

CS577 Assignment 2

Lei Kang, Liang Wang, Yan Zhai

March 11, 2013

PROBLEM 1

- 1.1 First we say L_i is the i th layer when perform BFS from a node v in a graph G (as the definition in book P79) $L_0 = \{v\}$.

We modify the original BFS. In each vertex x , we add an additional field C to count the number of shortest paths from the v to that vertex x . So initially for start node v , we set $C(v) = 1$ (itself), for other vertices x we set $C(x) = 0$.

Then we use BFS. During BFS process, for a node x in L_j , we set $C(x)$ to be the sum of the *paths* of its neighbor nodes in L_{j-1} . So

$$C(x) = \sum_{y \in \text{Neighbors}(x) \text{ AND } \text{layer}(y) = \text{layer}(x) - 1} C(y)$$

All the shortest paths from v to a node w (layer j) must have length i , and each of these shortest path is like $\langle v, x_1, x_2, \dots, x_i, \dots, x_j = w \rangle$, where x_i is the node at L_i in the BFS tree and $1 \leq i \leq j$. The only changed to the original BFS is the counter $C(x)$ for each vertex x . Obviously, the initialization takes $O(n)$ time and updating counter at any vertex x takes $O(|\text{Neighbors}(x)|)$ time. Since

$$\sum_{x \in V(G)} |\text{Neighbors}(x)| = 2E = 2m$$

and the original BFS takes $O(m + n)$, therefore the total time of this algorithm is:
 $O(m + n) + O(m) + O(n) = O(m + n)$

PROBLEM 2

- 2.(a) Let s be the root of G_π . Then consider the following 2 situations:

(1) If s has at most one child:

if s has no child, G must have exactly one node, so in this case, s is not an articulation point.

if s has one child x , if s is removed, only the edge (s, x) and the back edges to s will be removed. so that all other nodes will still be connected. So s is not an articulation point.

(2) If s has at least two children: since there is no any cross edge for each child tree in G DFS trees, then deleting s will make each child tree disconnected. Thus if s is an articulation point of G then it at least has two children.

2.(b) We prove it from two directions:

(1) If v has a child s such that there is no back edge from s or from any descendant of s to a proper descendant of v , then v is an articulation point.

Prove: Consider the subtree of the DFS tree rooted at s . Consider edge (x,y) , which x is in this subtree. Then y must either be v or within the subtree. Because if (x,y) is a tree edge, y must be in the subtree; otherwise, (x,y) is a back edge, y could not be connected to an ancestor of v in given condition, so that it must link to v or a node in the subtree. In this case, if we remove v , x and the parent of v must be disconnected. So v must be an articulation point.

(2) If for each child s of v , there is some back edge from s or from some descendant of s to a proper descendant of v , v is not an articulation point.

Prove: let p be the parent of v in the DFS tree. We assume v has k children in the DFS tree. Then consider delete v from the graph. For the DFS tree, it will be partitioned into exactly $(k+1)$ connected components, where the vertex p and v 's child vertices be indistinct parts. However, from our condition, each child vertex of v must be connected to p in G . Thus, the graph G after removal of v is still connected, so that v is not an articulation point.

2.(c) We can compute $v.\text{low}$ for all vertices v by starting at the leaves of the tree G_π . We compute $v.\text{low}$ as follows: $v.\text{low} = \min(v.d, \min y.\text{low}, \min w.d \mid y \text{ children}(v) \text{ backedge}(v,w))$. For leaves v , there are no descendants u of v , so this returns either $v.d$ or $w.d$ if there is a back edge (v, w) . For vertices v in the tree, if $v.\text{low} = w.d$, then either there is a back edge (v, w) , or there is a back edge (u, w) for some descendant u . The last term in the min expression handles the case where (v, w) is a back edge. If u is a descendant of v in G_π , we know that $u.d > v.d$ since u is visited after v in the depth first search. Therefore, if $w.d < v.d$, we also have $w.d < u.d$, so we will have set $u.\text{low} = w.d$. The middle term in the min expression therefore handles the case where (u, w) is a back edge for some descendant u . Since we start at the leaves of the tree and work our way up, we will have computed everything we need when computing $v.\text{low}$. For each node v , we look at $v.d$ and something related to all the edges leading from v , either tree edges leading to the children or back edges. So, the total running time is linear in the number of edges in G , $O(E)$.

2.(d) For root, we only need to check if it has more than 1 child based on (b). If root has more than 1 child, it is a articulation point $O(1)$ time.

Then for other vertices use the same algorithm in (c). If $s.\text{low} \geq v.d$, which means that subtree rooted at s has no back edge to the ancestors of v in G , then delete v would disconnect G , because based on (b) any nonroot vertex v is an articulation point if and only if it has a child s in G_π with no back edge to a proper ancestor of v . The algorithm in (c) take $O(E)$ time. So the total time is $O(E)$.

2.(e) We prove it from two directions:

(1) if an edge (u, v) is a bridge then it can not lie on a simple cycle.

Prove: Assuming (u,v) is a bridge. After we remove (u,v) , G will be disconnected, so then there is no path from u to v . However, if (u,v) is on a simple cycle, then based on the property of cycle, there will be a path like $u.x_1.x_2 \dots x_n.v.u$, remove edge (u,v) will not affect the other edges in this path, so there still is a path from u to v , which means (u,v) is not a bridge. This is against our assumption. So (u,v) is not on a simple cycle.

(2) if an edge (u, v) is not on a simple cycle, then it is a bridge. Prove: Because if edge (u,v) is not on a simple cycle, there will be only one path from u to v (If there are two paths $u.x_1.x_2 \dots x_n.v$ and $u.v$, then $u.x_1 \dots x_n.v.u$ is a cycle) Then delete (u,v) would disconnect u,v

2.(f) xxxxxxxxxxxxxxxxxxx Use the algorithms and conclusions in (c) (d)

Any bridge in the graph G must exist in the graph G . Otherwise, assume that (u, v) is a bridge and that we explore u first. Since removing (u, v) disconnects G , the only way to explore v is through the edge (u, v) . So, we only need to consider the edges in G as bridges. If there are no simple cycles in the graph that contain the edge (u, v) and we explore u first, then we know that there are no back edges between v and anything else. Also, we know that anything in the subtree

of v can only have back edges to other nodes in the subtree of v . Therefore, we will have $v.\text{low} = v.d$ since v is the first node visited in the subtree rooted at v . Thus, we can look over all the edges of G and see whether $v.\text{low} = v.d$. If so, then we will output that $(\text{parent}[v]G, v)$ is a bridge, i.e. that v and its parent in G form a bridge. Computing $v.\text{low}$ for all vertices v takes time $O(E)$ as we showed in part (c). Looping over all the edges takes time $O(V)$ since there are $|V| - 1$ edges in G . Thus the total time to calculate the bridges in G is $O(E)$.

2.(g) To prove this statement, we need to show:

a) Any non-bridge edge belongs to a biconnected component.

According to part(e), we know that such edge should appear on at least one simple cycle. Thus it means it must be in some biconnected component.

b) A non-bridge edge can only belong to one biconnected component.

If not then we assume a non-bridge edge $e = (u, v)$ belongs to two biconnected components G_1 and G_2 . From the definition we just need to show that: for any $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ in $G_1 \cup G_2$, there is a simple cycle containing e_1 and e_2 . If e_1 and e_2 both belong to G_1 or G_2 , then it is obvious. Not losing generality we assume $e_1 \in G_1/G_2$ and $e_2 \in G_2/G_1$, and we show that there is a simple cycle C containing e_1 and e_2 .

For convenience we use $a - b$ to represent there is a simple path between vertex a and b . The cycle construction could be following:

- according to definition of e , there would be a cycle $C_1 = (v_1, u_1) - p_1 - (u, v) - q_1 - (v_1, u_1)$ containing e and e_1 . Similarly, we have another cycle $C_2 = (v_2, u_2) - p_2 - (u, v) - q_2 - (v_2, u_2)$. Here p_1, q_1, p_2, q_2 are simple paths.
- starting from u_1 along the path $p_1 - u$, until we meet the first vertex x on p_2 or q_2 . The existence of such vertex is guaranteed by u . Similarly, we could find y by travelling along $q_1 - v$.
- if x and y are on the same path, for example, p_2 , then we can paste $x - (u_1, v_1) - y$ into cycle C_2 , replacing the path of $x - y$, and it's easy to see this new cycle is simple
- if x and y are on the different paths, let x on p_2 and y on q_2 , then paste $x - (u_1, v_1) - y$ into C_2 by replacing the path $x - (u, v) - y$, producing a simple cycle containing (u_1, v_1) and (u_2, v_2)

So we could always produce a simple cycle containing e_1 and e_2 , thus any non-bridge edge could only appear in one biconnected component.

2.(h) We could DFS once to compute the biconnected components. But an easier way is to DFS twice: the first time we find all the bridges. Then we remove all the bridges, and do DFS on the generated forest. Each connected sub-graph is then a biconnected component. The correctness is supported by part(2.g), since biconnected components are a partition of the non-bridge graph. The total cost of doing this is:

- cost of find all bridges, $O(E)$
- cost of removing bridges, doing on the original graph requires $O(E)$
- cost of DFS on the generated forest, requiring $O(E)$
- cost of labeling bcc of the bridges, requiring $O(E)$

so the total complexity is $O(E)$

PROBLEM 3

3.1 I think the running time should be $O(L \log L + Ln)$

We reduce the problem to the shortest-path problem:

First we construct a graph with L vertices, labeled $v_0, v_1 \dots v_{L-1}$. For a vertex v_x , we perform the following process: For each $l_i (i = 1, 2 \dots n)$, calculate $(x + l_i) \bmod L$, say y then we add an edge from v_x to v_y , and $w(v_x, v_y)$ is the value l_i . So for two nodes v_x and v_y , if $v_x - v_y \equiv l_i \pmod{L}$, then there is an edge between them.

Example: say $n = 2, l_1 = 2, l_2 = 3$ and $L = 7$. So we have 7 vertices: $v_0, v_1 \dots v_6$.

At vertex $v_5, 5 + l_1 = 5 + 2 \equiv 0 \pmod{7}$, so there is an edge from v_5 to v_0 with weight l_1 ,

$5 + l_2 = 5 + 3 \equiv 1 \pmod{7}$, so there is an edge from v_5 to v_1 with weight l_2

This problem is converted to find the shortest path from v_0 to v_0 .

Then, we add another node v_L , serve as the "mirror" vertex of v_0 , which means v_L connects to the same vertices as v_0 , so this problem is converted to find the shortest path from v_0 to v_L .

Then we use Dijkstra's algorithm. The graph has $L + 1$ vertices and at most $(L + 1)n$ edges; constructing it takes $O(Ln)$ time. The cost of running Dijkstra's algorithm on the graph is $O(L \log L + Ln)$, hence the total running time is also $O(L \log L + Ln)$.

PROBLEM 4

- 4.1 In the graph G , use BFS until find a cycle, then delete the heaviest edge on this cycle. For each 'delete' operation, we get a new Graph G_i and we say the original graph G is G_0 .
so G_i is still connected and has the same spanning tree with G_{i-1} , and but $E(G_i) = E(G_{i-1}) - 1 (i = 1, 2 \dots)$
(For any cycle C in the graph, if the weight of an edge e of C is larger than the weights of all other edges of C , then this edge cannot belong to an MST). Repeat same process 8 times (total 9 times), the G_0 become a connected graph G_9 . G_9 has at most $n-1$ edges and the same spanning tree as G_0 . In fact, G_9 is connected, has n vertices and $n-1$ edges, so G_9 is a tree. From previous process, we know G_9 is also the minimum spanning tree of $G_0 = G$.
BFS and find heaviest edge take $O(V+E) = O((n+8)+n) \approx O(n)$ time.