Songhao Li April 8, 2021

6.006 Problem Set 6

Collaborators: *Problem 1: Yukai Tan; Problem 2: Yukai Tan*

Problem 1

(1)
parent: $[v_1, v_5, v_3, v_3, v_3, v_2, v_5, v_6, v_1]$
order: $[v_7, v_6, v_0, v_8, v_1, v_5, v_2, v_4, v_3]$
topological sort: $[v_3, v_4, v_2, v_5, v_1, v_8, v_0, v_6, v_7]$

(2)
Intervals:
$(1(3(2(5(6(7,7)6)5)2)(4,4)3)1)$
Tree edges: $(1,3), (3,2), (2,5), (5,6), (6,7), (3,4)$
Back edges: $(2,1), (7,2), (5,1), (6,3)$
Forwrd edge: $(5,7)$

Problem 2

(1) Algorithm: We can build two graphs: a graph for SO's and a graph for luminaries. If an edge exists from node A to node B, means the person A is jealous of the person B. Run topological sort on both graphs, and assign values from 1 to n to them such that if s_i is jealous of s_j, $s_i > s_j$ and same for luminaries.

And then for each pair (l_i, s_j), assign the value $n^2 - l_i * s_j$ to it and count sort all such pairs with the value in ascending order. If we successfully run both topological sort without circle detected, then we are guaranteed to have a schedule.

Correctness:

We prove the correctness by two steps:

1) If the algorithm can find a schedule, then the schedule is a feasible one that satisfies all three rules.

if $l_i \to l_j$ and $s_i \to s_j$, then by the topological sorts we have $l_i > l_j$ and $s_i > s_j$, and by count sort we are guaranteed to place (l_i, s_i) before (l_j, s_j)

If $l_i \to l_j$, then for every SO s_k, we must have $l_i * s_k > l_j * s_k$, and we are guaranteed to that (l_i, s) is scheduled before (l_j, s). The same goes for jelousies betweeen SO's.

2) If the algorithm can not find a schedule, then there does not exist a schedule in nature

The reason that the algorithm can not find a schedule is detecting a circle in either the SO graph or luminaries graph. Without loss of generality, assume there is a circle in SO, then considering a set of matches including the SO's in the circle and any random luminary l, by rule (ii) their exists a circle of sequential orders

Runtime:

Build the graph takes $O(n^2)$, and the topological sorts each takes $O(|V| + |E|) = O(n^2)$. Count sort takes $O(n^2)$ since the values are bounded by $O(n^2)$

(2)
An example where there is no cycle of jealousies but there does not exist a suitable schedule: Suppose $l_1 \to l_2$,$s_2 \to s_3$, and $l_3 \to l_1$. There is no cycle of jealousies. However, consider matches (l_1, s_1), (l_2, s_2), and (l_3, s_3), match 1 should be placed earlier than match 2, match 2 should be placed earlier than match 3, and match 3 should be placed earlier than match 1. There is no cycle but there does not exist a suitable schedule.

Problem 3

(1) Statement to be proved: If uv is a bridge, then removing uv will increase the number of connected components

Since u and v are originally connected at least by uv Denote A for the set of nodes connected to u or v, B for the set of nodes connected to neither u nor v

Claim 1: a connected component can not have nodes from more than one of {A,B}

The proof is trivial, since any two nodes from different sets are not connected

As a result, the number of connected components for the whole graph is essentially the sum of number of connected components in A,B

Claim 2: A only has one connected component

The proof is trivial as well since all nodes in A are connected to u or v, u and v are connected thus they are all connected.

It is needless to prove that removing uv will not increase the number of connected componens from B. Now lets consider A.

We prove the contrapositive statement of the original statement as: if there is at least one path from u to v that does not include uv, then uv is not a bridge.

Removing uv wont affect other nodes' connection with u and v in A. As a result, all nodes in A are connected to u or v. If there is at least one path from u to v that does not include uv, u and v are still connected after removing uv; as a result, all nodes in A are still connected and there is still only one connected component in A. Removing uv wont increase the number of connected components so uv is not a bridge.

(2) u and v are connected and edge uv is the only way for them to be connected. Without loss of generality, assume u is discovered earlier than v. So just when u is discovered, edge uv is undiscovered and node v is undiscovered. As a result, v is discovered through uv and uv is thus a tree edge

(3) if u.in $\geq$ v.low, then there is a backedge in v's subtree that connects a v's decendant x to u or u's ascendant y. As a result, there is a another path from v to u apart from uv through the following paths: 1) v's tree edges to x 2) x's backedge to u directly, or u's ascendant y 3) y's tree edges to u. This contradicts fact proved in (1). Thus u.in ¡ v.low

(4) Run a DFS that updates v.low

Algorithm 1 DFS(u)

```
u.in = in++
u.low = u.in
mark u as explored
for v in u.adjacent_list: do
  if if not explored: DFS(v) then
    u.low = min(u.low,v.low)
  end if
  if u.in  v.low then
    bridges.append(uv)
  end if
end for
```

Runtime: every node is called by DFS once; every edge is accessed by the inner loop once. O(n+m)

(5) Run DFS as (4), while also maintain a list in_order that records the discovery order of nodes. in_order[k] is the node discovered at discovery time k

Design a iterating loop:

while v.low ¡ v.in: v = v.low

This loop finds the earliest ancestor of v that can be found through back edges.

We firstly run the loop on t, say the loop ends at p. There are at least two paths from p to t, so Avert can always arrive at p and find at least two ways to t. Now the question degenerate into blocking path from s to p. Since there is no child of p1 that can go to earlier discovered node, block p and its parent can do the job: this is one possible street.

Recurse the algorithm on s to p.parent, and end until p == s or p.parent == s

Correctness: proven in the algorithm.

Runtime: the DFS takes O(n+m). Every iteration in the loop directs to an earlier discovered node, so the total runtime of iteration is bounded by O(n). The whole algorithm is bounded by O(n+m)

Problem 4

(1) This is a classical topological sort.

Initiate a status list to store the *status* of every node, 0 for undiscovered, 1 for unfinished and 2 for finished. Initiate an empty list *order* for installation.

DFS(v):

if status[v] == 1, end the outter functions and return *None* to the console.

elif status[v] == 2, return

else, status[v] = 1, DFS nodes in dependencies[v]

When finished, status[v] = 2, append v to the end of *order*

Call DFS for every library, and *order* is a correct possible order to install

(2) We can augment DFS to store the depth of the node, while the depth is reversely defined as the maximum distance from a leaf node.

The DFS(v) returns the order of v. By DFS-ing its dependent libraries, we can know their depth, and we define the depth of v as the maximum depth of its dependent libraries plus 1. Other operations remains unchanged including early stop (if the status is finished, return to the outer function its depth; if the status is discovered but unfinished, end the program and return None to the console)