All correct: 100%
- A few minor errors: 80%
- On right track, but many/major errors: 50%
- Had some ideas, but wasnt on right track: 20%
- No answer: 0%

For problems with a stated runtime requirement, any solutions not meeting the requirement are not eligible for points.

For rubric items for giving a correct algorithm with proof and runtime justification (i.e. the three parts are not split into separate items), award yourself credit according to the guideline above, while keeping in mind that the algorithm is worth roughly the same as the proof and runtime justification together.

Redemption:

You can redeem points on problems 4 and 5.

Problem 1

- 2 pts for ordering (all or nothing, no partial credit)
- 7 pts: deduct 1 point for each incorrect pre-visit, post-visit, and edge label (no negative score, of course)
- 1 pt for strongly connected components (all or nothing)

Problem 2

(a)
- 2 pts for stating the correct answer $n-k$, where k is the number of connected components (all or nothing)
- 6 pts for proving that $n-k$ is the correct answer
- 2 pts for invoking DFS or BFS to argue that the running time is linear
- 0 pts if the algorithm can ever output a number $\geq n$

(b)
- 2 pts for stating the correct answer is $n-a+b$, where a is the number of wcc's in G, and let b is number of wcc's in G that have a directed cycle.
- 6 pts for proving that $n-a+b$ is the correct answer
- 2 pts for invoking DFS to argue that the running time is linear
- 0 pts if the algorithm can ever output a number strictly larger than n

Problem 3

- 10 pts for invoking topological sort/linearization, checking if the sorted vertices form a path and correct complexity analysis

Problem 4

- 10 pts for setting usp[v] = usp[u] when dist(v) $>$ dist(u) + $l(u,v)$ and setting usp[v] = `false` when when dist(v) $=$ dist(u) + $l(u,v)$
- 7 pts for proving algorithm correctness
- 3 pts for time analysis

Problem 5

- (a), (b), (c), (d) worth 4 pts each, (e) worth 14 pts
- (a), (b) 4 pts each for a correct proof
- (c), (d) each question gets 2 pts for each direction of the "if and only if"
- (e) 3 pts for computing `low`[.] in linear time
- (e) 4 pts for identifying separating vertices and biconnected components in linear time
- (e) 4 pts for proving algorithm correctness
- (e) 3 pts for time analysis

Problem 6

1 pt for any correct algorithm not slower than $O(k(|V|+|E|+|E'|)\log k|V|)$.