PS8-231830106

Problem 1

Bacon教授的想法是错误的，可以举出一个简单的反例
V={a,b,c} E={(a,b),(b,c),(c,b)}
在G^R上从b点开始进行DFS，先访问c。会得到cab的顺序，此时如果按Bacon教授的想法进行DFS，在G^R上先从b开始DFS会访问到a，判断出ba为一个强连通分量，这很明显是错误的。
此反例可说明Bacon教授想法错误，即证

Problem 2

1. 思路：通过tarjan得到G的强连通分量，在每个强连通分量内部的点直接构造成一个环，再统计各个强连通分量之间的连通性，在不同环上任意点间加边即可。
2. 伪代码

```
def find_SCCs_tarjan(G):
    """使用 Tarjan 算法找到强连通分量"""
    index = 0
    stack = []
    indices = {}  # 节点对应的索引
    lowlink = {}  # 节点对应的最低可达节点索引
    on_stack = set()
    SCCs = []

    def tarjan(v):
        nonlocal index
        indices[v] = index
        lowlink[v] = index
        index += 1
        stack.append(v)
        on_stack.add(v)

        # 遍历邻居
        for neighbor in G[v]:
            if neighbor not in indices:
                tarjan(neighbor)
                lowlink[v] = min(lowlink[v], lowlink[neighbor])
            elif neighbor in on_stack:
                lowlink[v] = min(lowlink[v], indices[neighbor])
```

```
        # 如果当前节点是一个 SCC 的根节点
        if lowlink[v] == indices[v]:
            current_SCC = []
            while True:
                node = stack.pop()
                on_stack.remove(node)
                current_SCC.append(node)
                if node == v:
                    break
            SCCs.append(current_SCC)

    # 遍历所有节点
    for v in G:
        if v not in indices:
            tarjan(v)

    return SCCs

def construct_G_prime(G):
    """构造图 G' 满足条件"""
    # Step 1: 使用 Tarjan 算法找到强连通分量
    SCCs = find_SCCs_tarjan(G)

    # Step 2: 构造分量图
    SCC_index = {node: idx for idx, SCC in enumerate(SCCs) for node in
SCC}
    num_SCCs = len(SCCs)
    component_graph = {i: set() for i in range(num_SCCs)}

    # 遍历所有边，记录强连通分量之间的关系
    for u in G:
        for v in G[u]:
            u_comp = SCC_index[u]
            v_comp = SCC_index[v]
            if u_comp != v_comp:
                component_graph[u_comp].add(v_comp)

    # Step 3: 构造 G'
    G_prime = {v: [] for v in G}

    # 在每个强连通分量内部形成一个环
    for SCC in SCCs:
        for i in range(len(SCC)):
```

```
            G_prime[SCC[i]].append(SCC[(i + 1) % len(SCC)])

    # 在强连通分量之间添加边
    for u_comp in component_graph:
        for v_comp in component_graph[u_comp]:
            u_node = SCCs[u_comp][0]  # 取分量 u_comp 中的任意一个节点
            v_node = SCCs[v_comp][0]  # 取分量 v_comp 中的任意一个节点
            G_prime[u_node].append(v_node)

    return G_prime
```

3. 时间复杂度分析：
    1. Tarjan算法：时间复杂度为$O(|V|+|E|)$。
    2. 构造分量图，时间复杂度为$O(|E|)$
    3. 在强连通分量内部构造环，增加跨分量边，时间复杂度为$O(|V|+|E|)$。

Problem 3

(a):

1. 思路：进行拓扑排序，逆序进行cost的更新。
2. 伪代码：

```
def compute_cost_DAG(V, E, prices):
    # Step 1: 计算拓扑排序
    topo_order = topological_sort(V, E)  # 返回拓扑排序的节点列表

    # Step 2: 初始化 cost 数组
    cost = {u: prices[u] for u in V}

    # Step 3: 按拓扑排序更新 cost
    for u in reversed(topo_order):  # 从拓扑排序的最后开始处理
        for v in neighbors(u, E):  # 遍历 u 的所有邻居
            cost[u] = min(cost[u], cost[v])

    return cost
```

3. 时间复杂度：拓扑排序时间复杂度为$O(|V|+|E|)$，更新cost的时间复杂度为$O(|V|+|E|)$，因此总时间复杂度为$O(|V|+|E|)$。

(b):

1. 思路：先通过Tarjan算法获得分量图，对每个分量图挑选其中cost最小的作为代表元素，对代表元素组成的图，这个图显然为DAG，因此套用(a)中算法即可。
2. 伪代码：

```
def compute_cost_general(V, E, prices):
    # Step 1: 使用 Tarjan 算法找到强连通分量
    SCCs, component_map = tarjan_SCC(V, E)  # 返回 SCC 列表和每个节点所属分量

    # Step 2: 初始化每个分量的最小价格
    min_price = {}
    for C in SCCs:
        min_price[C] = min(prices[u] for u in C)

    # Step 3: 构造分量图（DAG）
    component_graph = build_component_graph(SCCs, component_map, E)

    # Step 4: 按拓扑排序更新分量的最小可达价格
    topo_order = topological_sort(component_graph)
    for C in reversed(topo_order):  # 按拓扑顺序处理分量
        for D in component_graph[C]:  # 遍历分量 C 的邻居
            min_price[C] = min(min_price[C], min_price[D])

    # Step 5: 将结果还原到每个节点
    cost = {}
    for u in V:
        cost[u] = min_price[component_map[u]]

    return cost
```

3. 时间复杂度：Tarjan的时间复杂度为$O(|V|+|E|)$，(a)中算法的时间复杂度为$O(|V|+|E|)$，则总的时间复杂度为$O(|V|+|E|)$。

Problem 4

(a):

V = {s,a,b} E = {(s,a),(s,b)}
显然a不能到达b，b也不能到达a

(b):

1. 思路：寻找图中入度为0的点，如果数量大于一则命题不成立；若数量等于一，则记这个点为s，同理记录出度为0的点为e，由离散数学知识可知，原命题的等价条件即为s到e存

在一条包括V中所有顶点的有向通路。接着进行循环删除入度为0的点及其出边，再检测是否有多个入度为0的点，若有则返回false，若无则继续循环。直至到达e

2. 伪代码

```
def Is_Singlepath_Dag(graph):
    # Step 1: 初始化入度和出度
    in_degree = [0] * |V|
    out_degree = [0] * |V|
    for (v,w) in E:
            in_degree[w]+=1
            out_degree[v]+=1

    # 找入度为 0 的点和出度为 0 的点
    sources = [v for v in graph if in_degree[v] == 0]
    sinks = [v for v in graph if out_degree[v] == 0]

    # 如果入度为 0 或出度为 0 的点不唯一，直接返回 False
    if len(sources) != 1 or len(sinks) != 1:
        return False

    s = sources[0]
    e = sinks[0]

    # Step 2: 循环删除入度为 0 的点
    queue = [s]
    visited = set()
    count = 0

    while queue:
        if len(queue) > 1:  # 如果同时存在多个入度为 0 的点
            return False

        u = queue.pop(0)  # 当前节点
        visited.add(u)
        count += 1

        # 删除 u 的出边，更新邻接节点的入度
        for (u,w) in E:
            in_degree[w] -= 1
            if in_degree[w] == 0:
                queue.append(w)
```

```
    # Step 3: 检查是否访问了所有节点并到达终点 e
    return count == len(graph) and e in visited
```

3. 时间复杂度：找到s和e的时间复杂度为$O(|V|+|E|)$，检测是否存在包含所有点的有向通路的算法类似拓扑排序时间复杂度为$O(|V|+|E|)$。

(c):

1. 思路：将原图先进行Tarjan算法找到所有强连通分量，进行缩点后，再进行(b)的算法即可。
2. 伪代码：

```
Function IsSortOfConnected(Graph G):
    # Step 1: 使用 Tarjan 算法找到强连通分量
    SCCs = TarjanSCC(G)  # 返回强连通分量列表
    ComponentGraph = BuildComponentGraph(SCCs, G)  # 构建分量图

    # Step 2: 对分量图应用 (b) 的算法
    Return Is_Singlepath_Dag(ComponentGraph)  # 调用之前的 DAG 判断函数
```

3. 时间复杂度：Tarjan构造分量图的时间复杂度为$O(|V|+|E|)$，(b)中算法的时间复杂度为$O(|V|+|E|)$，因此总时间复杂度为$O(|V|+|E|)$。