Problem Set 4 Answer Sheet

151220131 谢旻晖

Problem 5.1

算法见 **Algorithm** 1

Problem 5.2

只可能是 CE。　说明如下：在 u 发现之前 v 已经完成了遍历，也即点着色为 u 白色，v 黑色，显然只可能是 Cross Edge

Problem 5.3

采用反证法证明，如果一个收缩图有从强连通片 x 到强连通片 y 的环，即有强连通片 x 到强连通片 y 的有向路径 $x \rightsquigarrow y$，同时也有强连通片 y 到强连通片 x 的有向路径 $y \rightsquigarrow x$，那么 x 与 y 就相互可达，这与 x,y 都是强连通片矛盾。

Problem 5.4

第一次 DFS 不可以被替换为 BFS，原因是：第一次 DFS 中每遍历完成一个点有一个 post order 的入栈操作，这是 BFS 不可完成的。

而第二次 DFS 仅仅是为了遍历而已，可以被替换为 BFS。

Problem 5.5

DFS 树根 v 是割点的充要条件为：v 有两棵及两棵以上子树。
证明：

$\Longrightarrow$:

采用反证法，如果结论不成立，那么 v 没有或有 1 棵子树。如果 v 没有子树，则图为平凡图或不连通图，显然 v 不是割点，矛盾; 如果 v 只有一个子树，则去掉 v 之后剩下的图也显然连通，矛盾；所以 v 有两棵及两棵以上子树。

$\Longleftarrow$:

显然成立，去掉 v 之后两棵子树不再连通了。

Problem 5.6

仍然可以正确。$back(v)$ 的意义为 v 的所有子树的 back edge 能回溯到的最早的 $discovertime$，当 v 可以回溯到更早的结点时，$back(v)$ 就一定会被更新了。如果 v 不是割点，那么经过更新缩小后的 $back(v)$ 一定小于等于 $discovertime(v.parent)$。所以把 $back(v)$ 初值往大里放缩，我们只是先初始化所有点为割点而已，再一个个筛去非割点的点而已，并没有影响。

Problem 5.7

1.

首先证明引理：若无向图 G 中各顶点的度均大于或等于 2，则 G 中必有回路。证明：在图 G 中寻找一条最长路 P, 并设其最后一个节点为 v, 考察 v 的邻点. 易知, v 的所有邻点都在 P 上, 否则, 若 u 是 v 的一个不在 P 上的邻点, 则 $P + v \to u$ 是一条更长的路, 矛盾. 由于 G 中每个顶点的度都大于等于 2, 故 v 存在一个邻点 x, 且 x 在 P 上, 但 x 与 v 在 P 上不相邻, 此时路 $x \xrightarrow{P} v$ 与 $v \to x$ 就构成了一个环.

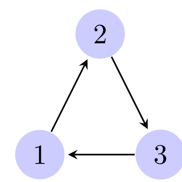
接着证明原命题：如果图中有度为 1 的点，那么删除任意一个这样的点之后 G 显然仍然连通; 如果图中没有度为 1 的点，那么图中所有点的度数均 ≥ 2,

Algorithm 1 NON_RECURSIVE_DFS

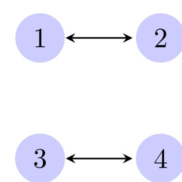
```
procedure DFSSweep(G)
//初始化颜色为 white
for each vertex v in V do
    color[v]=white
end for
for each vertex v in V do
    if color[v]==white then
        dfs(v)
    end if
end for
end procedure

//子过程 dfs
procedure dfs(v)
color[v] = gray
stack.push(v)
while !stack.empty do
    v=stack.top()
    v_isfinished=true
    for each u in adj[v] do
        if color[u]==white then
            color[u]=grey
            push(u)
            v_isfinished=false
            break
        end if
        if v_isfinished then
            color[v]=black
            stack.pop() //把 v 出栈
        end if
    end for
end while
end procedure
```

由引理图必定存在环，删除环上的任意一点之后，G 仍然连通。

2.

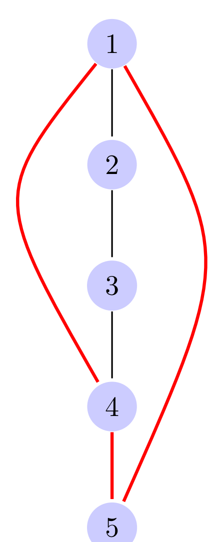


3.



Problem 5.8

该算法仅仅检测了由一条 BE $v \to w$ 和一些 TE 形成的路径 $w \rightsquigarrow v$ 构成的环, 而对于由两条 BE, 一些 TE 构成的环无法检测。例子如下: 下图中用红色加粗标记的环长为 3 的最小环 $1 \to 4 \to 5 \to 1$ 无法被检测到，这就导致了算法的错误。



Problem 5.10

首先易证如果这个图有 2 个及以上的入度为 0 的点，这个 DAG 就不存在哈密顿通路。基于这个事实，对原来的 DAG 进行一次拓扑排序，在拓扑排序途中如果有 2 个及以上入度为 0 的点，就不存在哈密顿通路。拓扑排序顺利结束，则存在这样一条哈密顿通路，且拓扑序就是通路点序列。

Problem 5.11

Algorithm 2 TOPO_SORT

```
degree[1...|V|] //所有结点的入度数组
queue
统计所有点的入度，存入 degree[]
queue.push(所有入度为 0 的点)
while not queue.empty() do
  v=queue.pop() //取出队头元素
  for each j in adj(v) do
    degree[j]=degree[j]-1
    if degree[j]==0 then
      queue.push(j)
    end if
  end for
end while
```

1.

算法思想：维护一个入度为 0 的点队列：初始化时统计所有点的入度，将入度为 0 的点进队。只要队列不为空，就取出队头并输出，同时处理以队头为起点的所有有向边，将边的终点的入度减一，如果减到 0 了，就入队。

算法伪代码，见 **Algorithm** 2

算法复杂度：一个点最多入队一次，一条边最多被删去一次，所以整体复杂度为 $O(m+n)$

2.

会在算法执行到一定时候找不到入度为 0 的点，非正常退出，拓扑排序无法完成。

Problem 5.12

1.

DFS(s)，看是否可以遍历完所有点。

2.

使用 SCC 收缩有向图 G，设收缩之后的 DAG 图为 G'。对 G' 进行拓扑排序，对起点（排序为第一的）进行一次 DFS，如果可以遍历完 G' 中的所有点，则存在这样一个"one-to-all" 的顶点，否则不存在。

整体复杂度 =SCC+ 拓扑排序 + 一次 DFS=$O(m+n)$

Problem 5.13

首先，可以很容易证明同一连通分量中的点的影响力值是相同的。

1.

STEP 1: 对原图 G，进行 SCC 收缩，形成分量收缩图 G'。

STEP 2: 找到出度为 0 的那些连通分量收缩点，其中包含最少点数的那个连通分量中所有的点是影响力最小的点。

整体复杂度 $= O(m+n) + O(n) + O(n) = O(m+n)$。

2.

STEP 1: 对原图 G，进行 SCC 收缩，形成分量收缩图 G'。

STEP 2: 找到入度为 0 的那些连通分量收缩点，对其中的每个连通分量收缩点的某一点 x 进行一次 DFS，同时记录下这个 x 可达的顶点个数，也即是该 x 所属的连通分量中任意一点的影响力。

STEP 3: 找出其中影响力最大的连通分量，该分量中任意一点都是影响力最大的点。

整体复杂度 $= O(m+n) + n * O(m+n) + O(n) = O(mn+n^2)$

Problem 5.14

对原图求关键路径，关键路径长度即为最少学期数。

算法复杂度为 $O(m+n)$。

Problem 5.16

1.

将问题建成图模型，每个小孩子都是一个顶点，如果"i 恨 j"，则向图中添加一条 $i \to j$ 的有向边，最终形成的图为 G。

进行一次 DFS 检测是否有环，若无环，根据结点的 $finishTime$ 对 G 进行逆拓扑排序，逆拓扑排序的顺序就是小孩排队的顺序；否则，G 存在环，不存在符合条件的排队方法。

2.

将问题建成图模型，每个小孩子都是一个顶点，如果"i 恨 j"，则向图中添加一条 $i \to j$ 的有向边，最终形成的图为 G。

利用 DFS 求出 G 的关键路径长度，即为最少行数。若 G 有环（即 DFS 中有 $Back\ Edge$），此时不存在满足要求的排法。

Problem 5.18

BE: 当某个点发现一条 BE 时，由无向图的特性，祖先早已用这条边遍历了它，这就矛盾了。

FE: 由于 BFS 没有回退的过程，所以绝对没有 FE。

Problem 5.19

1.

可以的，我们设图的两部分可以最终被着色为蓝和红色。我们从任意一点开始，把他染成蓝色，进行无向图的 DFS。

在 DFS 的过程中：$preorder$ 部分，将该结点染成**非**父结点颜色的颜色；每当即将访问一个已经访问过的点（Check NonTree Edge），检测该点与自身是否颜色不同，如果相同，则该图不是二部图。完成遍历之后，判定该图为二部图。

2.

本题显然使用 BFS 更佳，BFS 与 DFS 的优劣对比如下:

从处理角度来看：DFS 在有 $postorder$ 后序的处理时有优势，BFS 则没有后序处理。

从问题类型来看：DFS 对于解决遍历和求所有问题有效，对于问题搜索深度小的时候处理速度迅速，然而在深度很大的情况下效率不高;BFS 对于解决最短或最少问题特别有效，而且寻找深度小。

Problem 5.20

从任意一点进行 DFS，如果遇到了 $BackEdge$，那么就可以移除它，且去掉之后 G 仍然连通（由于 DFS 生成树），否则就不存在。

算法复杂度分析：无向图的 DFS 中只存在 $TreeEdge$ 和 $BackEdge/ForwardEdge$(从不同方向看类型不同)。且最多有 $n-1$ 条 $TreeEdge$ 边，当检测到第 n 条边的时候他一定是 $BackEdge/ForwardEdge$, 这就保证了我们的算法最多只要检测 n 条边, 所以复杂度为 $O(n)$。

Problem 5.22

算法思路：维护一个存有当前最大的完美二叉树**集**的队列 $queue$,$queue$ 中元素具有的形式为 [h,root]，h 为树高，root 为树根。每当向队列中压入一棵完美二叉树时，就把队头所有高度小于他的树全部出队。这样，算法结束后，队列中所有树就是最大完美子树。

算法伪代码见 **Algorithm** 3。

Algorithm 3 PERFECT_SUBT(*root*)

Input: root 为树的根结点

Output: [h,isPerfect] h 为树的高度，isPerfect 是一个布尔型变量代表是完美二叉树

```
GLOABL queue
if root==NULL then
  return [0,true]
end if
[h_L,isPerfect_L]=PERFECT_SUBTREE(root->left)
[h_R,isPerfect_R]=PERFECT_SUBTREE(root->right)
if h_L == h_R && isPerfect_L && isPerfect_R then
  queue.push([h_L + 1,root])
  while queue.top().h<h_L + 1 do
    queue.pop()
    //让队列前端树高小于当前元素的树们出队
  end while
  return [h_L + 1,true]
else
  return [max(h_L, h_R) + 1,false]
  //返回树高，false 代表非完美二叉树
end if
```

Problem 5.24

1.

DFS–有向图: 当检测到有 $BackEdge$ 时，原图有环。

BFS–无向图: 无向图只有 $TreeEdge$ 和 $CrossEdge$，所以检测到有 $CrossEdge$ 时，原图有环。

2.

采用这样的 $adversary$ 策略，对于顶点个数 n，构造一个 $|E| = n^2$ 条有向边的图，则 $O(n)$ 的算法不可能遍历完所有的边。可以构造这样的一个恶意输入，使得算法老师的算法遍历到的边没有环，而剩下的边我们随意构造出一个环, 使得算法产生错误的结果，这就证明了算法老师的算法的错误性。

Problem 5.25

1.

由于边权值为 1，任何生成树都是最小生成树。所以，从任意一点出发进行一次 DFS，得到生成树，权为 $n - 1$。

2.

使用 DFS 找出所有的非割边，从中删去权值最大的一条，重复 11 次，直到剩下 $m = n - 1$ 条边即为 MST。

算法复杂度为 $11 * O(n + m) = O(n)$。

3.

使用类似 kruskal 算法的方法，首先将所有边分类为权值为 1 的和为 2 的；去掉原图中的所有边，往图里不断加入权值小的 (权为 1 的用完了用权为 2 的) 且不构成环的边，直到图所有结点连通。

算法复杂度为: 边分类 $O(m)$, 不断加边要加 $O(n)$ 次，每次检测是否有环为 $O(1)$(使用并查集)，总共 $O(m + n)$。