数据结构与算法

DATA STRUCTURE

第二十一讲 强连通分支

胡浩栋

信息管理与工程学院

2018 - 2019 第一学期

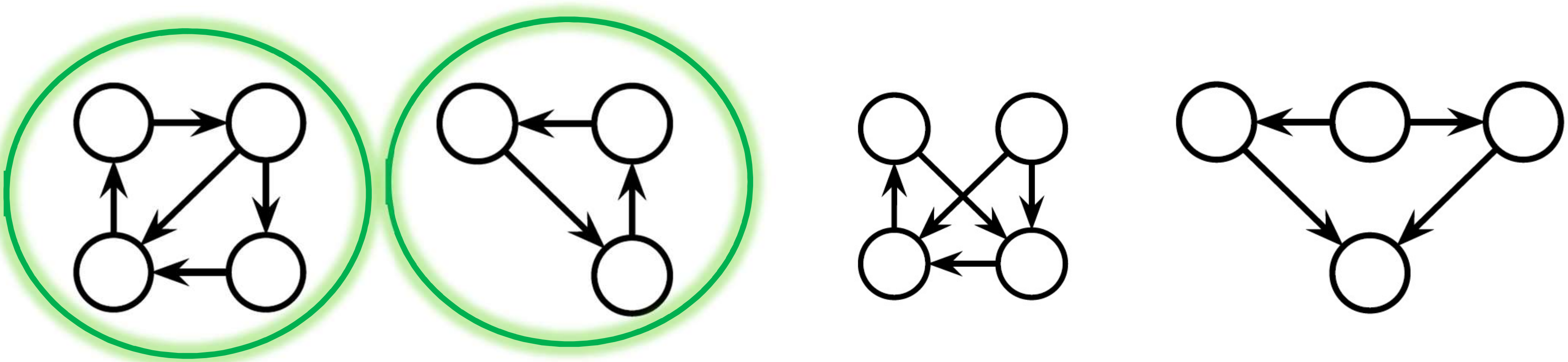
课堂内容

- 强连通分支
- 欧拉回路

强连通分支
Strongly connected components

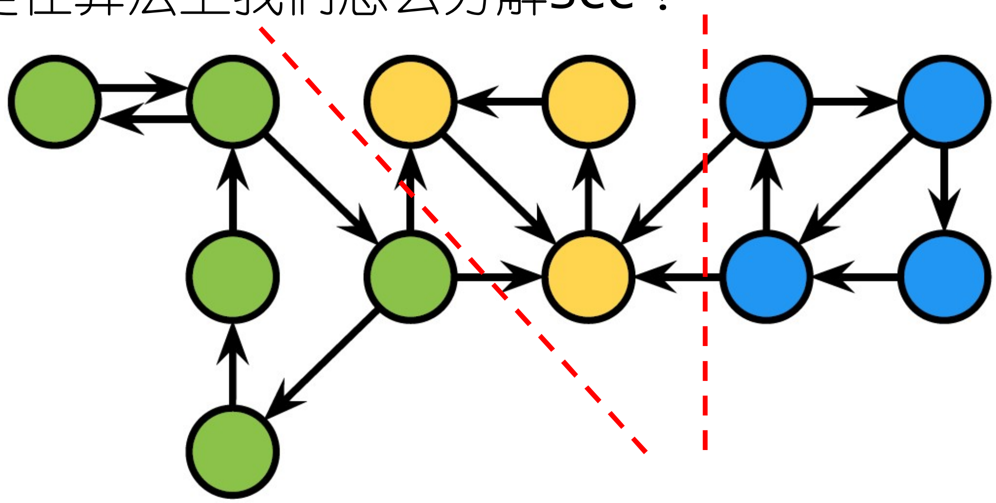
强连通性

- 一个有向图 $G = (V, E)$ 是强连通的**strongly connected**，如果对于任意顶点u，v，他们是**相互可达**的，即
  - 既有u到v的路径
  - 也有v到u的路径



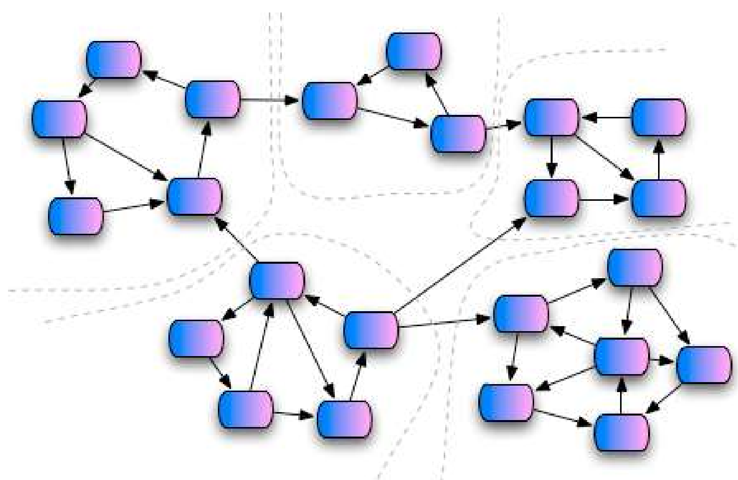
强连通分支

- Strongly Connected Components (缩写为*SCC*)
- 有向图的极大强连通子图称为G的*强连通分支*
  - 有向图不见得是强连通的
  - 有向图可以分解成强连通分支
    - 可达具有传递性，**相互可达**具有等价类性质
  - 问题是在算法上我们怎么分解SCC？



SCC用途

- 社交网络里用强连通分支划分联系紧密的社区
- 经济学家用来对工作市场进行划分
- 可以把复杂的图简化，易于发现内在联系



如何找SCC

直观的思路

- 对每对顶点u，v，用dfs算法找u到v的路径，和v到u的路径
- 然后把**相互可达**的顶点聚类，即
  - **相互可达**的顶点看作是等价关系，按这个关系分类

- 这个算法的时间复杂度至少需要$O(n^3)$

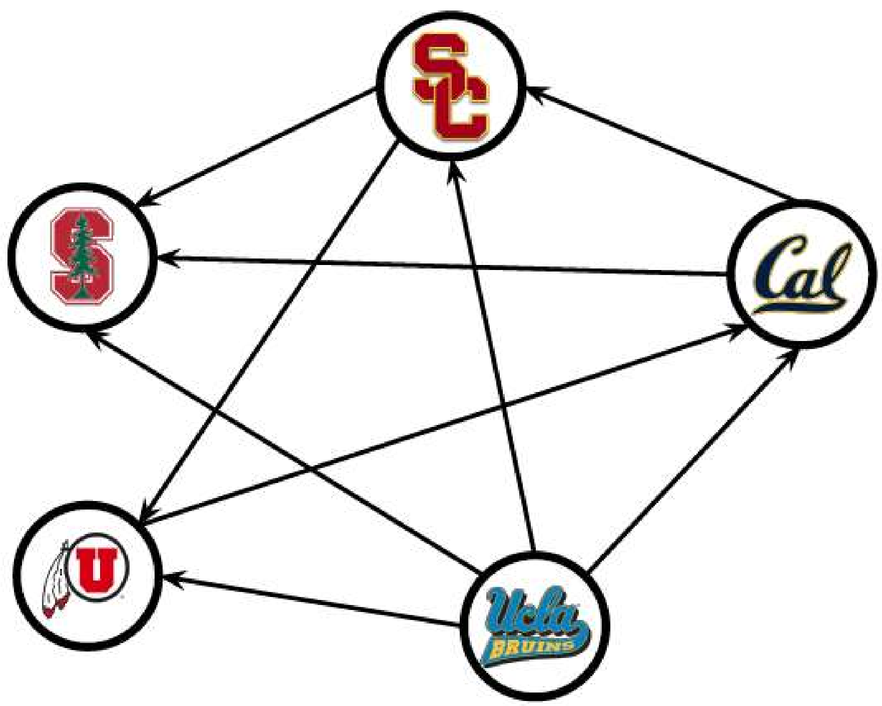
找SCC算法

- 能不能在线性时间内找到有向图的SCC

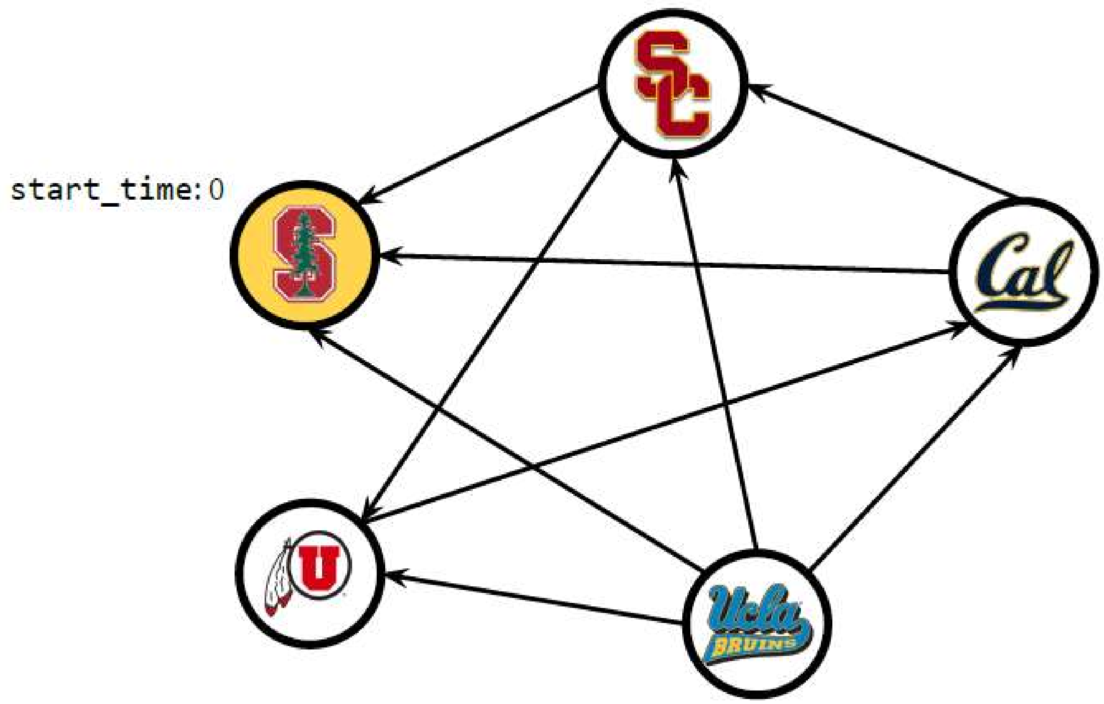
  O(|V|+|E|)

- **Kosaraju**算法
  1) 使用DFS算法+始末时间遍历所有顶点
     - 按起始的位置不同，可能会产生一个DFS森林
  2) 把图里的所有边反向
  3) 再运行一遍DFS算法，不过顺序是从第一次DFS算法中结束时间最晚的顶点作为起始点
     - 同样会产生一个DFS森林
  4) 第二遍DFS算法找到的不同DFS树就是强连通分支

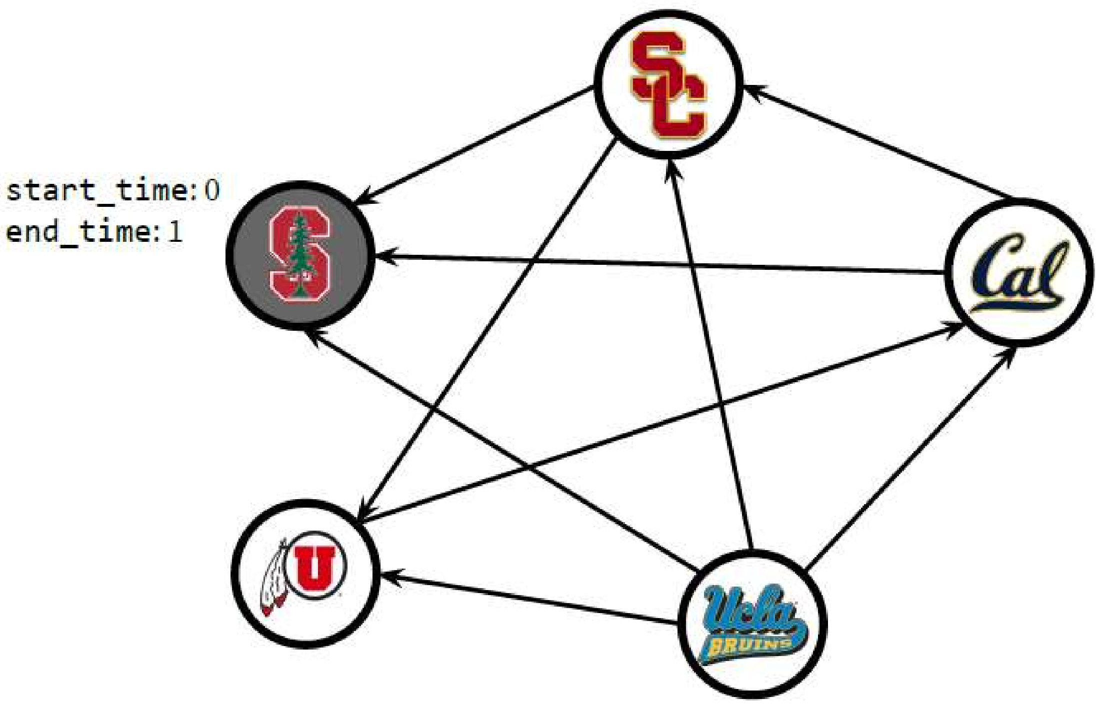
例子：找SCC



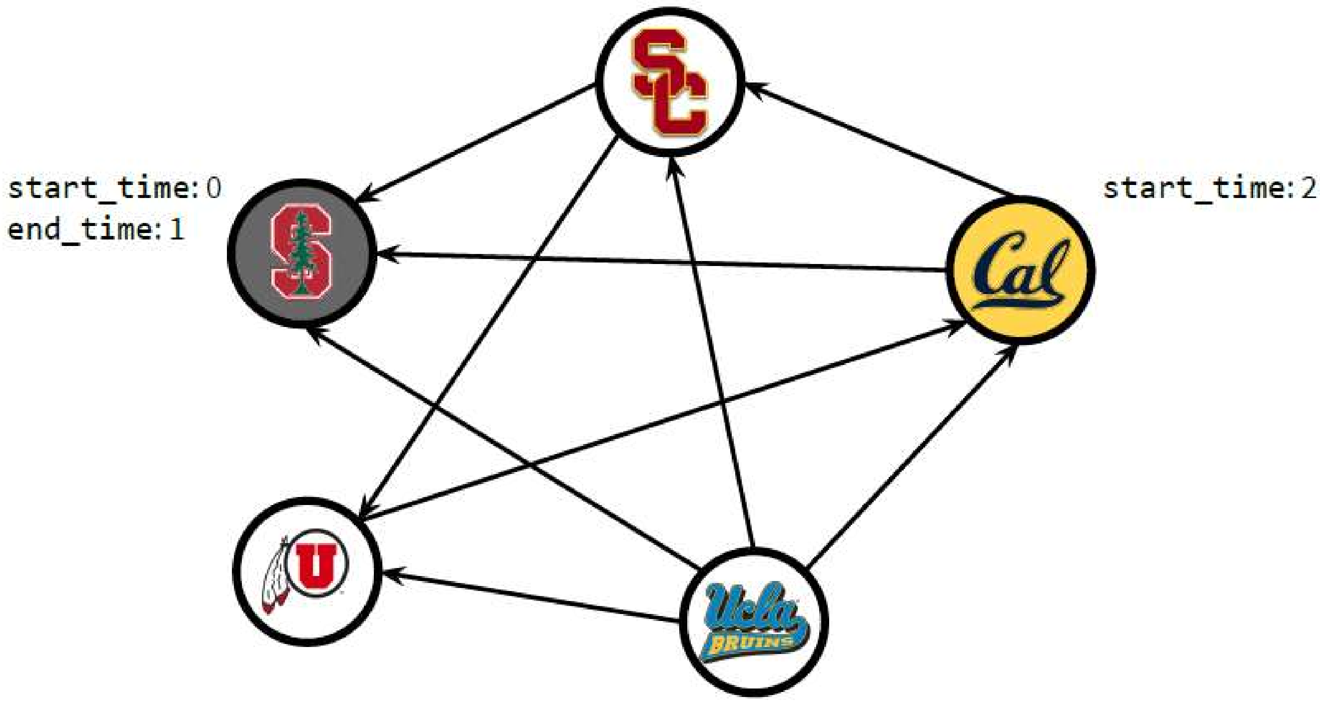
先从一个顶点开始



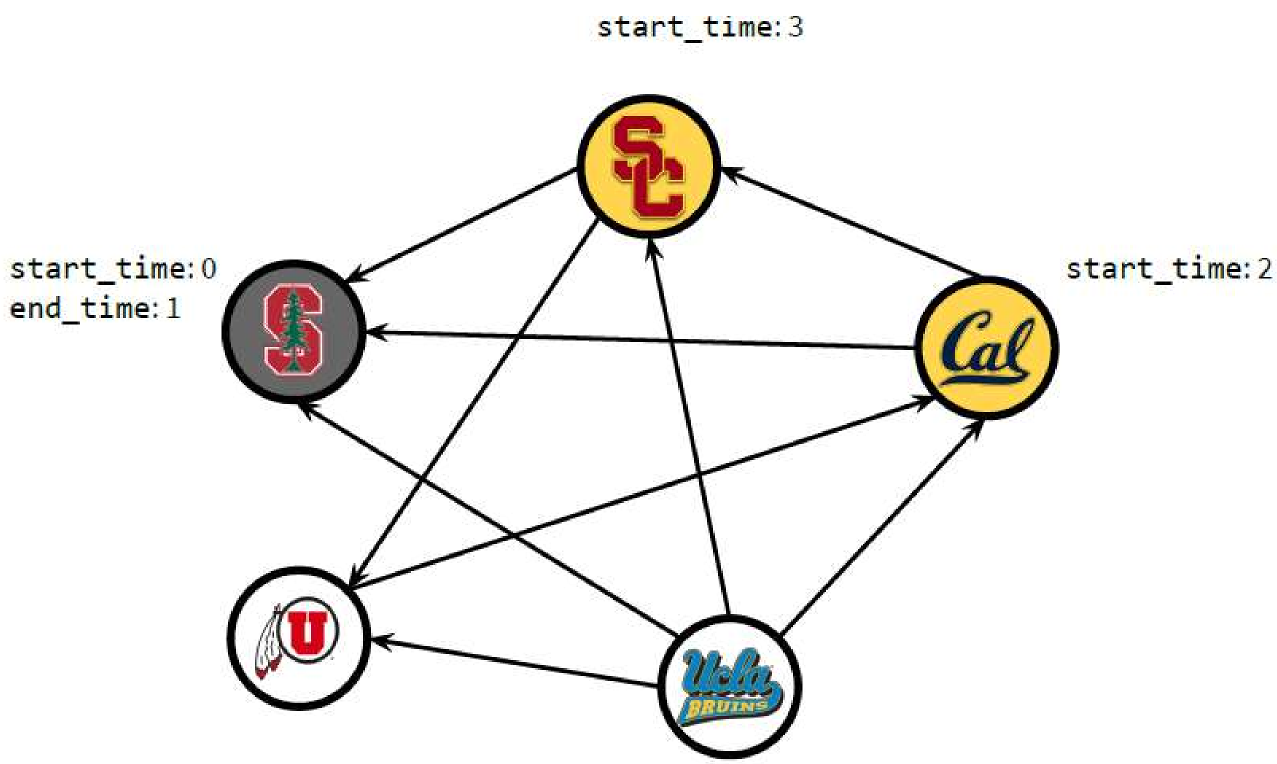
没有邻接顶点，所以完成第一次DFS



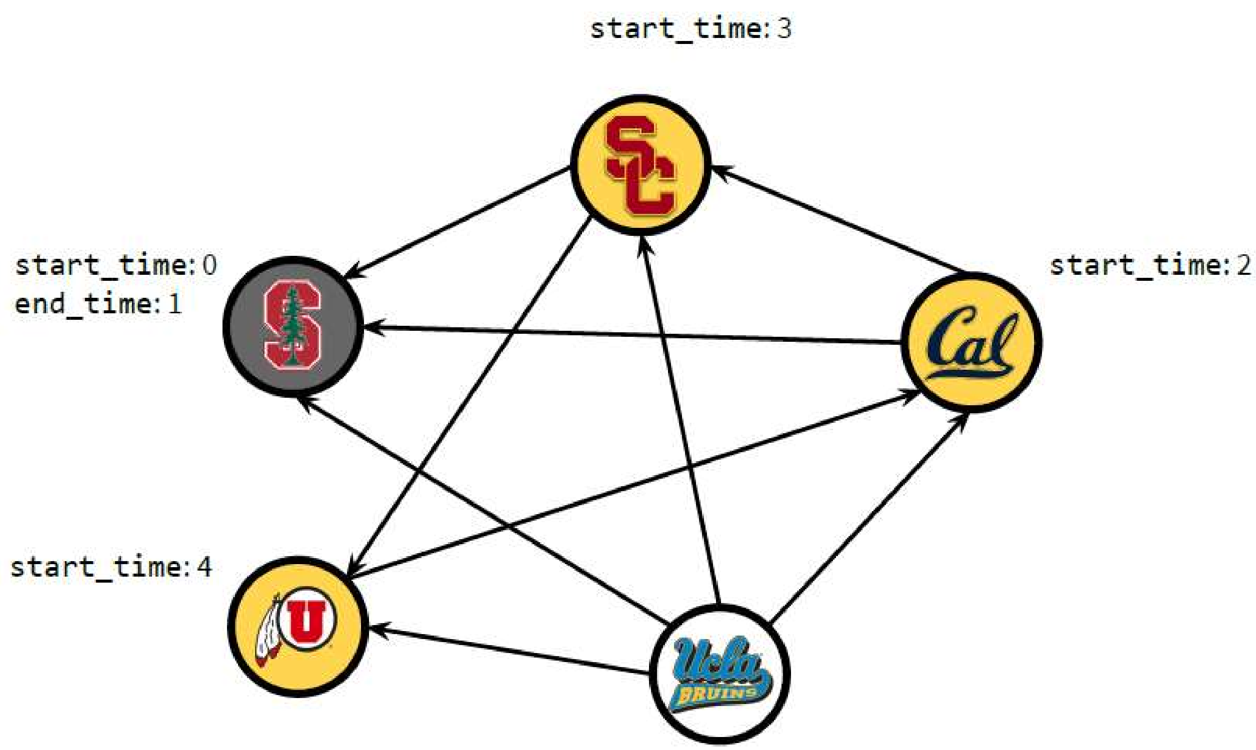
再从剩下未访问顶点选一个开始DFS



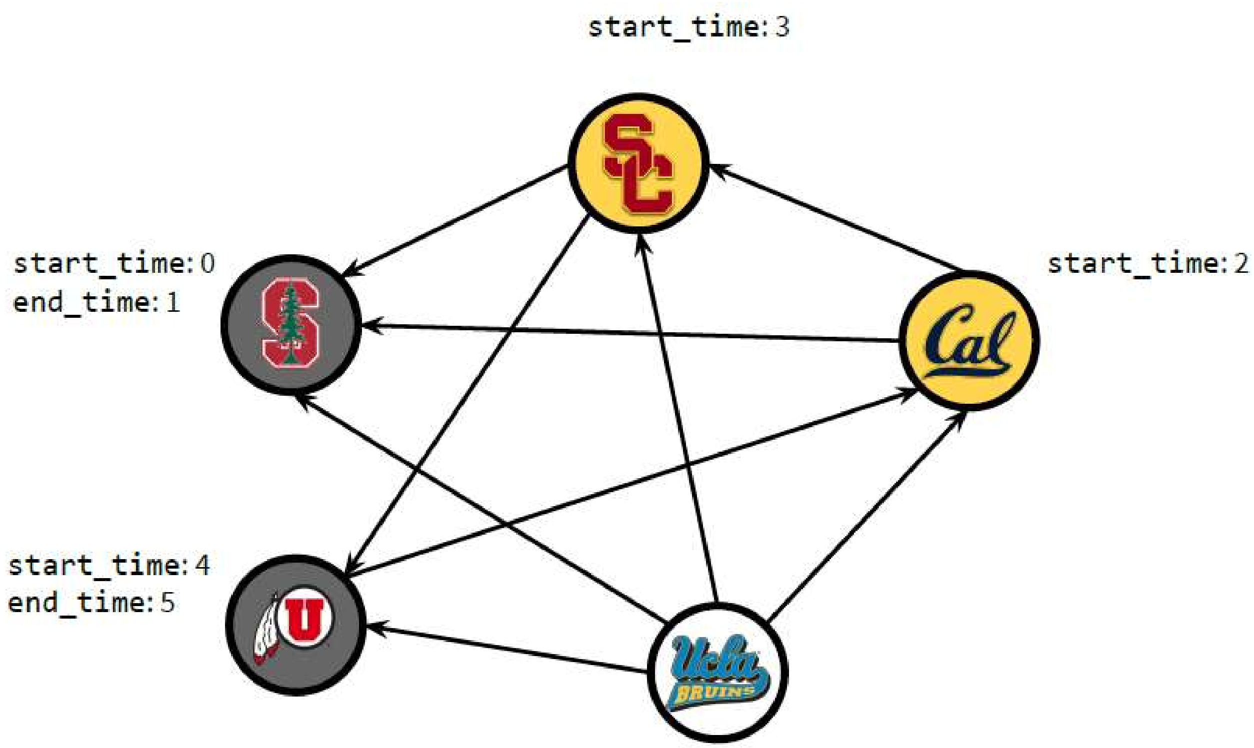
访问其中一个邻接顶点



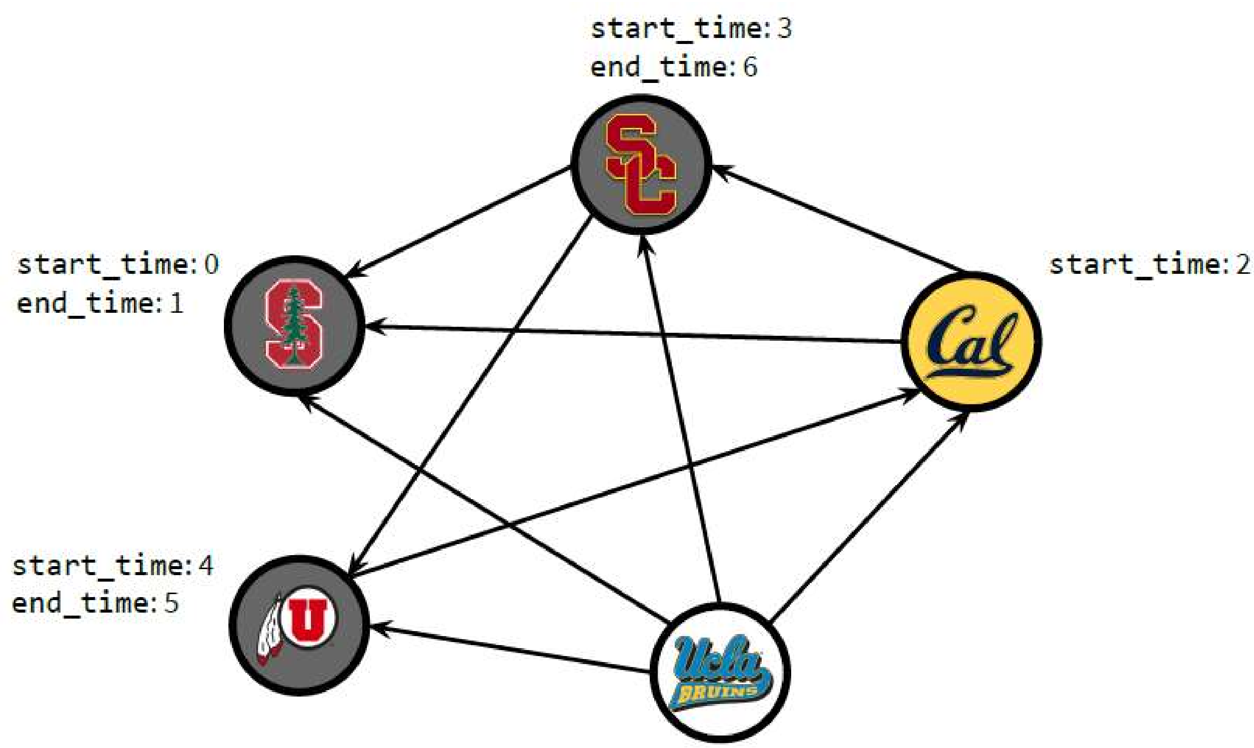
继续访问下一个邻接顶点



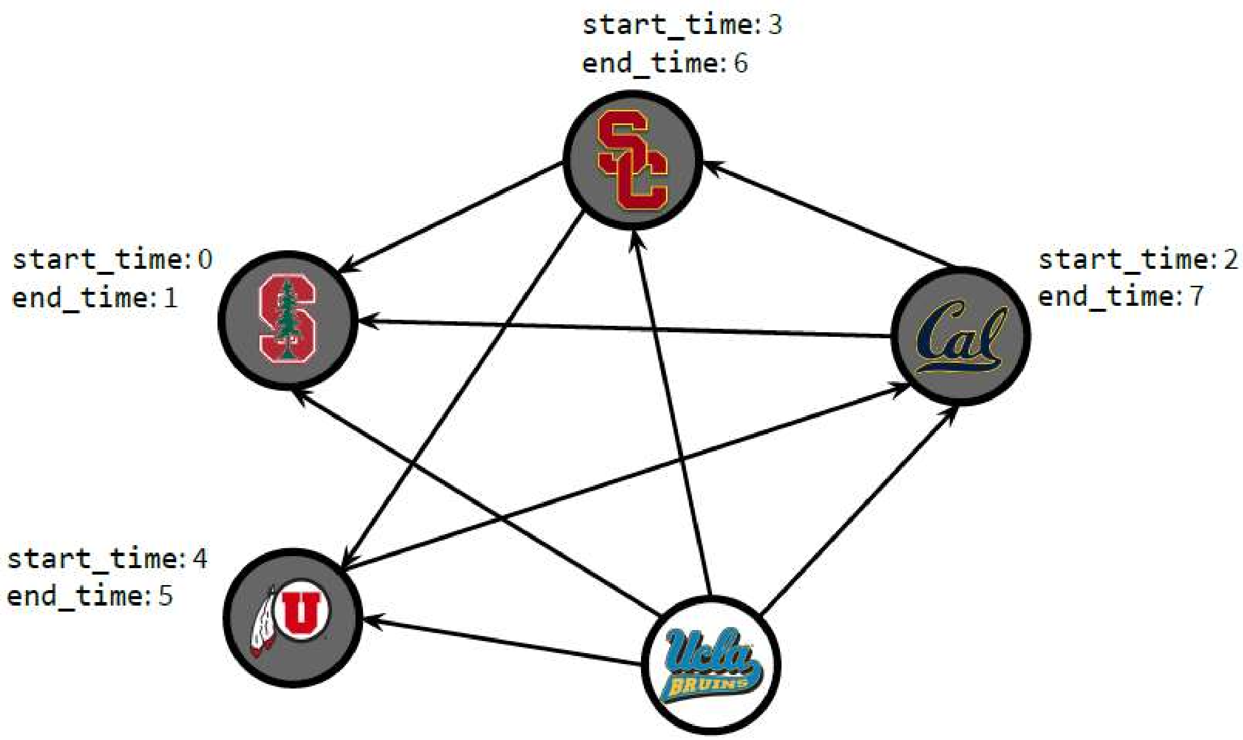
当前访问节点没有未访问邻接顶点



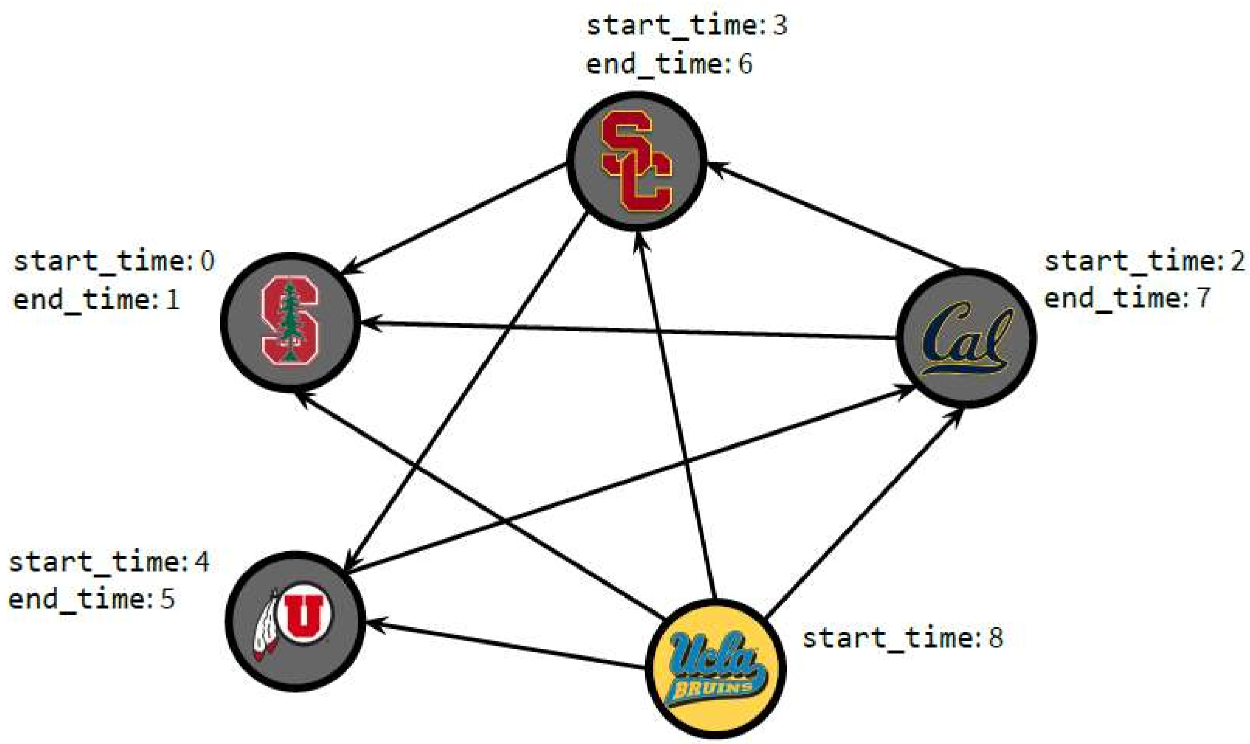
返回后，当前节点也没有未访问邻接顶点



返回后，结束这一次DFS

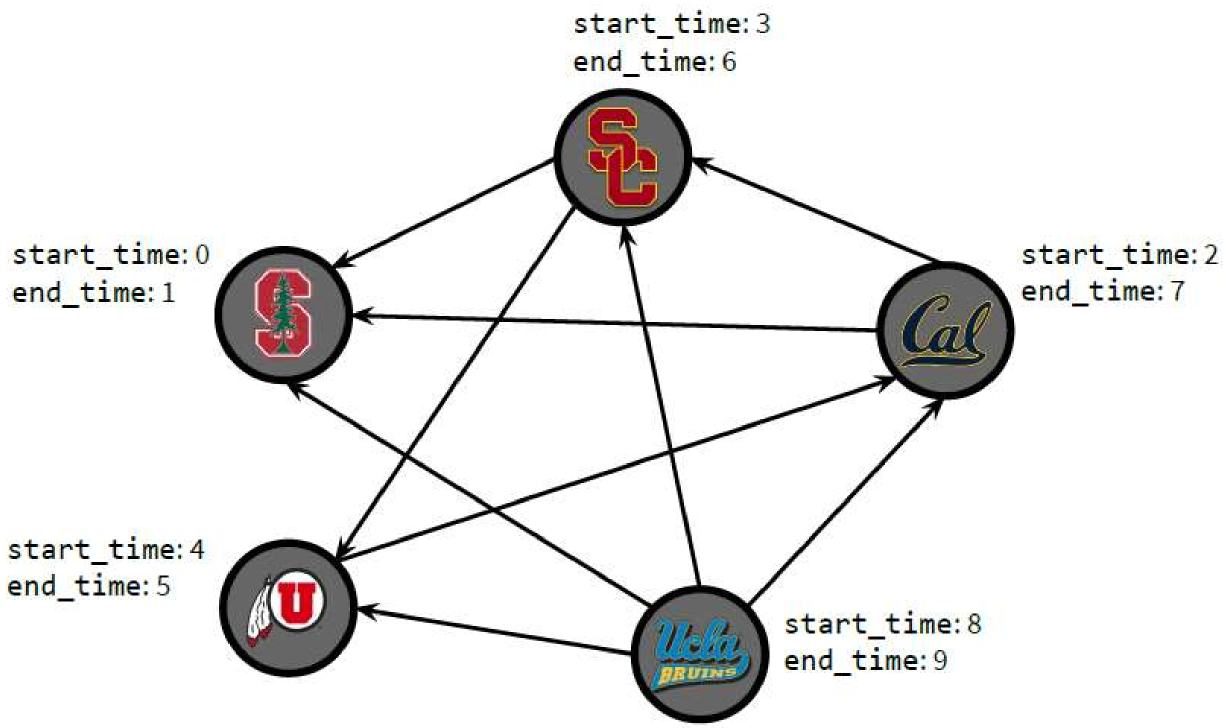


再开始一次DFS

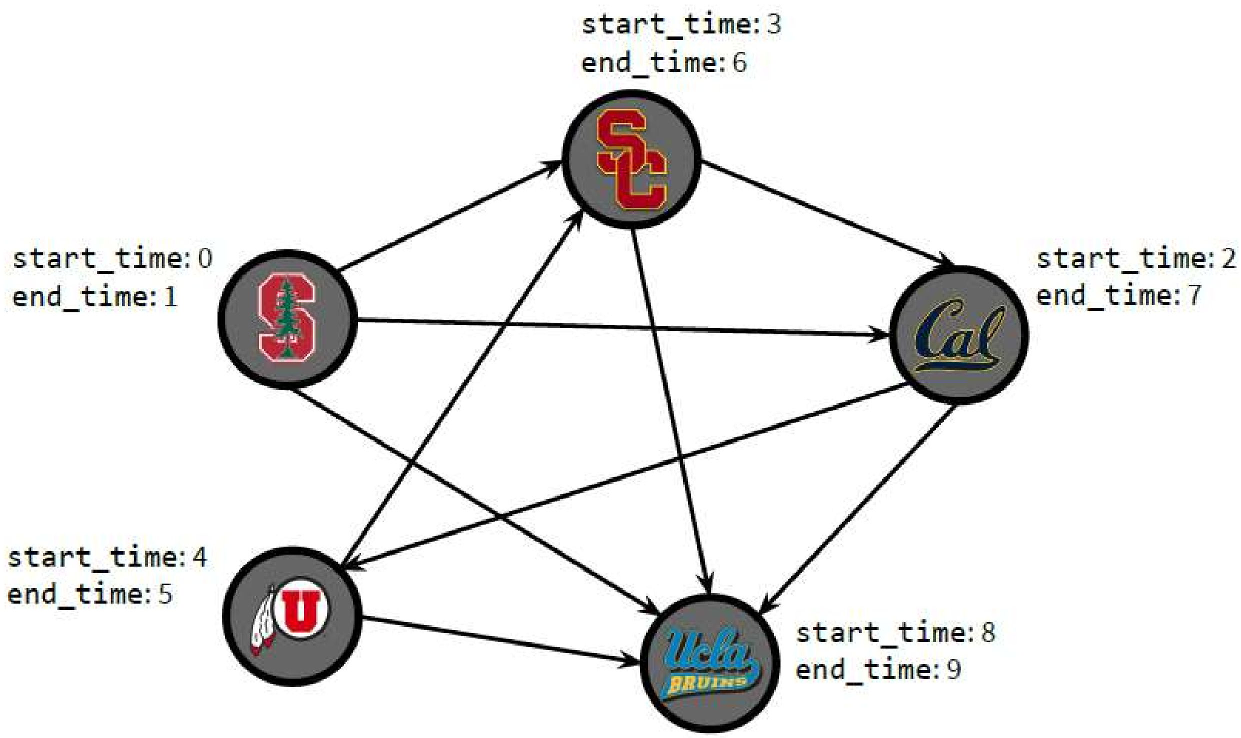


start_time: 3
end_time: 6
start_time: 0
end_time: 1
start_time: 2
end_time: 7
start_time: 4
end_time: 5
start_time: 8

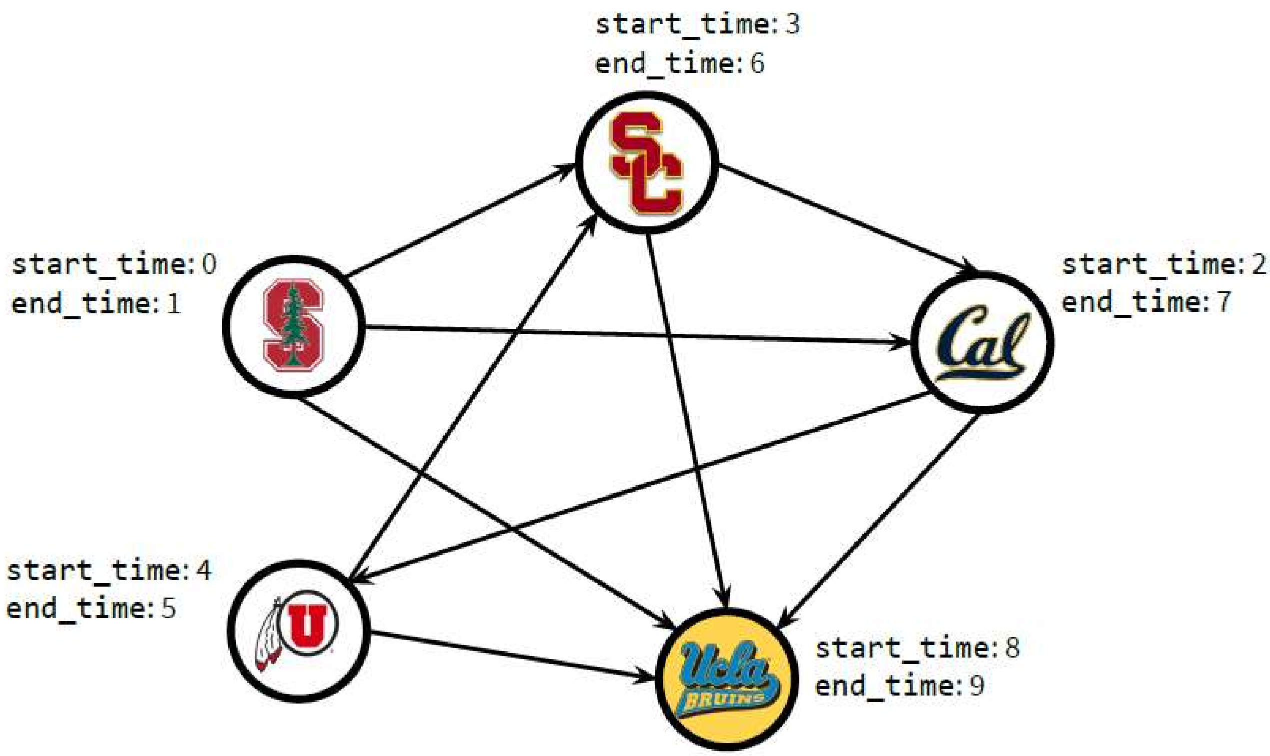
最后一个未访问的顶点结束



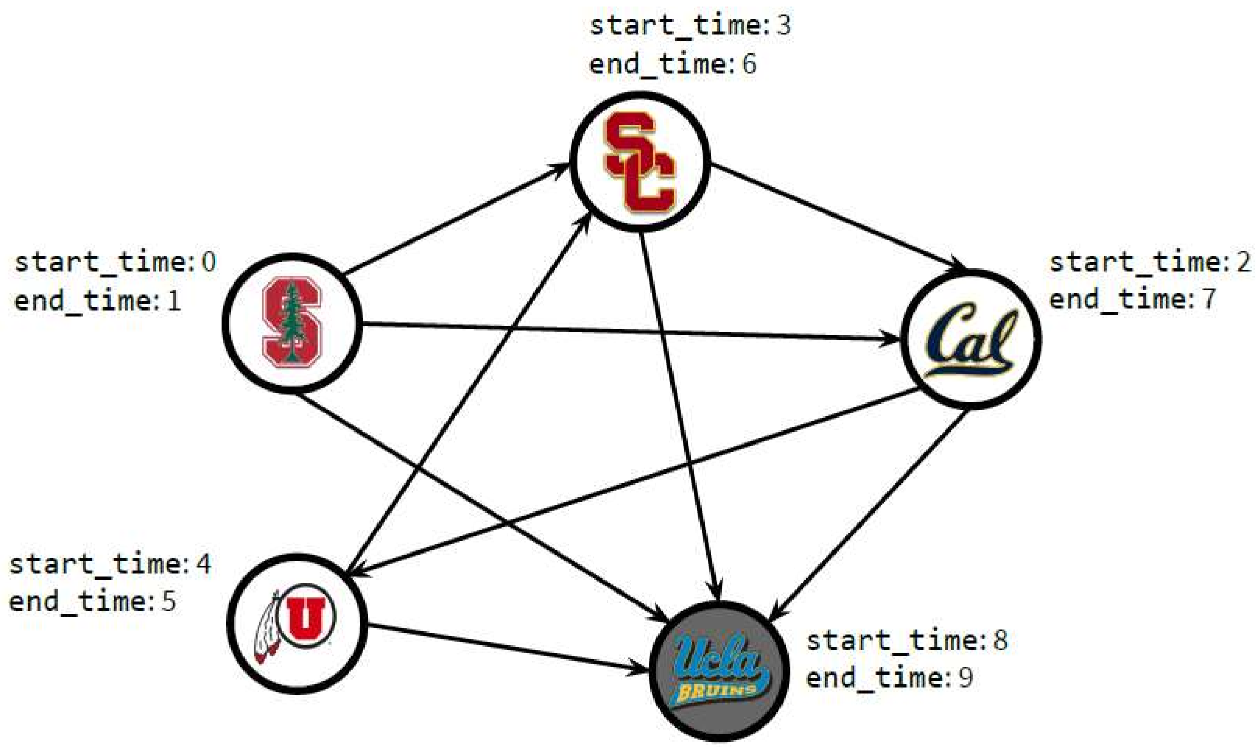
第二步，把所有边反向


start_time: 3
end_time: 6
start_time: 0
end_time: 1
start_time: 2
end_time: 7
start_time: 4
end_time: 5
start_time: 8
end_time: 9

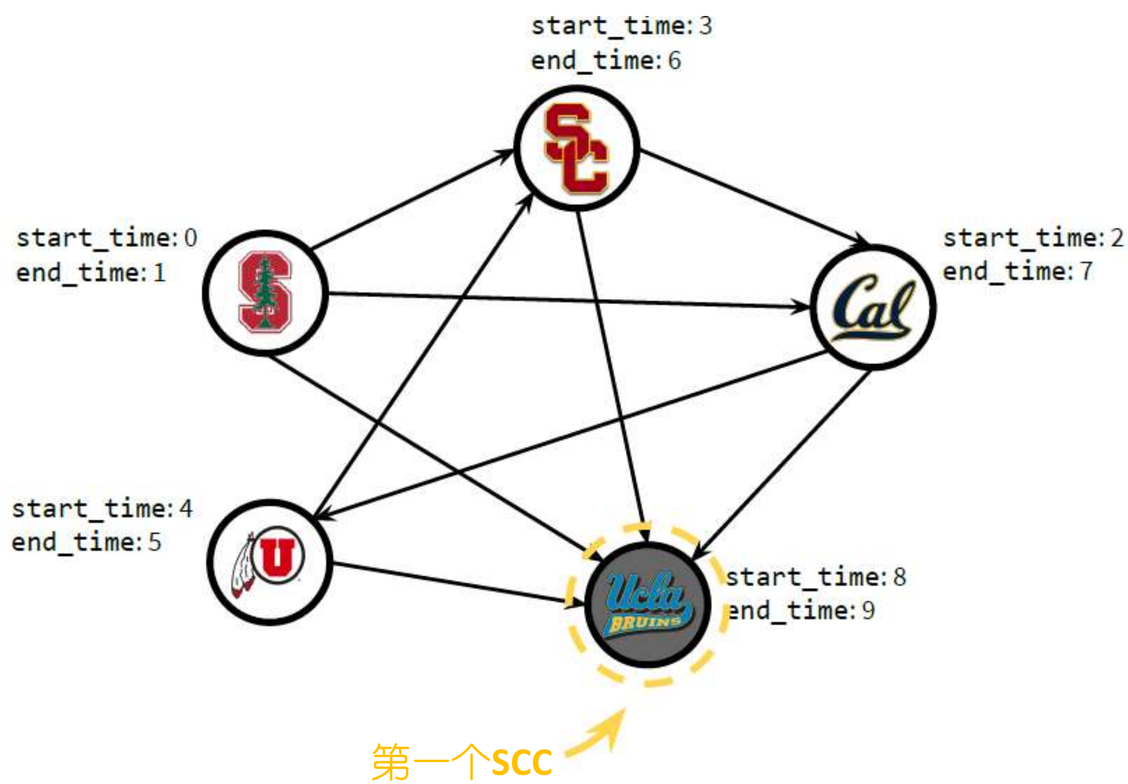
第三步，从结束时间最大的节点开始DFS



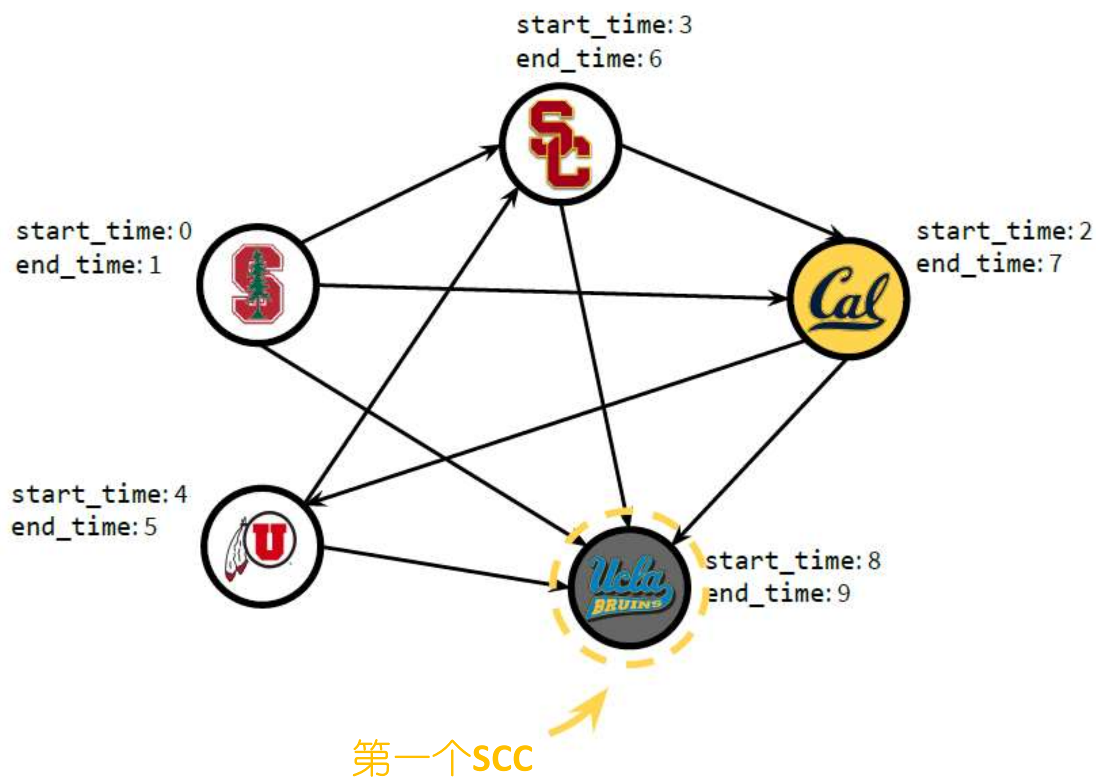
因为没有出边，所以结束访问



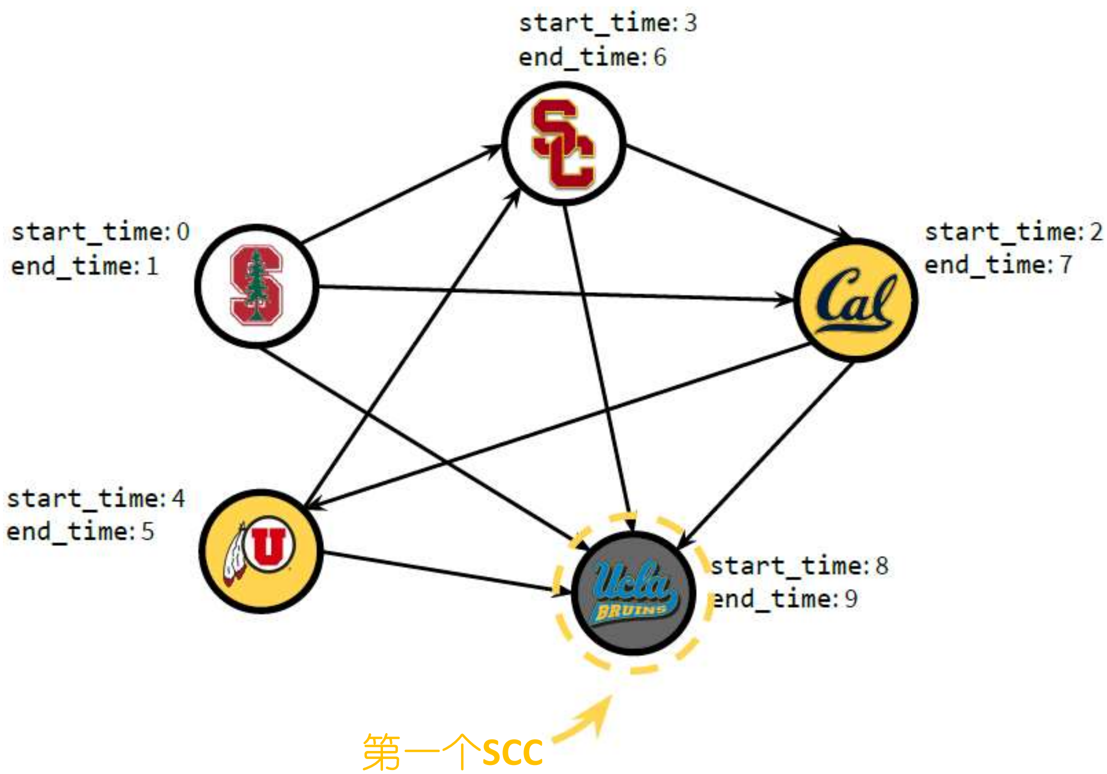
这个DFS树就是一个SCC



再选一个结束时间最大的顶点开始DFS

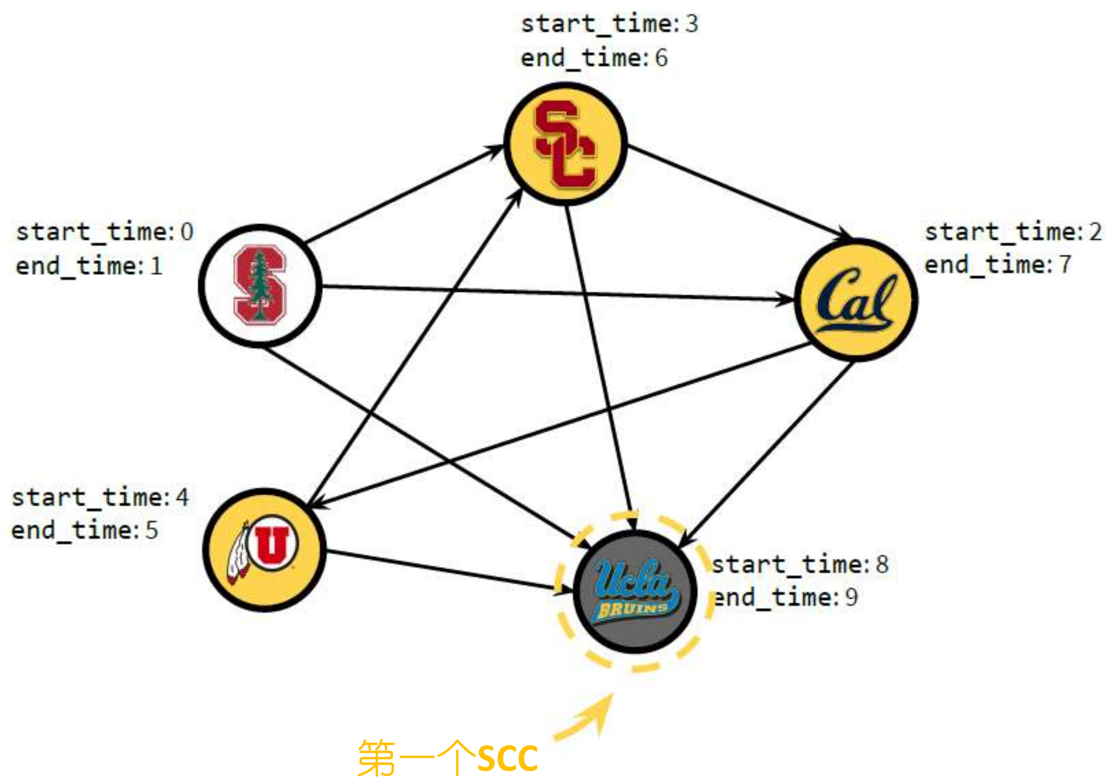


访问其中一个邻接顶点

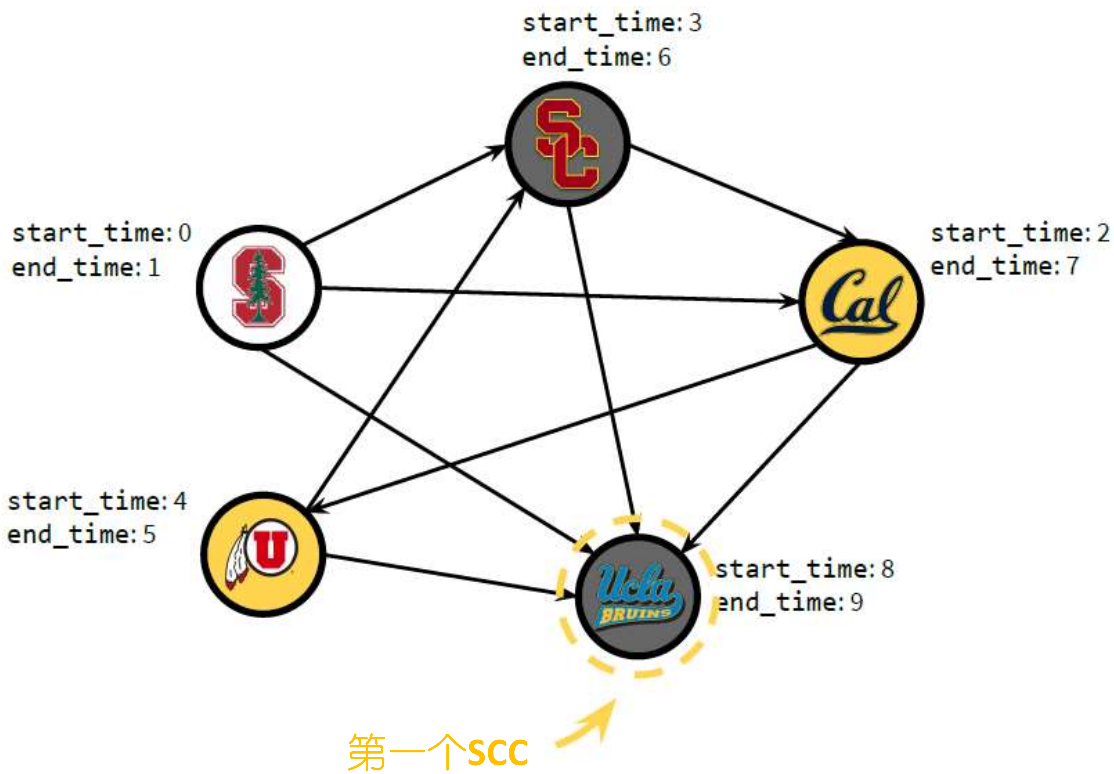


start_time: 3
end_time: 6
start_time: 0
end_time: 1
start_time: 2
end_time: 7
start_time: 4
end_time: 5
start_time: 8
end_time: 9
第一个SCC

继续访问邻接顶点

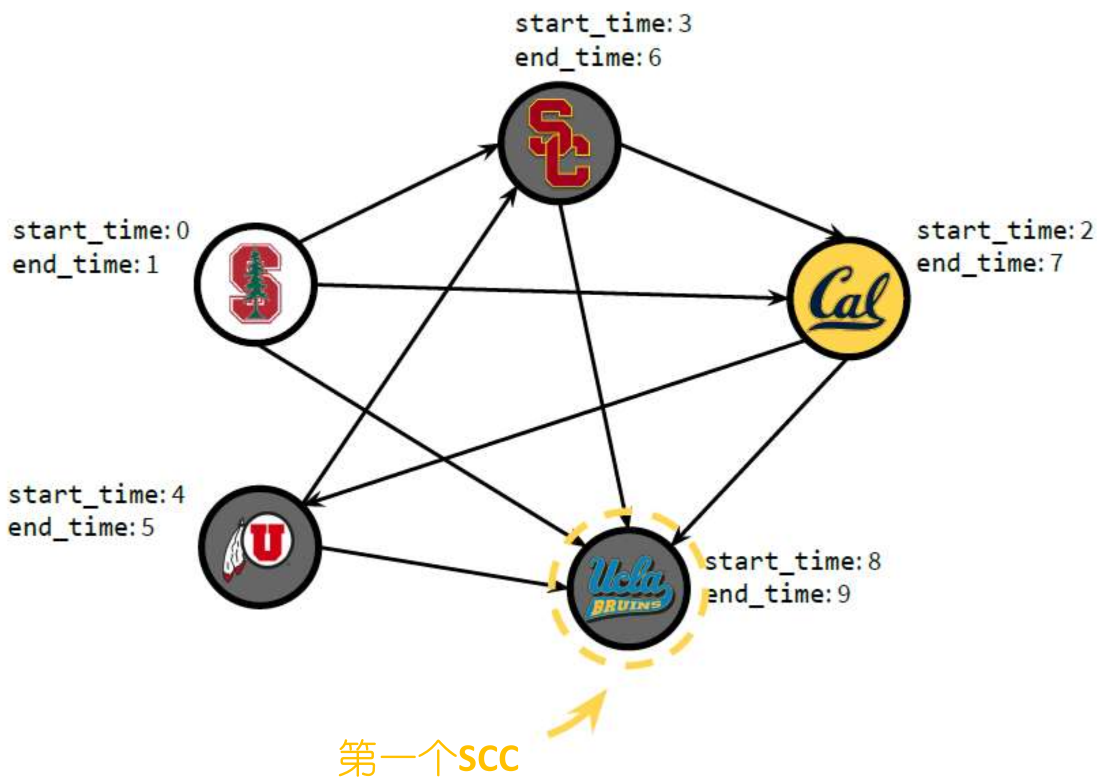

start_time: 3
end_time: 6
start_time: 0
end_time: 1
start_time: 2
end_time: 7
start_time: 4
end_time: 5
start_time: 8
end_time: 9
第一个SCC

当前顶点结束，返回

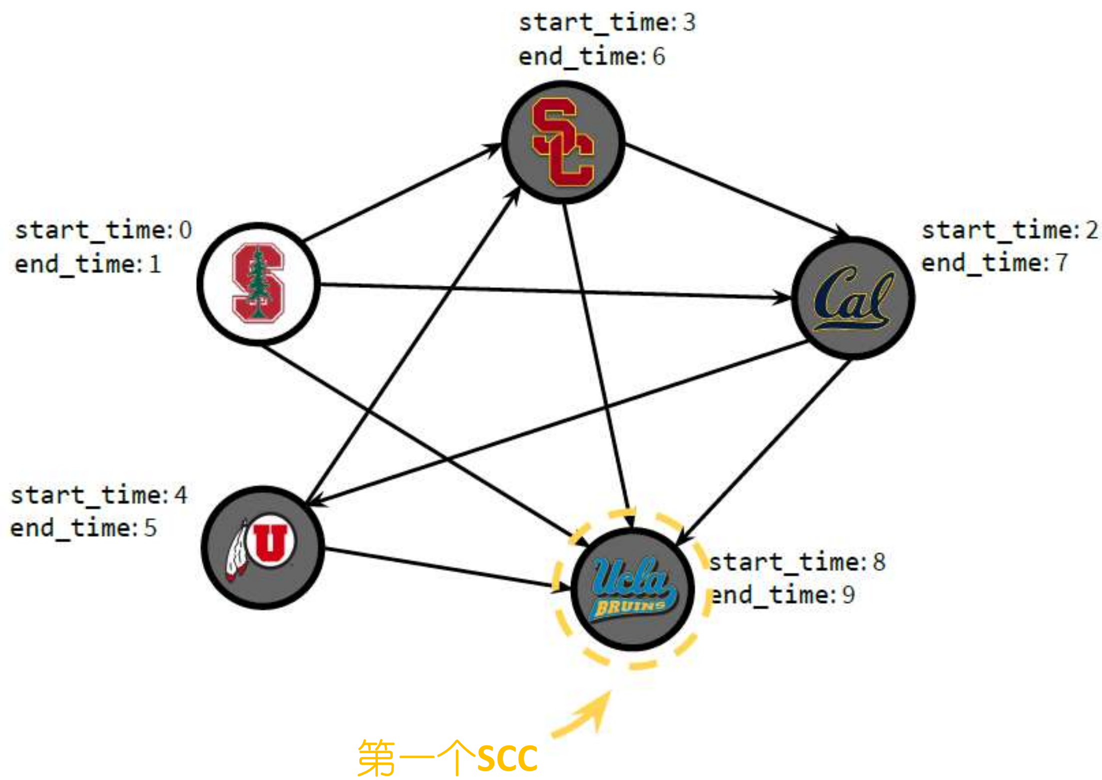


start_time: 3
end_time: 6
start_time: 0
end_time: 1
start_time: 2
end_time: 7
start_time: 4
end_time: 5
start_time: 8
end_time: 9
第一个SCC

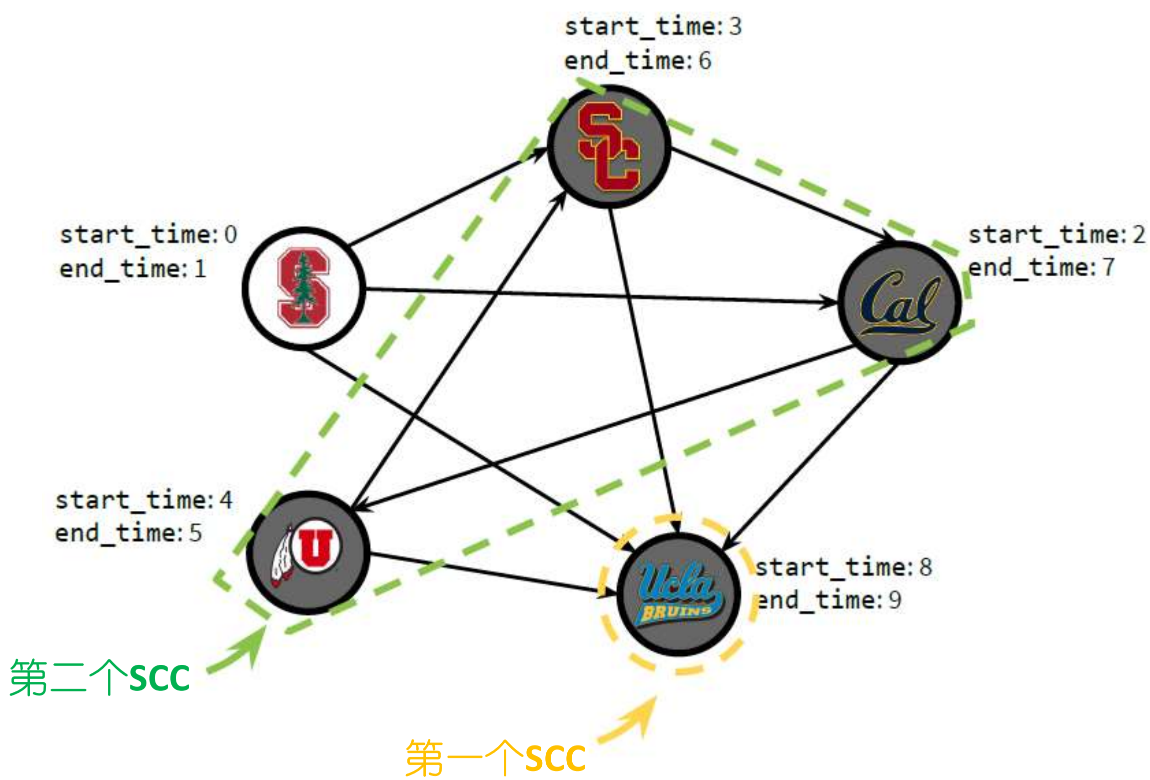
返回后，当前顶点也结束访问



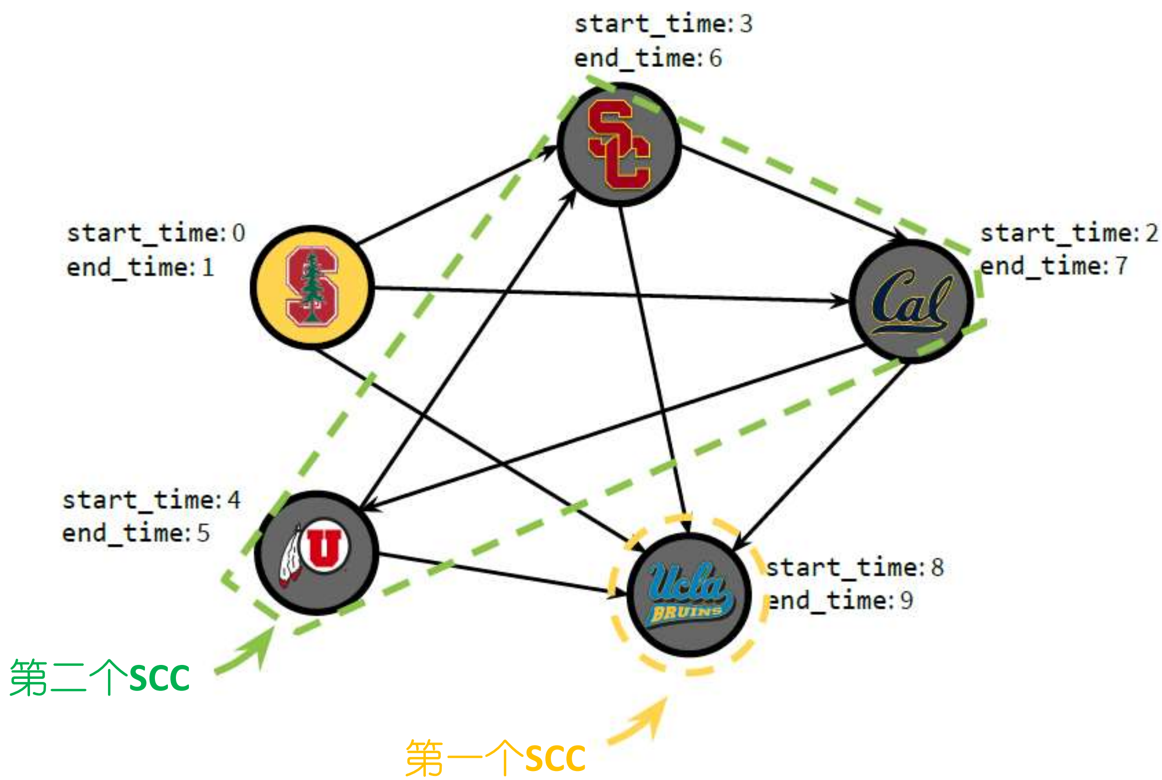
继续返回，第二次DFS结束



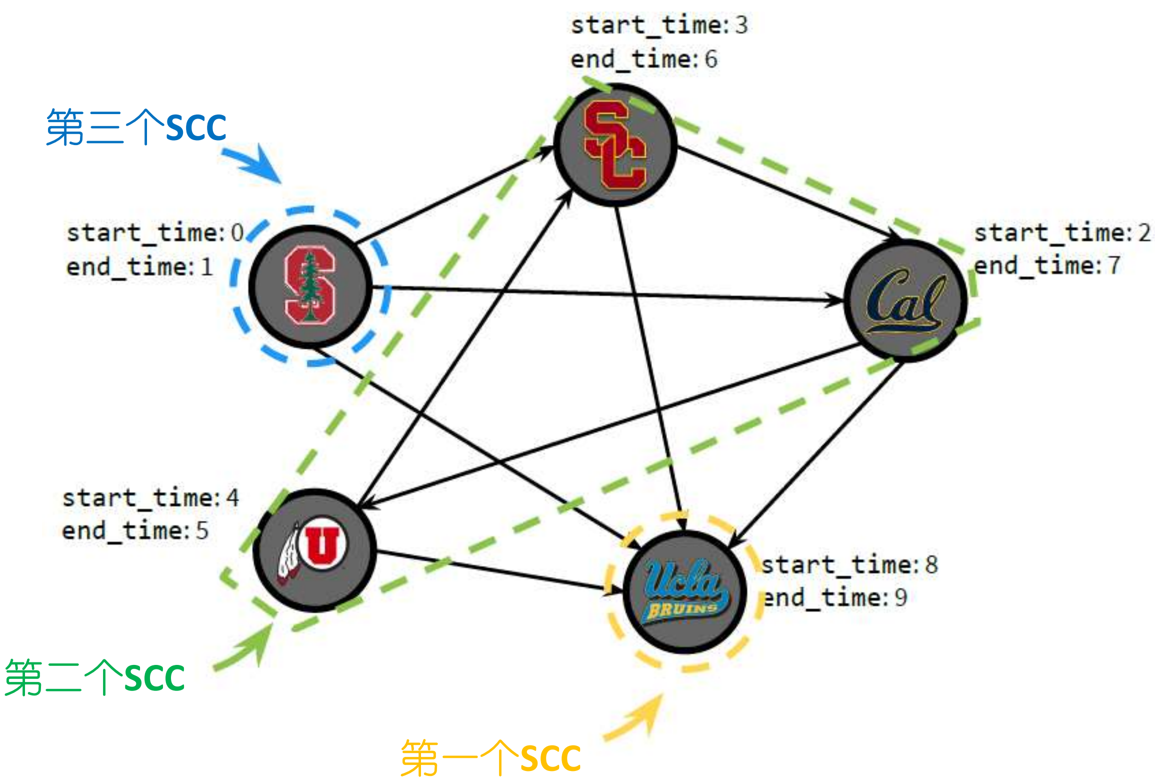
第二个DFS树又是一个SCC



只剩最后一个顶点，它也是一个DFS树



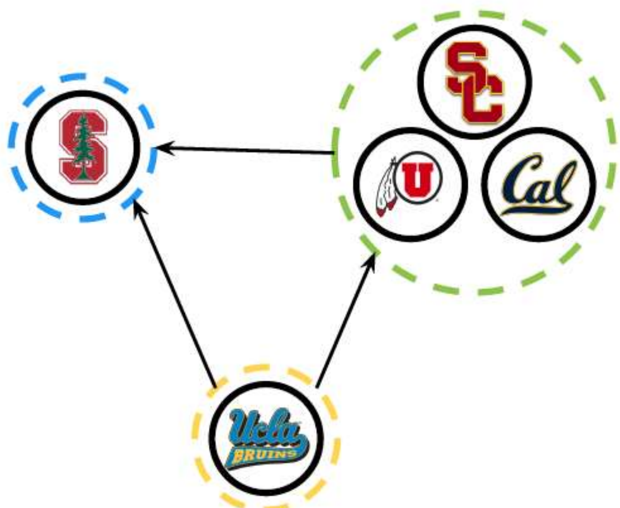
这样就找到三个SCC


start_time: 3
end_time: 6
第三个SCC
start_time: 0
end_time: 1
start_time: 2
end_time: 7
start_time: 4
end_time: 5
start_time: 8
end_time: 9
第二个SCC
第一个SCC

为什么正确？

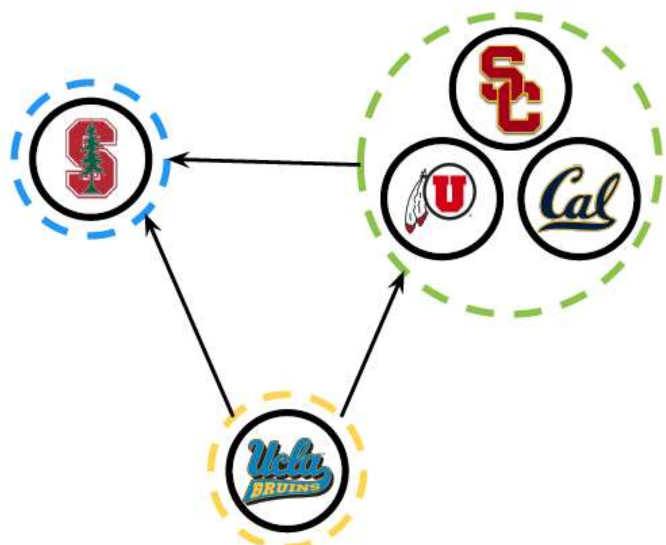
SCC图

- 把每个强连通分支塌陷成一个顶点
- 那么这样生成的图叫SCC图



那么，

- SCC图是一个有向无环图**DAG**
- 思路：如果不是**DAG**，那么有环，然后环路上的SCC节点可以合并强连通分支（相互可达）

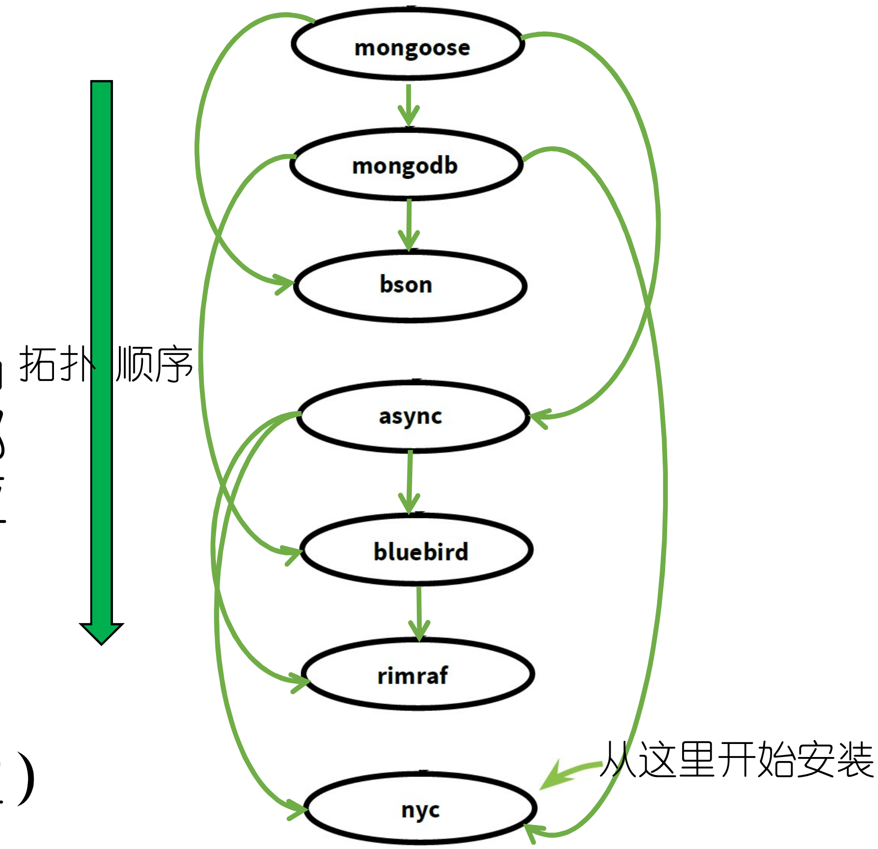


思路

- 回忆DAG可以拓扑排序

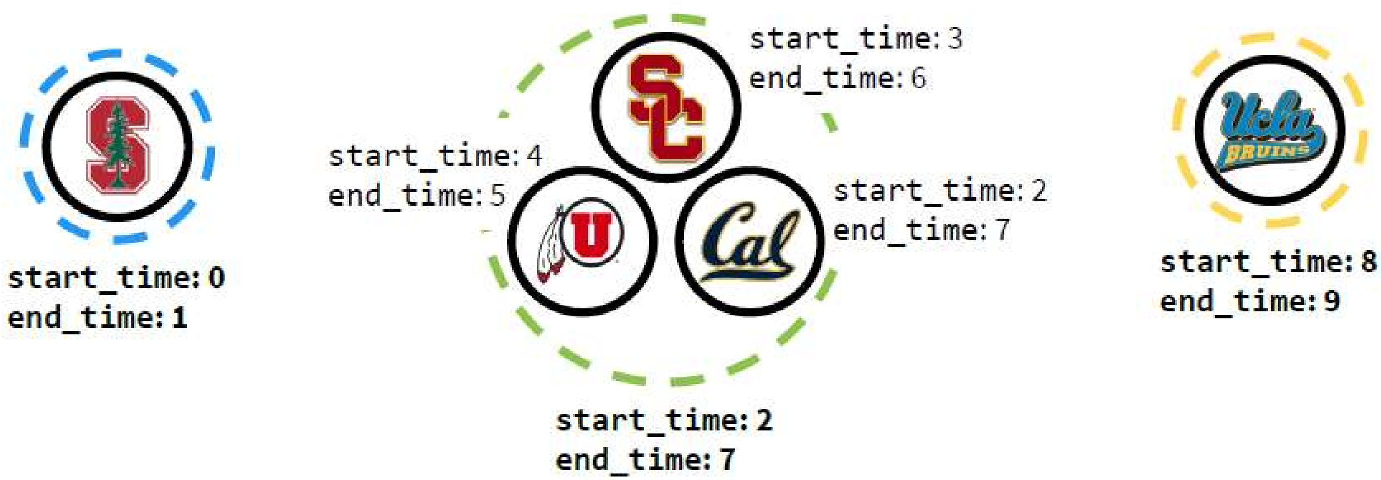
- 如果我们从最下面的SCC节点中选一个顶点开始做DFS搜索，那么得到的第一个DFS树就是相应的SCC节点

- **问题**是我们不知道哪个顶点属于最下面的SCC节点（只有入边）
  - 不是结束时间最早的顶点



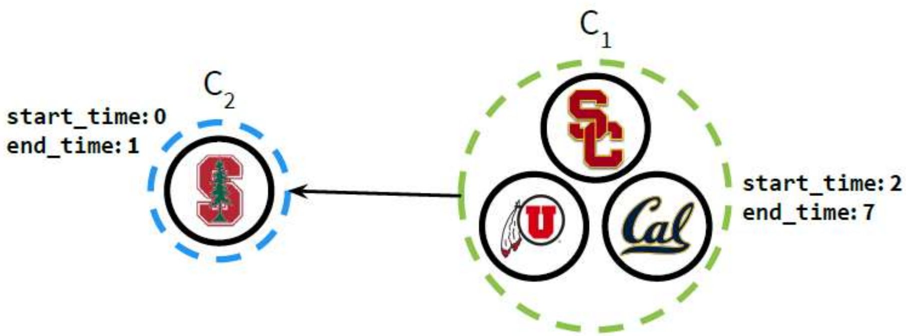
SCC的开始时间和结束时间

- 定义SCC的开始时间是其中节点开始时间的最小值
- 定义SCC的结束时间是其中节点结束时间的最大值



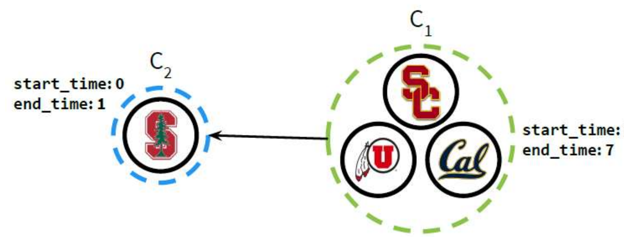
一个事实

- 如果强连通分支C_1到C_2有边，那么C_1的结束时间一定大于C_2
  - 在DAG中，这个结论我们已经证明了
  - 现在需要证明同样适用SCC



证明

- 情形1： 如果先访问C_1
  - 假设U是C_1中第一个访问的顶点
  - 假设X是C_1中结束时间最大的顶点
  - 假设Y是C_2中结束时间最大的顶点

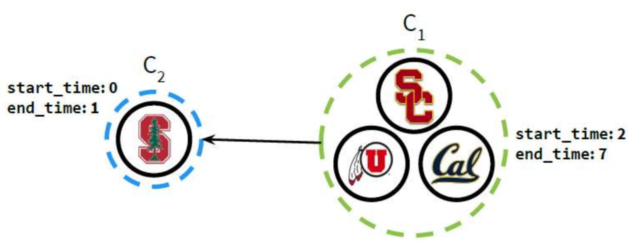


- 那么Y一定在U的DFS子树里，因为U和Y是可达的

- 所以C_1的结束时间一定大于C_2

证明

- 情形2：如果先访问C_2
  - 那么在C_2全部访问结束之后也到不了C_1
  - 因为SCC图是DAG，没有环路
  - 只有再重新开始新的DFS，才会访问C_1

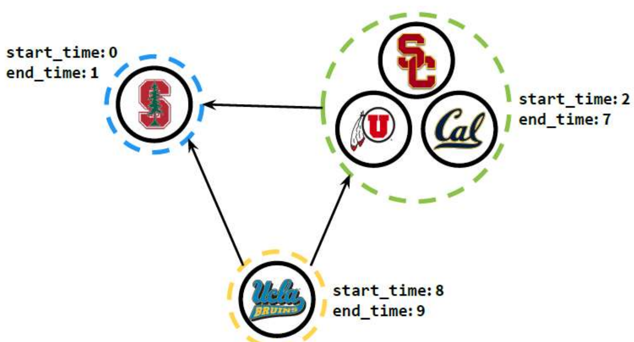


- 所以C_1的结束时间一定大于C_2
- 结束时间最大的SCC只有出边

主要的思路是

- 第一次DFS后，有了所有顶点访问的始末时间
- 希望找到一个起点，再运行DFS后，相应的DFS树就是一个SCC

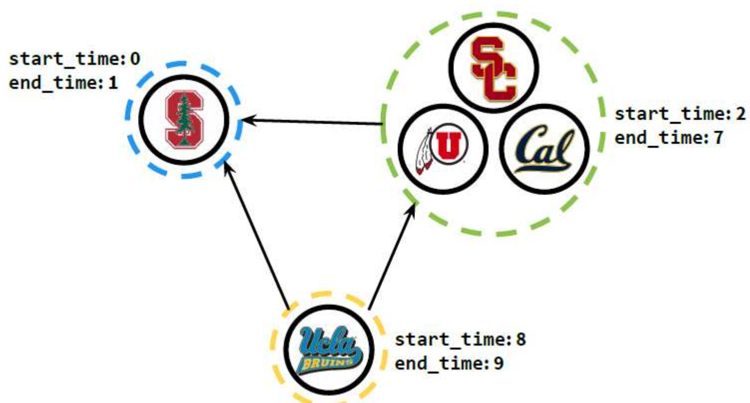
- 观察：结束时间最大的SCC没有入边，
  结束时间最大的顶点一定在这个SCC里面



所以如果反转方向，从结束时间最大的顶点开始DFS，就能达到目的

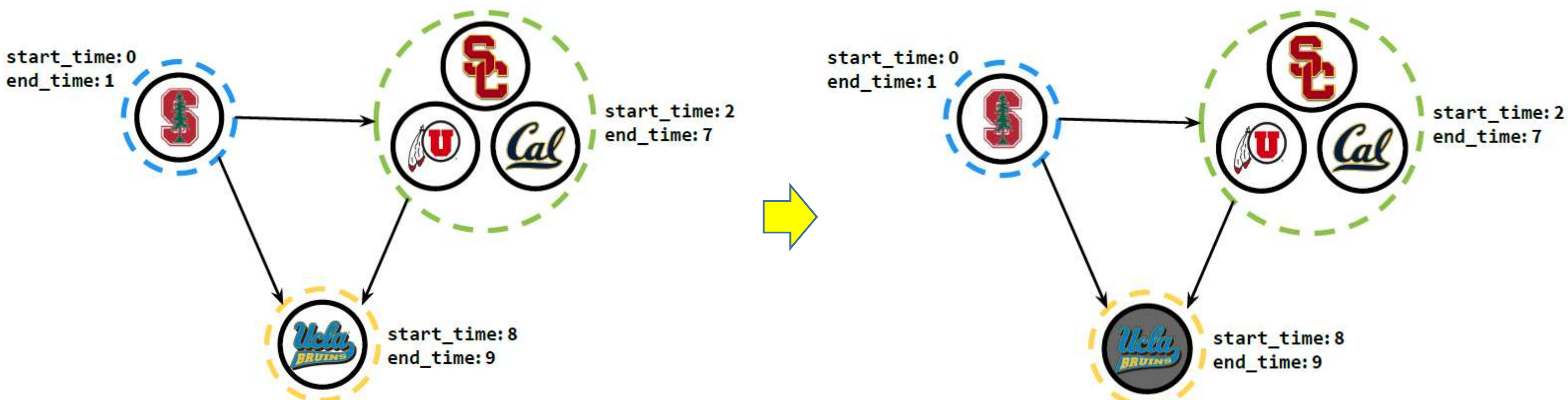
如果不反转方向，先找结束时间最小SCC？

- 注意：虽然结束时间最小的SCC没有出边
- 但是我们不知道哪个顶点在这个SCC里面
- 因为结束时间最小的顶点不见得在这个SCC里面



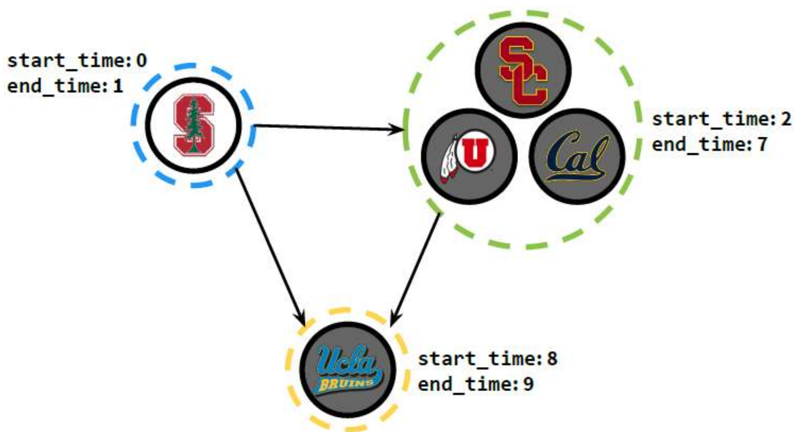
所以必须找结束时间最大的SCC

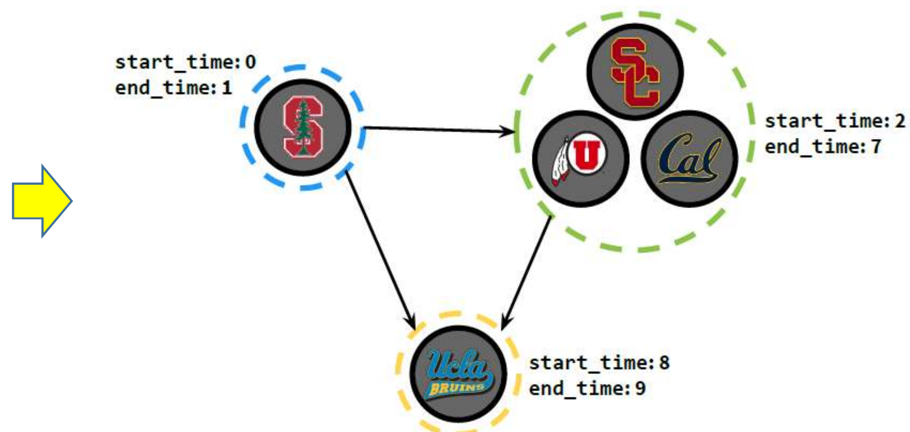
- 先反转图中边的方向，使得结束时间最大的SCC只有入边
- 那么从结束时间最大的顶点开始DFS，得到的DFS树就是一个SCC



然后

- 忽略找到的SCC中的顶点，剩下的图还是满足之前的性质
- 在剩下的顶点中找结束时间最大的顶点，继续DFS
- 直到所有SCC都找到





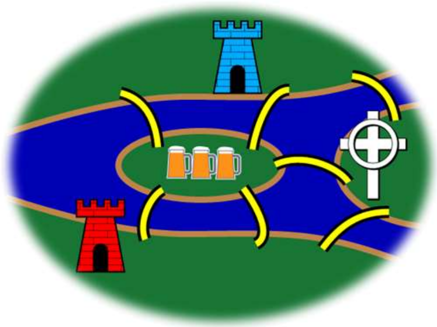
小结

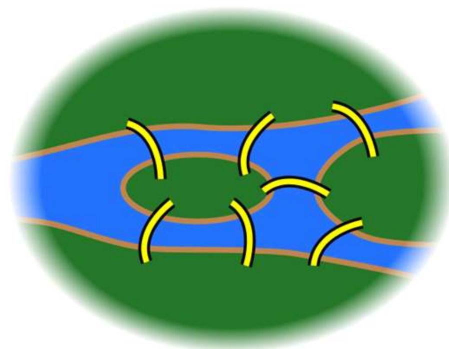
- **Kosaraju**算法的时间复杂度是 O(|V|+|E|)
  - 第一遍DFS O(|V|+|E|)
  - 反转边方向 O(|V|+|E|)
  - 第二遍DFS O(|V|+|E|)

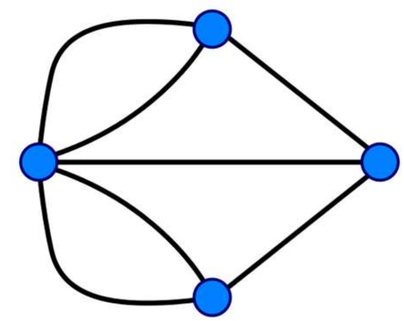
- 还有别的在有向图中找强连通分支的线性算法
  - Tarjan算法
  - Gabow算法

Euler Tour

- 给定一个有向（无向）图，$G(V,E)$
  - 访问每条边正好一次的回路就是欧拉回路
  - 七桥问题
  - 一笔画



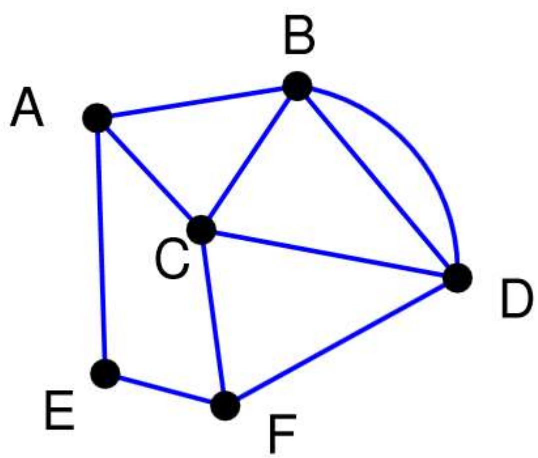


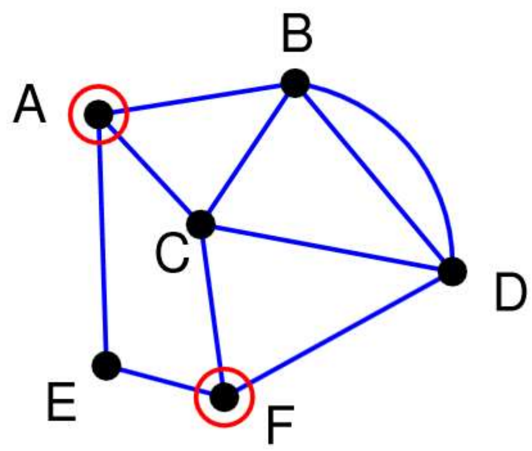


  - 解法：验证图连通，且degree为奇数的顶点个数是0或者2

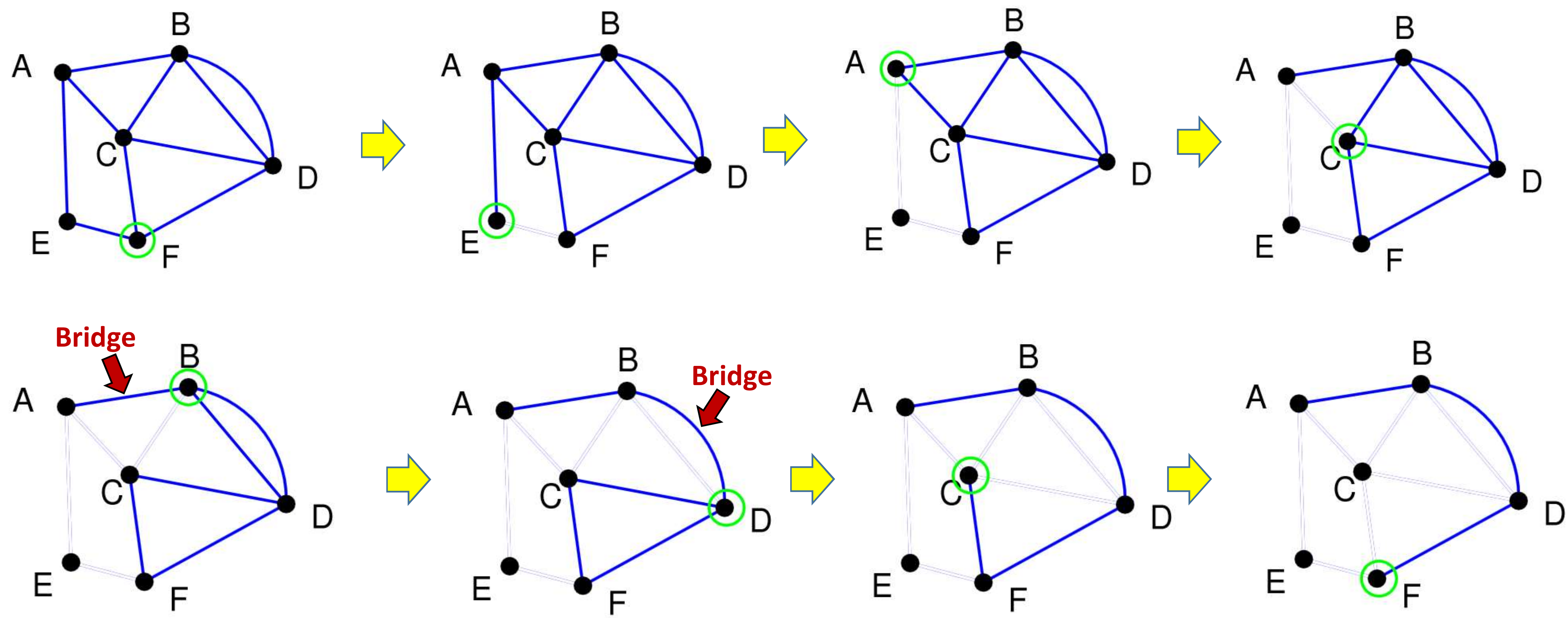
Euler Tour

- **Fleury's algorithm**
  - Removing a single edge from a connected graph can make it disconnected. Such an edge is called a bridge.
  - “Don’t burn your bridges.”





Euler Tour


A B C D E F
A B C D E F
A B C D E F
A B C D E F
Bridge
A B C D E F
Bridge
A B C D E F
A B C D E F
A B C D E F

Fleury's algorithm

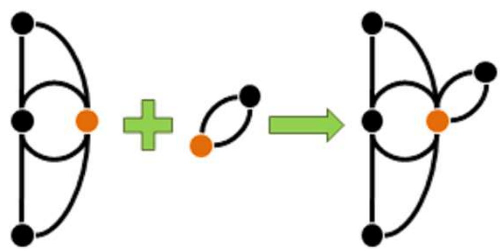
- 确定图中是否存在欧拉回路
  - 如果没有度为奇数的顶点，可以任选一个顶点开始
  - 如果有2个度为奇数的顶点，从中选一个开始
- 每次选一条没走过的边走
  - 如果有**不是Bridge**的边，那一定走它
- 直到所有边都走完

- 时间复杂度($O(E^2)$)

Hierholzer's algorithm

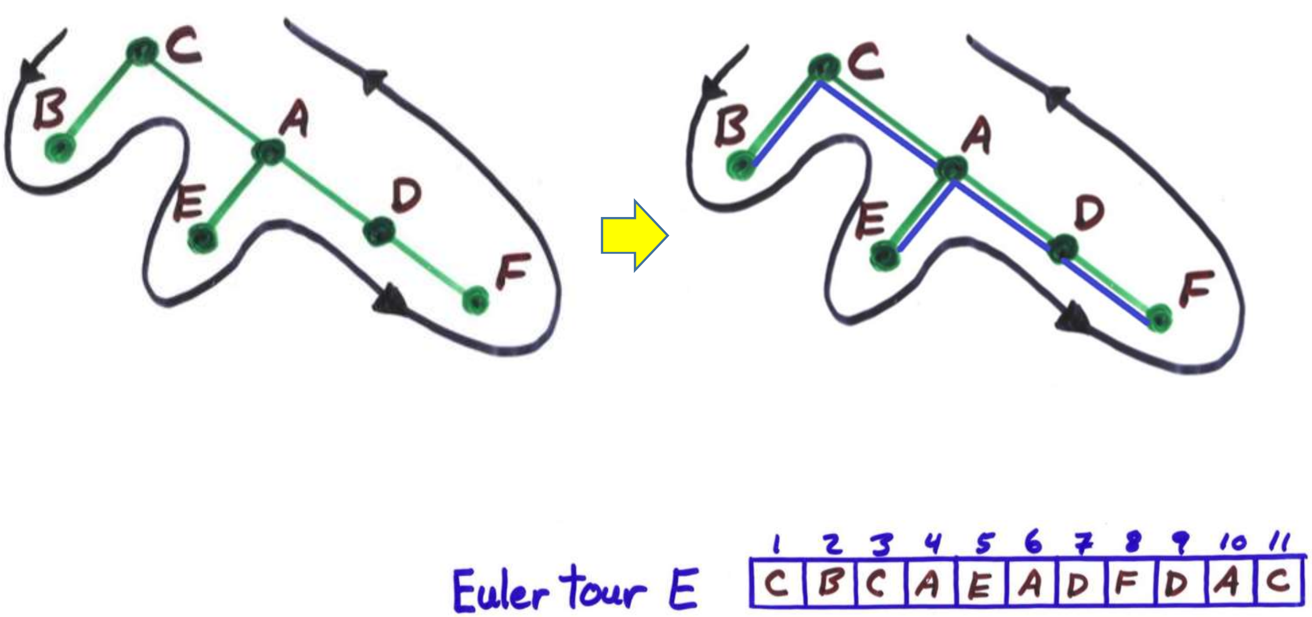
- `Hierholzer's algorithm` $(O(E))$
- 一个Euler Tour，在某点相交，可拆成两个Euler Tour

- 反之，两个Euler Tour，在某点相接，可合成一個 Euler Tour



- 算法就是从任意一点出发，先走成一个Euler Tour
- 然后在剩下的边里继续找Euler Tour，和之前的合并

树的欧拉回路




```
void EulerTour(const Node *root)
{
    if(root == nullptr)
    {
        return;
    }

    cout<< root->key << " ";

    EulerTour(root->left);
    if (root->left)
    {
        cout<< root->key << " ";
    }

    EulerTour(root->right);
    if (root->right)
    {
        cout<< root->key << " ";
    }
}
```

Q&A

Thanks!