CS170–Spring 2015 — Solutions to Homework 4

Kevin Chau, SID 23816929, kevinchau321@berkeley.edu

Collaborators: Howard Chiao

1. Depth-First Search Warm-up

(a) The order of nodes visited in a DFS: $A, B, G, D, E, F, C, H, I$
Let the tuple (X_{pre}, X_{post}) represent the previsit and postvisit number of node X. After DFS, A through I have the following traversal numbers:
A: (1,12)
B: (2,11)
C: (13,18)
D: (4,7)
E: (5,6)
F: (8,9)
G: (3,10)
H: (14,15)
I: (16,17)
Now we classify the edges according to whether it is a Tree, Back, Forward, or Cross edge in the DFS:
Tree Edges: $\vec{AB}, \vec{BG}, \vec{GD}, \vec{DE}, \vec{GD}, \vec{CH}, \vec{CI}$ (in order of traversal)
Back Edges: $\vec{DB}, \vec{ED}$
Forward Edges: $\vec{AE}$
Cross Edges: $\vec{IH}$

(b) There are 6 strongly connected components in the graph: 1 group consisting of (E,D,B,G) and all 5 other nodes (A), (C), (F), (H), and (I) form their own singlet groups (which are considered disjointly connected to themselves) for a total of 6 strongly connected components.

2. Minimal Graphs

(a) (a) Main Idea: Given an undirected graph G = (V,E), create a new graph G' = (V,E') with the same vertices but a minimized number of edges such that if there is a path from u to v in G then there is also a path in G'.

(b) Pseudocode: Run a DFS on G starting at any point. If during the algorithm a backedge is discovered (that is, the algorithm sees a child that has already been visited), delete the backedge from E. Delete all such back edges from G and the final result is a minimum spanning tree G'.

(c) Proof of Correctness: First we note that a tree with N vertices needs N-1 edges to span all nodes in a line. If there is more than N-1 edges, there must be a cycle. A cycle in an undirected graph is detected everytime DFS finds an already visited node, aka a back edge, and thus this algorithm removes all extra cycles by running DFS. Another way to phrase how this DFS algorithm works is that G' only contains the DFS's tree edges, as in the edges going forward to unvisited nodes. Since this algorithm only allows tree edges in the final graph, there can be no loops.

(d) Running Time: $O(\|V\| + \|E\|)$

(e) Justification: This algorithm runs in the exact same time as Depth First Search. There is the modification that the algorithm may need to delete an edge from the graph, but this is a constant time algorithm and there are at most $O(\|E\| - (V-1)) = O(E)$removals. Adding this factor in, the run time is unchange and still linear in the number of edges and vertices.

(b) (a) Main Idea: Given an directed graph G = (V,E), create a new graph G' = (V,E') with the same vertices but a minimized number of edges such that if there is a path from u to v in G then there is also a path in G'.

(b) Pseudocode:

i. Make a graph G_R where all the edges in G are reversed.

ii. Run DFS on G_R.

iii. Run an undirected connected components algorithm on G (just anothe modified DFS algorithm), making sure to process the vertices in decreasing order of the post traversal numbers from DFS(G_R) (step 2). This is in summary a strongly connected components algorithm which gives you a list of all the strongly connected components in G, which together form a meta graph DAG called G_meta.

iv. In each strongly connected component, the number of vertices is equal to the minimum number of edges needed to make the component strongly connected. Thus we can add up all the number of vertices in all strongly connected components and call this sum thus far $NumEdges$.

v. Considering each strongly connected component as a node in an undirected graph, we can utilize the algorithm in part (a). On the DAG G_meta (replacing the components with single nodes) run a DFS and increment the NumEdges count everytime we forward traverse a tree edge, just like in part a. After this last DFS, return NumEdges.

(c) Proof of Correctness: The algorithm just breaks the graph into a DAG of strongly connected components. For each strongly connected component, the number of edges in G' needed to ensure that we do not destroy the connectivity of any two nodes is just the number of vertices in the component. This is because we can make a cycle of all the

vertices which would maintain strong connectivity, so if u and v in the strongly connected component have a path in G then they have a path in G'. After counting up all the vertices in all the strongly connnected components, we just need to add the number of edges it would take to connect these components. Thus we just run the same algorithm on the metagraph DAG as in part (a), which essentially ensures that the DAG is linearized with only (number of stronly connected components - 1) edges between them. Thus the algorithm is correct.

(d) Running Time: $O(\|V\| + \|E\|)$

(e) Justification: Other than modifying the structure of G once by doing a reversal (O(E) time) and removing edges in the same O(E) manner as part A, the algorithm just runs multiple DFS passes on the graph. Since this is just a constant number of passes not dependent on the input size, the algorithm runs in the same time as DFS: linear in the size of the input graph.

3. DAG Revisited

1. Main Idea: An algorithm that takes a directed acyclic graph, G = (V,E) as input. It returns true if G contains a directed path that touches every vertex exactly once; false otherwise. The algorithm runs in $O(\|V\| + \|E\|)$ time.

2. Pseudocode: Linearize the DAG. That is, run a DFS on G, and then sort the nodes in order of decreasing post traversal number so that we get a linearized list of V. For every pair of consecutive vertices v_i and v_{i+1} in the linearized list of nodes, check whether the DAG has edge (v_i, v_{i+1}) in E in either direction. If every pair of consecutive nodes in the lineariztion has an edge in the DAG then there is a path that touches every node only once and the algorithm returns True. If any pair of consecutive nodes form an edge that is not in G, then return False.

3. Proof of Correctness: We know that if the linearization is missing an edge between two consecutive nodes v_i and v_{i+1}, then we had to get to v_{i+1} from some node before v_i. Since we are looking for a path that runs through every node just once, having an edge to v_{i+1} from a previous node but not from v_i implies that v_{i+1} is not strongly connected to the previous nodes in the linearization and that v_i is a sink. Since edges can only go in increasing direction in the linearization, it is thus it impossible to create such a path through a graph missing an edge between two consecutive nodes in the linearization. So the algorithm detects correctly when a DAG does not have a path that touches every vertex once. If on the other hand the linearization has an edge between all consecutive nodes, we know the graph indeed has such a path since it is exactly the path given by the DFS linearization.

4. Running Time: $O(\|E\| + \|V\|)$

5. Justification: Linearization runs on Depth-First Search which is known to be $O(\|E\| + \|V\|)$. After linearizing the DAG, the algorithm checks all consecutive nodes in the linearization; the number of pairs of consecutive nodes is of course less than the total number of edges $\|E\|$, so this scan of the linearization takes O(E) time. Adding these two steps together we get a constant factor in front of the magnitude of E, so the running time is linear in $\|E\|$ and $\|V\|$.

4. Unique Shortest Path

1. Main Idea: Shortest paths are not always unique: sometimes there are two or more different paths with the minimum possible length. Show how to solve the following problem in O((|V|+ |E|)$log|V|$) time.
   Input: An undirected graph G = (V, E); edge lengths $l_e > 0$; starting vertex s ∈ V.
   Output: A Boolean array usp[]: for each node u, the entry usp[u] should be true if and only if there is a unique shortest path from s to u.(Note:usp[s] = true.)

2. Pseudocode: This algorithm modifies Dijkstra's algorithm. Initialize an array of booleans usp[] to True for all vertices. When looping over edges $(u, v) \in E$, add an else if clause to catch the case where dist(v) == dist(u) + l(u,v). If this condition is True, set the boolean usp[v] to False. After running this modified Dijkstra's algorithm, return usp[]. The pseudocode looks like:
   UniquePath(G,s):
   for all $u \in V$: dist(u) = ∞; usp[u] = True
   dist(s) = 0
   H = makequeue(V)
   while H is not empty:
   u = deletemin(H)
   forall edges $(u, v) \in E$:

   (a) if dist(v) > dist(u) + l(u,v):
   usp[v] = True dist(v) = dist(u) + l(u,v)
   decreasekey(H,v)

   (b) else if dist(v) == dist(u) + l(u,v):
   usp[v] = False

   return usp[]

3. Proof of correctness: Since all edge lengths are weighted greater than 0, Dijkstra's Algorithm is guaranteed to find the length of the shortest path from s to all reachable nodes. If at any point in running the algorithm finds a new path dist(u) + l(u,v) which has the same length as the already known path distance dist(v), we have potentially found two non-unique paths to v. Thus the algorithm sets the boolean to false. If the algorithm later finds a shorter path than these two equal paths, it sets the boolean back to true. Thus our algorithm is guaranteed to find whenever a path is not a unique shortest path.

4. Running Time: $O((\|V\| + \|E\|) \log(\|V\|))$

5. Justification: This algorithm adds the constant time operation of checking for equality within the main loop of Dijkstra's algorithm. Since the addition is O(1), the running time of the algorithm is the exact same as the running time of Dijksta's Algorithm, which we know runs in $O((\|V\| + \|E\|) \log(\|V\|))$.

5. Biconnected Components

(a) Show that two distinct biconnected components cannot have any edges in common: We have two completely different biconnected components C_1 and C_2, which we can define as $C_1 = \{e_1 \in E \mid e_1 \sim e'\}$ and $C_2 = \{e_2 \in E \mid e_2 \sim e''\}$. In order for the components to be distinct, we require $e' \neq e''$. Suppose for the sake of contradiction that C_1 and C_2 share an edge, namely the edge $e_3 \in C_1$ and $e_3 \in C_2$. Then $e_3 \sim e'$ and $e_3 \sim e''$. It's easy to show that the biconnectivity relation between edges is an equivalence relation, so transitively $e' \sim e''$. Hence we can rewrite the definition of C_1 as $C_1 = \{e_1 \in E \mid e_1 \sim e''\}$. But e_1 and e_2 are just dummy variable edges, so C1 and C2 now have the same definition. Thus they are not distinct and by contradiction e_3 is not a valid edge so C_1 and C_2 must not share any edges.

(b) Let the distinct biconnected components C_i have the set of edges E_i and the set of vertices V_i. Then the components C_i and C_j share the vertices in the intersection $V_i \cap V_j$. First we show that C_i and C_j cannot have more than 1 vertex in common. Suppose that we have two distinct vertices v and $w \in V_i \cap V_j$. Then there is a path from v to w in C_i and in C_j. The union of these two paths gives a cycle that contains edge in both C_i and C_j. This implies that there are edges in E_i and E_j such that $e_i \sim e_j$, which contradict that the components are distinct. Next we must show that if the components intersect, they intersect at exactly 1 point. Suppose the point where they intersect is at u. Then (u,v_i) and (u,v_j) are edges in C_i and C_j. If we remove u, then v_i and v_j are no longer connected, unless there is another path from v_i to v_j. But this contradicts our assumption that there is only 1 seperating vertex. Thus if two components intersect they only intersect at 1 point. If the components do not intersect, then it is clear that they have distinct vertices since they have distinct edges. Thus the vertex sets must either be disjoint or have 1 vertex in the intersection.

(c) First we show the forward direction: if the DFS is a seprating vertex, then it has more than 1 child. From part (b) we know that a seperating vertex is the only vertex in the intersection of two (or more) biconnected components. Thus by definition we already know that a DFS root that is a seperating vertex has 2 or more sub trees.
Next we show the reverse direction: if the DFS root has more than 1 child, it seperates biconnected components. The properties of DFS guarantee that there are not any cross edges between the childrens' subtrees since DFS finds all nodes reachable from the children (thus any node has to be in either one child's sub tree or the other's). Since there are no cross edges, each sub tree is a distinct biconnected component, hence proving that the root is a seperating vertex.

(d) First we prove the forward claim. If a non-root vertex v is a seperating vertex, then from part b we know its children are distinct biconnected components. If we have a backedge from v to one of its ancestors, at least one of the biconnected children cannot contain this back edge. Thus at least one of v's children has no descendants with a back edge to v. Now we prove the reverse direction. If v has a child v' none of whose descendants has a backedge to proper ancestor of v, then the edge between v and v' itself is a biconnected component. Thus v is a seperating vertex.

(e) (a) Main Idea: An algorithm that computes all seperating vertices and biconnected components contained in a graph G in linear time.

(b) Pseudocode: Perform a DFS with following modifications. Suppose the current vertex getting explored is u who has parent u'. For all backedges (u,v), when we previsit u set

a value low(u) = min(previsit(u), previsit(v)). If there are no backedges, low(u) is just the previsit value. During post visits, low(u') = min(low(u), low(u')) if u has at least one backedge. After we have determined all low(u) from running the first DFS scan, we again do an additional DFS. During previsits, for each edge (u,v) if pre(u) ¡ low(v) for all v, then return u as a seperating vertex. The biconnected components can just be read after dividing the DFS trees into sub trees at each seperating vertex.

(c) Proof of Correctness:

(d) Running Time: $O(\|E\| + \|V\|)$

(e) Justification: The algorithm is based on a constant number of DFS runs and the addition of constant time operations in previsitng and postvisiting. Just like other algorithms based on DFS, the running time is just the same as the running time of DFS, so the algorithm is linear.