算法基础HW5

PB19071535徐昊天

问题1

答：

(a)模拟先进先出的队列需要两个栈，分别用于输入和输出，对应伪代码如下：

```
stack<int>stack_1
stack<int>stack_2

function push(x)    // 入队
    stack_1.push()

function pop()      // 出队
    if stack_2.isEmpty() == true
        while stack_1.isEmpty() == false
            stack_2.push(stack_1.pop())
    return stack_2.pop()
```

以上即为入队和出队的伪代码。

(b)利用聚合分析的方法分析以上每个操作的均摊代价：

易得push操作的时间复杂度为O(1);pop操作每次需将n个元素从stack_1中弹出并将这n个元素压入stack_2中，最后将栈顶元素弹出，时间复杂度为O(2n)。

由于每次pop操作时间复杂度为O(n)，n次pop时间复杂度即为$O(n^2)$。然而这并非一个确界。

假设执行了n次入队操作，则入队代价为n；将这n个元素从stack_1中弹出并压入stack_2中代价为2n；将stack_2中n个元素全部弹出代价为n。总代价为4n。故对队列进行n次合法操作所花费的代价不会超过4n，即为O(n)。

故每个操作的均摊代价为$\frac{O(n)}{n}=O(1)$。

问题2

答：

(a)考虑树的每一个结点及以它为根结点的子树，计算从该根结点到子树中其他结点的最长路径d1和次长路径d2(两个路径不存在重复部分)。所有节点中d1+d2的最大值即为所求的最长路径。对应伪代码如下：

```
function find_path(T,node,max_length,max_path)
    d1=d2=0, path1=path2=""
    if node do not have child   // 结点为叶子结点，无孩子
        node.length=0
        node.path=node
    else if node has one child x    // 只有一个孩子结点
```

```
        find_path(T,x,max_length,max_path)
        node.length=d1=x.length+w(node,x)
        node.path=path1=x.path+node
        if d1>max_length
            max_length=d1
            max_path=path1
    else    // 有多个孩子结点
        for each child x of node
            find_path(T,x,max_length,max_path)
            node.length=d=x.length+w(node,x)
            node.path=path=x.path
            if d>d1
                d2=d1, path2=path1
                d1=d, path1=path
            else if d>d2
                d2=d, path2=path
        if d1+d2>max_length
            max_length=d1+d2
            max_path=path1+node+reverse(path2)

function longest_path(T)
    max_length=0,max_path=""
    find_path(T,T.root,max_length,max_path)
    return (max_length,max_path)
```

如上，对每一个结点分别维护length、path属性(用于记录以该结点为根结点的子树中以该结点为起点能到达的最长距离及其对应的路径)。采用递归的方式自底向上分别对结点分析，若为叶子节点则将length置0、path置为单个节点；若只有一个孩子结点则只对一个孩子结点分析，判断该路径是否最长；若有多个孩子结点则从多个孩子结点中选出最长和次长路径并取和，判断该路径是否最长。递归结束后即可得到最长路径。

时间复杂度分析：

该算法对树中的每个顶点和每条边最多访问一次，根据树的性质：V=E+1。故算法的时间复杂度为$O(|V|)$。

正确性分析：

对于三种情况分别考虑：

1. 为叶子节点时，显然路径长度为0，故路径为单个叶子节点。
2. 仅有一个孩子结点时，最长路径一定以根结点为起点，否则不为最长路径。故路径为孩子结点的路径加上根结点。
3. 有多个孩子结点时，以根结点为起点可存在多个路径，故子树中最长路径一定为起点可到达的最长路径和次长路径之和。故路径为最长路径加上根结点再加上次长路径的转置。

故分为以上三种情况能够找到最长路径，可证算法正确。

(b)首先对于树中任意一个结点，寻找到距离该结点最远的结点u，再寻找到距离结点u最远的结点v，则(u,v)即为最长路径。对应伪代码如下：

```
function find_path(T,node,far_node,max_length,max_path)
    if node do not have child   // 结点为叶子结点，无孩子
        max_length=0
        max_path=node
        far_node=node
    else
        max_length=0
```

```
        for each child x of node
            find_path(T,x,far_n,length,path)
            if w(x,node)+length > max_length
                far_node=far_n  // 记录最远的结点
                max_path=path+node  // 记录到最远结点的路径
                max_length=length+w(x,node)     // 记录到最远结点的路径长度

function longest_path(T)
    let root be a random node of T
    max_length=0,max_path=""
    let u,v be nodes
    find_path(T,root,u,max_length,max_path)
    find_path(T,u,v,max_length,max_path)
    return (max_length,max_path)
```

如上，find_path函数通过深度优先搜索的方式寻找距离原结点最远的结点及其对应的最长路径和长度，该函数的far_node、max_length、max_path参数是用于返回结果的参数，分别返回：距离原结点最远的结点、原结点到最远结点的距离、原结点到最远结点的路径。主函数longest_path通过两次调用find_path函数先后得到距离root结点最远的u结点和距离u结点最远的v结点，得到的max_length、max_path即为所求的最长路径及其长度。

时间复杂度分析：

由于主函数longest_path两次调用find_path函数，find_path函数执行深度优先搜索，且根据树的性质：V=E+1。故算法的时间复杂度为$O(|V|)$。

正确性分析：

首先证明以下命题：

首次调用find_path函数得到的结点u一定存在于最长路径中。

证明如下：

用反证法，假设u不存在于最长路径中，最长路径的两个端点为a,b。分为以下情况讨论：

- root和u在(a,b)路径上最接近的点为同一个点c
    - root和u在不同侧

      如下图所示：

      ```
              root
               |
      a o------o------o b
               | c
               u
      ```

      考虑如上情况，由于u是root在树中能达到的最远结点，故有：

      w(root,u)>w(root,b)=>w(c,u)>w(c,b)=>w(c,u)+w(a,c)>w(c,b)+w(a,c)=>w(a,u)>w(a,b)

      故与原假设矛盾。
    - root和u在同侧

      如下图所示：

```
        root
         |
        d|---○ u
         |
a ○-----○-----○ b
         c
```

考虑如上情况，由于u是root在树中能达到的最远结点，故有：

w(root,u)>w(root,b)=>w(d,u)>w(c,b)+w(c,d)=>w(d,u)+w(a,c)+w(c,d)>w(c,b)+w(a,c)+2w(c,d)=>w(a,u)>w(a,b)+2w(c,d)=>w(a,u)>w(a,b)

故与原假设矛盾。

- root和u在(a,b)路径上最接近的点分别为点c,d
  - c较为靠近a,d较为靠近b

    如下图所示：

```
        root u
         |   |
a ○-----○---○--○ b
         c   d
```

    考虑如上情况，由于u是root在树中能达到的最远结点，故有：

    w(root,u)>w(root,b)=>w(d,u)>w(d,b)=>w(d,c)+w(a,c)+w(d,u)>w(d,c)+w(a,c)+w(d,b)=>w(a,u)>w(a,b)

    故与原假设矛盾。
  - c较为靠近b,d较为靠近a

    此情况与以上情况同理，根据a,b的对称性同理可证与原假设矛盾。

综上，原命题得证，结点u一定存在于最长路径中。故在此前提下结点u与查询到的最远结点v构成的路径一定为最长路径，故可证算法正确。

问题3

答：

考虑G的转置G^T，通过修改DFS算法从而计算所有顶点v的min(v)，伪代码如下：

```
DFS(G)
    for each vertex u ∈ G.V
        u.color=WHITE
        u.π=NIL
    time=0
    for each vertex u ∈ G.V sorted by subscript from smallest to largest
        if u.color == WHITE
            DFS-VISIT(G,u,u)
DFS-VISIT(G,root,u)
    time=time+1
    u.d=time
    min(u)=root.subscript
    u.color=GREY
    for each v ∈ G:Adj[u]
```

```
        if v.color==WHITE
            v.π=u
            DFS-VISIT(G,root,v)
    u.color=BLACK
    time=time+1
    u.f=time
```

首先得到G的转置G^T,再对G^T执行以上经过修改的DFS算法。DFS算法中根据每个顶点v_j的下标j从小到大进行遍历，并在DFS-VISIT函数中增加一个参数作为本次深度优先搜索的的根结点，在本次深度优先搜索到的所有顶点v都设置min(v)=root。DFS函数结束后便得到了所有顶点v的min(v)。

时间复杂度分析：

由于此算法是在DFS算法的基础上做出微量的修改，且对每个顶点和每一条边最多遍历一次，故算法时间复杂度与DFS相同，为$O(|V|+|E|)$。

正确性分析：

根据DFS算法的特点，某个顶点v能够在调用一次DFS-VISIT函数后遍历图中所有v可达的结点，此时v是遍历到的所有顶点的root。由于在DFS中根据顶点v_j的下标j从小到大进行遍历，故每一个顶点u都会被能够搜索到该顶点且下标最小的顶点作为root遍历到。故在DFS结束后每一个顶点v的min(v)储存的是能够到达该顶点的下标最小的顶点下标。

而由于是对原有向图G的转置G^T执行以上DFS算法，故在DFS结束后每一个顶点v的min(v)储存的是该顶点能够到达的下标最小的顶点下标，即为题中所求。

综上，该算法正确。

问题4

答：

利用定理：一张图是二分图，当且仅当图中不存在奇数长度的环路。对有向图G的每一个强连通分量分别考虑，伪代码如下：

```
function DFS(G1,u,c)
    u.color=c
    for each v ∈ G1:Adj[u]
        if v.color==c
            return false
        if v.color==-c && !DFS(v,-c)
            return false
    return true

function odd_cycle(G)
    STRONGLY-COMMECTED-COMPONENTS(G)
    for each strongly_connected_components in G
        let G1 be the strongly_connected_components in the form of an
undirected graph
        if DFS(G1,G1.root,1) == false
            return true
    return false
```

根据以上代码，首先调用算法导论中的算法STRONGLY-COMMECTED-COMPONENTS计算出有向图G中的所有强连通分量，然后对所有强连通分量单独进行分析，通过调用DFS函数判断其对应的无向图是否为二分图，若存在非二分图，则必然根据定理：`一张图是二分图，当且仅当图中不存在奇数长度的环路`，可知图中存在奇数长度的环路；反之则不存在奇数长度的环路。

时间复杂度分析：

根据算法导论，算法STRONGLY-COMMECTED-COMPONENTS的时间复杂度为$O(|V|+|E|)$;然后对每一个强连通分量分别调用的DFS算法对每个顶点和每条边最多遍历一次，故时间复杂度也为$O(|V|+|E|)$。

故该算法的时间复杂度为$O(|V|+|E|)$。

正确性分析：

首先证明以下结论：

有向图有一个奇数长度的有向环当且仅当它的某一个强连通分量是非二分的（当被视为无向图时）。

证明如下：

根据定理：`一张图是二分图，当且仅当图中不存在奇数长度的环路`，可以推出对于图G的某一个强连通分量，若其对应的无向图为非二分图，则该无向图中存在奇数长度的环路。根据强连通分量的定义，该奇数长度的环路中的所有顶点都应当在同一强连通分量中，否则违反了强连通分量的规则。故只需对每一个强连通分量单独分析，无需考虑跨越多个强连通分量的情况。

当无向图中存在奇数长度的环路时，考虑该无向图对应的原有向图。若该环路中存在某条边v->w方向与其他边不同，则需寻找另一条从w指向v的奇数长度的路径。根据强连通分量的定义，必然存在从w指向v的路径P：若P为奇数长度，则直接取代v->w，则可继续维持奇数环路；若P为偶数长度，则将v->w和路径P作为新的奇数环路。因此，当无向图中存在奇数长度的环路时，该无向图对应的原有向图也必存在奇数长度的环路。

故原结论得证。

故只需对G的每一个强连通分量对应的无向图判断是否为二分图即可判断G中是否存在奇数长度的环路。可证算法正确。

问题5

答：

首先考虑G中除去和S相关的顶点和边得到的图G'的最小生成树，然后将S中的顶点和部分有关边即可得到结果，具体伪代码如下：

```
function MST_withS(G,w,S)
    V' = G.V - S
    E' = {(u,v):u,v∈G.V-S∧(u,v)∈G.E}
    let G' = (V',E')
    T'= MST-KRUSKAL(G',w)
    E''= {(u,v):u∈S∧v∉S∧(u,v)∈G.E}
    for each vertex u∈S
        MAKE-SET(u)
    sort the edges of E'' order by weught w
    for each edge(u,v)∈E'',taken in nondecreasing order by weight
        if FIND-SET(u)!=FIND-SET(v)
            add (u,v) to T'
            UNION(u,v)
    return T'
```

根据以上代码，算法整体思路为：

首先考虑G将S集合中的顶点和S集合中的顶点相关的边全部删除的图G'，通过调用Kruskal算法找到G'的最小生成树T'；然后考虑S中的顶点，对S中顶点和G.V-S中顶点连接的边进行排序，然后对S中的每一个顶点，选出最短的一条边并连接到T'上，即得到最终结果。

时间复杂度分析：

首先考虑对G'调用Kruskal算法的部分，Kruskal算法的时间复杂度为$O(|E|lg|V|)$；考虑将S中顶点加入T'的部分，该部分相当于对Kruskal算法的适当修改，对象为S中的顶点和E''中的边，故时间复杂度为$O(|E|lg|V|)$。

综上，算法时间复杂度为$O(|E|lg|V|)$。

正确性分析：

首先考虑题中条件，由于S中顶点必须为最小生成树的叶子，故V-E中的顶点必须与另一个V-E中的顶点相连，即关联一条不与S中顶点关联的边。对于满足题中条件的生成树T'，若将该生成树中S中的顶点及其相关的边全部删除，得到的依然是一棵树(删除的全部是叶子)。故可将题中所求生成树的权重分为两部分考虑：删除S中的顶点及其相关的边得到的树的权重(w1)、S顶点与V-S顶点构成的边的权重(w2)。

根据以上算法，Kruskal算法能够得到最小的w1；对于S中的顶点，找到其与V-S顶点连接的最小边并加入T'中，故得到了最小的w2。两者相加，即得到了满足条件的最小生成树。

综上，可证算法正确。