CSCI-GA.1170-001 Problem 11-2,3

Ankit Sati

TOTAL POINTS

22 / 22

QUESTION 1

1 Cool Orderings **8 / 8**

✓ **- 0 pts** *Correct*

QUESTION 2

2 Path Minimum **14 / 14**

✓ **- 0 pts** *Correct*

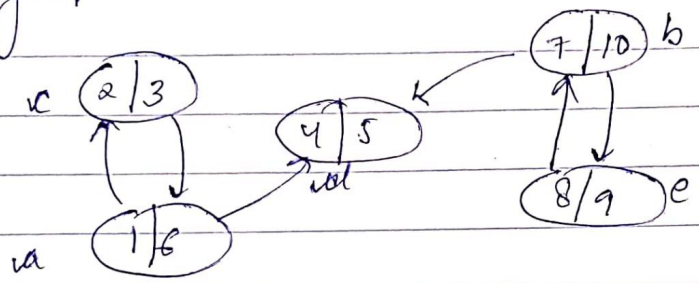
Problem 2

2(a) "Sorting the vertices in decreasing order of their finishing time is always cool".

→ This statement is incorrect.

→ We can show that with the help of the example given in the question.

given graph.



Order of visit- a, b

This DFS when sorted on decreasing finishing time, gives us the ordering of b, e, a, d, c

This ordering is not cool as

→ The k groups are not in topological order and

→ The vertices cannot be split into k groups corresponding to the SCC's of the graph

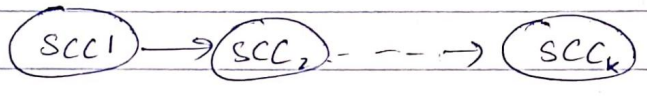
⇒ Not cool.

Hence, we have disproved the statement by showing a counter example.

2(b) "If DFS is run on G^T w.r.t some cool ordering, the resulting DFS trees are on the SCCs of G"

Proof

Since vertices are cool-ordered, they can be split up into k groups corresponding to the k SCC's of G such that they are in sequence and is a topological order.

i.e 

for G

↳ But for G^T, this ordering of vertices is same but the edges are flipped.

∴ we have, 

for G^T.

→ This is the structure of G^T with cool ordering.

Now if we run DFS on it, starting from vertices in SCC, it is clear to not that there is no edge leading out of SCC.

Hence when DFS visit enters SCC it will terminate when it has traversed all the vertices in it.

And the resulting tree is indeed on SSC of G.

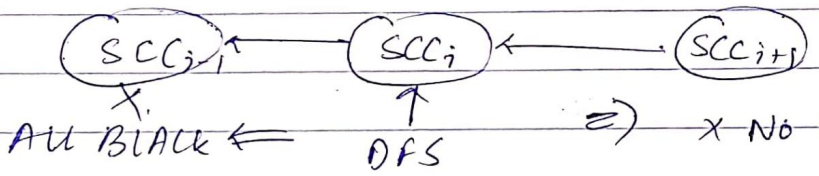
Now, when DFS reaches vertices of SCC_2, it cannot enter SCC, as all the vertices there are BLACK

and processed. There is no other leading edges out of SCC_2.

Hence it will also terminate after traversing all vertices of SCC_2 and resultly there will indeed be an SCC of G.

→ In this manner, we can see that all SCC of graph G will be returned when DFS is run of G^T.

Hence for each SCC it cannot go to previous SCC vertices as they are black and cannot go to further SCC vertices are not such edge exists.



Therefore it will return after traversing all vertices of SCC and DFS will move on the next element.

This will go on till end of the ordering.

Thus we have proved that resulting DFS trees from DFS runs in G^T with cool ordering are SCCs of G.

1 Cool Orderings 8 / 8

✓ **- 0 pts** *Correct*

3(a) G - directed tree with root s.

Note - Node u is in a part of the subtree.

Algo.

REGDFS-VISIT (G, u);

1. time = time + 1
2. u.d = time
3. u.color = GRAY.
4. u.min = u.val.
5. for each v ∈ G.Adj [u]
6. if v.color == WHITE:
7. v.IT = u.
8. REC DFS-VISIT (G, v)
9. u.min = min (u.min, v.min)
10. u.color = BLACK.
11. time = time + 1
12. u.f = time

DRIVER:

DFS (G);

1. for each u ∈ G.V.
2. u.color = WHITE
3. u.min = ∞
4. time = 0
5. REC-DFS-VISIT (G, s)

Correctness: Since G is a directed tree with root s, we only need to call DFS-VISIT on the root s. The recursive algorithm will take care of the root.

We modify the algorithm by simply adding two lines (4th & 9th). The 4th line initializes the min value of node u as u.val. This is because $u.min$ is the minimum value of subtree u and it includes Node u as well.

Then in the for loop, we call DFS-VISIT on all children of u where their min values are calculated.

In the 9th line we store the minimum of $u.min$ & $v.min$ into $u.min$.

In this way, we keep updating u.min after each of its children have been traversed.

Runtime $\Theta(m)$ where m = number of edges.

As each edge is visited ~~at~~ exactly once just like regular DFS procedure and the modifications happen in constant time.

The only difference is that the only root node gets called. All other nodes are automatically visited.

3(b) G: DAG

→ Since G is acyclic, we will call Top sort on it and get output vertices in atleast finite time.

→ We store vertices be stored in
Top. sorted_vertices.

Now, our DAG has transformed into a left to right ordered graph and we can call modified DFS simply based on vertices from TOP SORT.

All these edges would be left to right and each path will traverse till its very end.

→ Hence it would be like multiple directed trees. But there is a condition that if a child lies in two trees, it may get processed by one tree and hence the 2nd tree might skip it.
To avoid that use we modify our algorithm from part a.

Algorithm.

```
1. top-sorted-vertices = TOP-SORT(G)
   DFS-VISIT(G,u):
2.    time = time + 1.
3.  u.d = time
4.  u.color = GRAY.
5.  u.min = u.val
6.  for each v ∈ G.Adj[u]
7.        if v.color == WHITE
              v.π = u
```

```
8.          DFS-VISIT(G, v).
9.          u.min = min(u.min, v.min)
10. u.color = BLACK.
11. time = time + 1
12. u.f = time.
```

DRIVER.

DFS(G):

```
1. for each u ∈ G.V.
2.        u.color = WHITE
3         u.min = ∞
4. time = 0.
5. for v ∈ top.sorted-vertices:
6.    DFS-VISIT(G, V)
```

Correctness:

This algorithm is just like part a, but we have multiple directed trees (since its a DAG). So we need to arrange the vertices such that they flow in one order and there are no back edges.
This ensures that the min value gets correctly computed for all vertices.

We call TOPSORT (from lecture) to get the order of vertices and then call modified DFS on this ordered set of vertices. The DFS-VISIT is exactly

same as part (a) except the change in 9th line where we check $v.min$ even if v can be part of multiple directed paths in a graph.

In this way we find min for all vertices of the graph.

Runtime : Topsort = $O(m+n)$

DFS = $O(m+n)$

modifications = $O(1)$

∴ Total runtime = $O(m+n)$ as each node is visited exactly once like regular DFS & modifications happen in constant time.

3(c). G: Arbitrary Directed Graph.

We will use SCC algorithm from lecture and part b's algorithm which calculates $v.min$ for v in G where G is directed acyclic graph.

The algorithm will start by calling SCC(G) which will return G^{SCC}

$$G^{SCC} = (V^{SCC}, E^{SCC})$$

and let $C_1, C_2, \ldots, C_K \in V^{SC}$ where $C_1, C_2, \ldots C_K$ represent the SCC of G.

Note: G^{SC} is a DAG.

We modify $c_1, c_2, \ldots c_k$ to hold a min value which is actually the minimum of all the values of their respective vertices.

i.e for $C \in V^{scc}$.

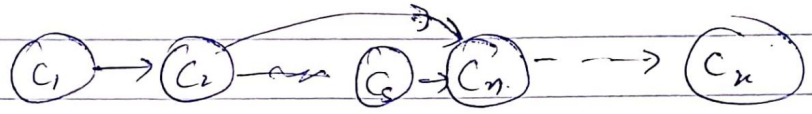
$C.min = \infty$

for $V \in C$:

$C.min = \min(V.val, C.min)$.

$C.val = C.min$.

Now we have initialized C.val & C.min for all $C \in V^{scc}$.

G^{scc}: 

It is a DAG.

→ Now, we call part b's algorithm on G^{scc}.

→ Note: vertices of G^{scc} are already in topologically sorted.

→ Therefore after calling part b's algorithm, we have appropriate C.min values $\forall C \in V^{scc}$

→ This C.min value will be passed to all vertices $V \in C$.

→ for $C \in V^{scc}$

for $V \in C$

$V.min = C.min$

→ Thus all vertices store their respective C.min value into their own and the algorithm ends.

Correctness:
This algorithm runs of SCC() which was proven correct in lecture & algorithm from part B which was justified earlier. The arbitrary directed graph is first converted to a directed acyclic graph with the help of SCC(). Then this DAG is initialized & updated so that it can be correctly used as an input for part B's algorithm.

This works because in a SCC, all vertices are reachable to all others in some version of DFS calls. Hence they ~~are~~ all need to have the same min value. We use this fact to treat the SC's as vertices with a min value in a DAG. And simply call part B's algorithm over it.

The C.min values get updated, returned & stored in each of this own vertex.
Hence we have justified its correctness.

Runtime: SCC - $O(m+n)$

Part B algorithm - $O(m+n)$

Initializing & updating C.min values & V.min values

- $O(n)$

as each node gets compared/updated only once

Total Runtime - $O(m+n)$.

2 Path Minimum **14 / 14**

✓ **- 0 pts** *Correct*