PS7-231830106

Problem 1

(a):

$(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16)$
$Find(x_2) = Find(x_9) = x_1$

(b):

先对n个节点进行n-1次Union操作，形成了一个高度为$\log n$的二叉树，再进行$m-n+1=O(m)$次Find操作，显然此时总时间复杂度为$O(m\log n)$

Problem 2

```
diretions =[(1,0),(-1,0),(0,1),(0,-1)]

def DFScomponet(x,y,B,visited):
        if B[x][y] == 0 or visited[x][y] == 1:
                return 0
        visited[x][y] = 1
        size = 1

        for dx,dy in directions:
                x, y = x + dy, y + dy
                if x<=n and x>=1 and y>= 1 and y<=0 and visited[x][y] = 0:
                        size += DFScomponet(x,y,B,visited)

        return size

def FindmaxCompoent(B,n):
        visited = [[0] * n for _ in range(n)]
        max_size = 0

        for x=1 to n:
                for y=1 to n:
                        size = DFScompoent(x,y,B,visited)
                        max_size = max(max_size,size)

        return max_size
```

时间复杂度：最坏情况下每个节点需要访问一次，即时间复杂度为$O(n)$。

Problem 3

(a):

A A.p=Null, A.d=0
B B.p=A,B.d=1; E E.p=A,E.d=0
C C.p=B C.d=2; F F.p=E F.d=2
I I.p=f I.d=3
D D.p=Null D.d=0
G G.p=D G.d=1; H H.p=D H.d=1

(b)

一、边

1. 树边: (A,B) (B,C) (C,D) (D,H) (H,G) (G,F) (F,E)
2. 回边: (D,B) (F,G) (E,D) (E,G)
3. 前向边:(B,E) (A,F)
4. 横向边:Null
   二、 时间戳
5. A 1 16
6. B 2 15
7. C 3 14
8. D 4 13
9. E 8 9
10. F 7 10
11. G 6 11
12. H 5 12

(c)

A B C D E F G H

Problem 4

(a):

$V = s, a, b, c, d$
$E = \{(s, a), (s, b), (a, c)(b, c)(c, d)\}$
源点：s
$E_{tree} = \{(s, b), (b, c), (c, d)\}$
bfs_result={(s,a),(s,b),(a,c),(c,d)}

(b):

$V = s, a, b, c, d, e$
$E = \{(s, a), (a, b), (b, c)(c, d)(s, e), (e, c)\}$
$E_{e-c} = (e, c)$
e.d = 1 c.d = 3
$DFStree = \{(s, a), (a, b), (b, c)(c, d)(s, e)\}$

(c)

当G中有环时，拓扑排序无法给出一个答案，因此命题不成立

Problem $I.1$

1. 思路：在BFS的基础上，为每个节点加上一个新属性d，用于描述到s点点距离模三的结果。当访问到(t,0)时为true，否则为false。
2. 伪代码

```
Algorithm HasWalkDivisibleBy3(G = (V, E), s, t):
    Input:
        G = (V, E): 有向图，其中 V 是顶点集合，E 是边集合
        s: 起始节点
        t: 目标节点

    Output:
        返回 True 如果存在从 s 到 t 的路径，且路径长度是 3 的倍数；否则返回 False

    // 初始化
    Create a queue Q and enqueue (s, 0)   // (current_node, length % 3)
    Create a 2D array visited[n][3] and set all values to False
    visited[s][0] = True                 // 起始节点 s，路径长度为 0，余数为 0

    // 开始 BFS
    while Q is not empty:
        (current, remainder) = Dequeue(Q)

        // 检查是否到达目标节点 t，且路径长度为 3 的倍数
        if current == t and remainder == 0:
            return True

        // 遍历 current 节点的所有邻居
        for each neighbor in Adj[current]:
            next_remainder = (remainder + 1) mod 3
```

```
            // 如果 neighbor 该余数状态未访问过
            if visited[neighbor][next_remainder] == False:
                visited[neighbor][next_remainder] = True
                Enqueue(Q, (neighbor, next_remainder))

    // 若未找到满足条件的路径，返回 False
    return False
```

3. 时间复杂度：每个节点至多访问3次，因此时间复杂度仍为O(|V|+|E|)

Problem $I.2$

1. 思路：和I.1类似，用(u,lc,cc)表示搜索状态，u表示当前节点，lc表示上一个颜色，cc表示连续的颜色，在此基础上进行bfs即可
2. 伪代码：

```
Algorithm ShortestWalkWithColorConstraint(G = (V, E), s, t):
    Input:
        G = (V, E): 有向图，其中 V 是顶点集合，E 是边集合，边有红色和蓝色两种颜色
        s: 起始节点
        t: 目标节点

    Output:
        返回从 s 到 t 的最短路径长度，满足路径中没有三个连续相同颜色的边

    // 初始化
    Create a queue Q and enqueue (s, None, 0, 0)  // (current_node, 
last_color, streak, distance)
    Create a 3D array visited[n][2][3] and set all values to False
    visited[s][0][0] = True
    visited[s][1][0] = True  // last_color 为 None 时，两个状态都标记为访问过

    // BFS 搜索
    while Q is not empty:
        (current, last_color, streak, dist) = Dequeue(Q)

        // 检查是否到达目标节点 t
        if current == t:
            return dist

        // 遍历 current 的所有邻居
        for each neighbor in Adj[current]:
            edge_color = Color(current, neighbor)  // 获取边 (current, 
```

```
neighbor) 的颜色

            // 检查转移状态
            if edge_color != last_color:
                // 边颜色变化，重置 streak 为 1
                next_streak = 1
                color_index = 0 if edge_color == 'R' else 1

                if not visited[neighbor][color_index][next_streak]:
                    visited[neighbor][color_index][next_streak] = True
                    Enqueue(Q, (neighbor, edge_color, next_streak, dist +
1))

            elif streak < 2:
                // 边颜色相同，且 streak < 2 时可以继续
                next_streak = streak + 1
                color_index = 0 if edge_color == 'R' else 1

                if not visited[neighbor][color_index][next_streak]:
                    visited[neighbor][color_index][next_streak] = True
                    Enqueue(Q, (neighbor, edge_color, next_streak, dist +
1))

    // 若未找到路径，返回 -1
    return -1
```

3. 时间复杂度：每个节点至多访问6次，因此时间复杂度仍为O(|V|+|E|)

Problem $I.3$

1. 思路：先找到所有入度为0的点，即第一学期需要修的点，设为集合S0，删除S0中点所有的出边，再寻找所有的入度为0的点，如此重复直至最后一个点被删除后没有入度为0的点，此时的集合数即为学期数量
2. 伪代码：

```
Algorithm MinSemesters(G):
    Input:
        G = (V, E): 课程先修图
    Output:
        semesters: 完成所有课程所需的最小学期数

    // Step 1: 计算所有节点的入度
    in_degree = [0] * |V|
```

```
for each (v, w) in E:
    in_degree[w] += 1

// Step 2: 初始化队列，用于存储当前入度为 0 的节点
queue = []
for each v in V:
    if in_degree[v] == 0:
        queue.append(v)

// Step 3: 初始化学期计数器
semesters = 0

// Step 4: 拓扑排序，逐层删除入度为 0 的节点
while queue is not empty:
    // 新学期开始，处理当前所有入度为 0 的节点
    next_queue = []  // 存储下一轮入度为 0 的节点
    for each course in queue:
        for each (course, neighbor) in E:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                next_queue.append(neighbor)

    // 更新队列
    queue = next_queue
    // 增加学期数
    semesters += 1

// Step 5: 返回学期数
return semesters
```