CS/ECE 374 P22

Pengxu Zheng, Jiawei Tang

TOTAL POINTS

95 / 100

QUESTION 1

1 22.A. **20 / 20**

✓ **- 0 pts** **Correct**

- 5 pts Correct with minor errors (typos or something)

- 10 pts Correct algo but non-linear time

- 20 pts Wrong algo that is not DAG

- 15 pts IDK

- 5 pts No/wrong run time analysis

- 10 pts Only code, not enough explaination

QUESTION 2

2 22.B. **10 / 10**

✓ **- 0 pts** **Correct**

- 2.5 pts Correct with minor errors (typos or something)

- 5 pts Correct algo but non-linear time

- 10 pts Wrong algo that gives wrong bridge

- 2.5 pts Missing/wrong running time analysis

- 5 pts Not enough explaination about the code

- 7.5 pts IDK

QUESTION 3

3 22.C. **20 / 20**

✓ **- 0 pts** **Correct**

- 5 pts Minor errors

- 20 pts Incorrect (see comments below)

- 15 pts IDK

QUESTION 4

4 22.D. **45 / 50**

- 0 pts Correct algo and proof

- 30 pts Correct algo without proof

- 10 pts [If not rubric 2] Wrong proof that root of DFS tree can reach any vertex

- 20 pts [If not rubric 2] Wrong proof that any vertex can reach root of DFS tree

- 5 pts Minor errors

✓ **- 5 pts** **Missing/wrong running time analysis**

- 50 pts Incorrect (see comments below)

- 37.5 pts IDK

HW Solution

CS/ECE 374: Algorithms & Models of Computation, Spring 2019 Version: **1.0**

Submitted by:

- **≪Jiawei Tang≫**: **≪jiaweit2≫**
- **≪Pengxu Zheng≫**: **≪pzheng5≫**

22

Solution:

22.A. For this problem, we are trying to orient an undirected graph to make it a DAG. Therefore, we need to avoid cycles when we orient the graph. The basic idea of my algorithm is to use DFS to orient each edge. The default orientation will follow the direction of how DFS goes. Once if the next vertex is visited, we orient this edge to the reverse direction of the DFS in order to avoid cycles. The detailed pseudocode is as below. We are given two vertices, s and t. In DFS-ALL(G), we want to start with s. At the beginning, we want to call DFS-All(G).

DFS-All(G):

```
for each v ∈ V do
  unmark v
end for
for each v ∈ V do(starts with s) do
  if v is unmarked then
    DFS(G, v)
  end if
end for
```

DFS(G, v):

```
mark v
for each u ∈ neighbors of v that the edge (u, v) is undirected do
  if u == t then
    Orients v → u
  else if u is marked then
    Orients v ← u//reversed orientation
  else
    Orients v → u
    DFS(G, u)
  end if
end for
```

This algorithm is based on the DFS algorithm in the lecture slides so it has linear time complexity, $O(n + m)$ where n is number of vertices and m is number of edges. This algorithm works like some vertices that are on the high ground will have directed edges to the vertices on the low ground but vertices can never have directed edges to the ground that is higher than itself(can't go back to its ancestor).

1 22.A. **20 / 20**

- ✓ **- 0 pts** Correct
- **- 5 pts** Correct with minor errors (typos or something)
- **- 10 pts** Correct algo but non-linear time
- **- 20 pts** Wrong algo that is not DAG
- **- 15 pts** IDK
- **- 5 pts** No/wrong run time analysis
- **- 10 pts** Only code, not enough explaination

22.B. We want to find a bridge on a connected undirected graph. We will have an array *disc* which records the discovery time for each vertex. We will then have another array *low* which records the earliest visited vertex that can be reached from subtree rooted with each vertex. We can define *low* as:

$$low[u] = min(disc[u], disc[w])$$

where u is an arbitrary vertex, w is an ancestor of u and there is a back edge from some descendent of u to w. If there are two vertices v, u such that $low[v] > disc[u]$. Then the edge (u, v) is a bridge.

$v =$ an arbitrary vertex
$time = 1$
return $DFS(v)$

$DFS(v)$ is defined as below:

mark v
$low[v] = disc[v] = time++$
for each $u \in$ neighbors of v **do**
  if u is visited **then**
    $low[v] = min(low[v], disc[u])$
  else
    $bridge = DFS(u)$
    if $bridge \neq null$ **then**
      return $bridge$
    end if
    $low[v] = min(low[v], low[u])$
    if $low[u] > disc[v]$ **then**
      return edge (u, v)
    end if
  end if
end for
return $null$

This algorithm has linear time complexity, $O(m)$ where m is the total number of edges.

2 22.B. **10 / 10**

- ✓ **- 0 pts** Correct
- **- 2.5 pts** Correct with minor errors (typos or something)
- **- 5 pts** Correct algo but non-linear time
- **- 10 pts** Wrong algo that gives wrong bridge
- **- 2.5 pts** Missing/wrong running time analysis
- **- 5 pts** Not enough explaination about the code
- **- 7.5 pts** IDK

22.C. When we direct an undirected connected graph G, the resulting directed graph G' can be either strongly connected or not strongly connected. We want to prove that if there exists a bridge in G, then G' can't be strongly connected. By definition, there exists a path between any two vertices in G'. On the other hand, a bridge is defined to be an edge that its removal disconnects the graph. If G has a bridge, it means that there will be two connected components a, b. For any vertex v in a, and any vertex u in b, the path from v to u or u to v must contain the bridge. Therefore, once we direct G, the bridge edge will be oriented. G' can't be strongly directed because there will only exist a path from v to u or u to v but NOT both. Therefore, the claim holds.

22.D.(discussed with Zhuoyue Wang's group)
We use the DFS codes from Mar.12 Lecture B's slides(26/60). The right function $DFS(u)$ should be rewritten as below:

Mark u as visited
$pre(u) = ++time$
for each uv in $Out(u)$ **do**
  if v is not marked **then**
    add edge (u, v) to T(Orient $u \to v$)
    $DFS(v)$
  else if uv is not directed **then**
    add edge (v, u) to T(Orient $v \to u$)
  end if
end for
$post[u] = ++time$

Define vertices $v_1, v_2, ...$ are ordered in discover order. For any vertex v_i, v_i or its descendent v_k must have a back edge. Otherwise, (v_{i-1}, v_i) is a bridge. It means that there can be a path between v_i and its ancestor v_j through v_k which is a descendent of v_i, where $j \leq i \leq k$. We can have three cases to discuss for v_i to reach an arbitrary vertex v_x.
If $j \leq x < i$, we can have the path v_i $\to$(through discovery edges) v_k $\to$(through back edges) v_j $\to$(through discovery edges)v_m.
If $x < j$, we can have the path v_i $\to$(through discovery edges) v_k $\to$(decendent of v_k) v_q $\to$(through back edge) v_x, where $x < j \leq q$
If $i < x$, v_i $\to$(through discover edges) v_x.

3 22.C. **20 / 20**

✓ **- 0 pts** Correct

- 5 pts Minor errors

- 20 pts Incorrect (see comments below)

- 15 pts IDK

22.C. When we direct an undirected connected graph G, the resulting directed graph G' can be either strongly connected or not strongly connected. We want to prove that if there exists a bridge in G, then G' can't be strongly connected. By definition, there exists a path between any two vertices in G'. On the other hand, a bridge is defined to be an edge that its removal disconnects the graph. If G has a bridge, it means that there will be two connected components a, b. For any vertex v in a, and any vertex u in b, the path from v to u or u to v must contain the bridge. Therefore, once we direct G, the bridge edge will be oriented. G' can't be strongly directed because there will only exist a path from v to u or u to v but NOT both. Therefore, the claim holds.

22.D.(discussed with Zhuoyue Wang's group)
We use the DFS codes from Mar.12 Lecture B's slides(26/60). The right function $DFS(u)$ should be rewritten as below:

Mark u as visited
$pre(u) = ++time$
for each uv in $Out(u)$ **do**
  if v is not marked **then**
    add edge (u, v) to T(Orient $u \to v$)
    $DFS(v)$
  else if uv is not directed **then**
    add edge (v, u) to T(Orient $v \to u$)
  end if
end for
$post[u] = ++time$

Define vertices $v_1, v_2, ...$ are ordered in discover order. For any vertex v_i, v_i or its descendent v_k must have a back edge. Otherwise, (v_{i-1}, v_i) is a bridge. It means that there can be a path between v_i and its ancestor v_j through v_k which is a descendent of v_i, where $j \leq i \leq k$. We can have three cases to discuss for v_i to reach an arbitrary vertex v_x.
If $j \leq x < i$, we can have the path v_i →(through discovery edges) v_k →(through back edges) v_j →(through discovery edges)v_m.
If $x < j$, we can have the path v_i →(through discovery edges) v_k →(decendent of v_k) v_q →(through back edge) v_x, where $x < j \leq q$
If $i < x$, v_i →(through discover edges) v_x.

4 22.D. **45 / 50**

- **- 0 pts** Correct algo and proof
- **- 30 pts** Correct algo without proof
- **- 10 pts** [If not rubric 2] Wrong proof that root of DFS tree can reach any vertex
- **- 20 pts** [If not rubric 2] Wrong proof that any vertex can reach root of DFS tree
- **- 5 pts** Minor errors
- ✓ **- 5 pts Missing/wrong running time analysis**
- **- 50 pts** Incorrect (see comments below)
- **- 37.5 pts** IDK