数据结构与算法

DATA STRUCTURE

第二十三讲 强连通分支

胡浩栋

信息管理与工程学院

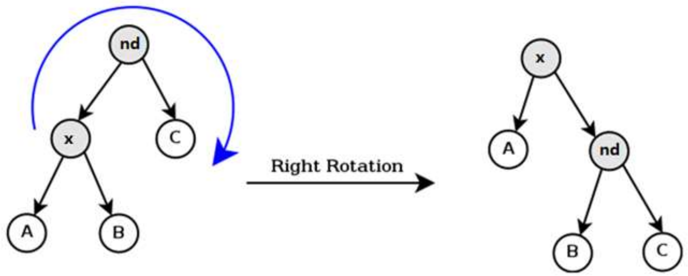
2017 - 2018 第一学期

课堂内容

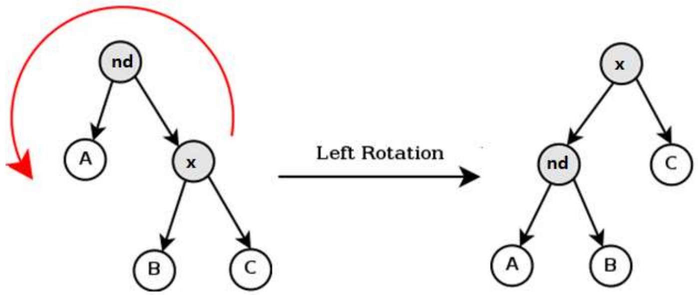
- 作业回顾
- 强连通分支

旋转

```
Node* AvlTree::RotateR(Node *nd)
{
    Node *x = nd->left;
    nd->left = x->right;
    x->right = nd;

    nd->height = max(getHeight(nd->left), getHeight(nd->right)) + 1;
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

    return x;
}
```



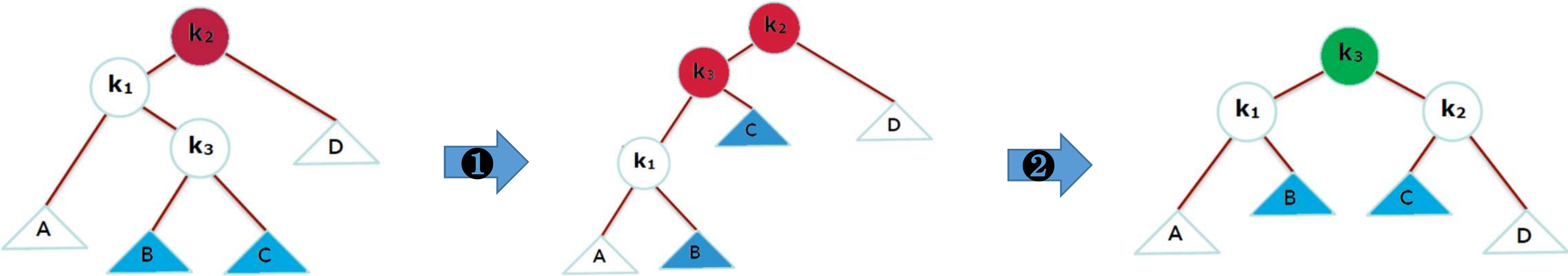

```
Node* AvlTree::RotateL(Node *nd)
{
    Node *x = nd->right;
    nd->right = x->left;
    x->left = nd;

    nd->height = max(getHeight(nd->left), getHeight(nd->right)) + 1;
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

    return x;
}
```



左旋右旋LR 和 右旋左旋RL

```
Node* AvlTree::RotateLR(Node *nd)
{
    Node *x = nd->left;
    nd->left = RotateL(x);
    return RotateR(nd);
}
```

```
Node* AvlTree::RotateRL(Node *nd)
{
    Node *x = nd->right;
    nd->right = RotateR(x);
    return RotateL(nd);
}
```



辅助函数

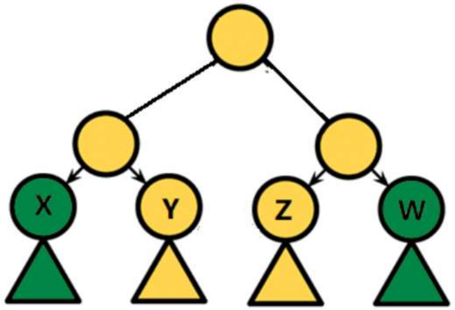
```
struct Node
{
    int key;
    int height;
    Node * left;
    Node * right;

    Node(int val)
    {
        key = val;
        height = 0;
        left = right = nullptr;
    }
};
```

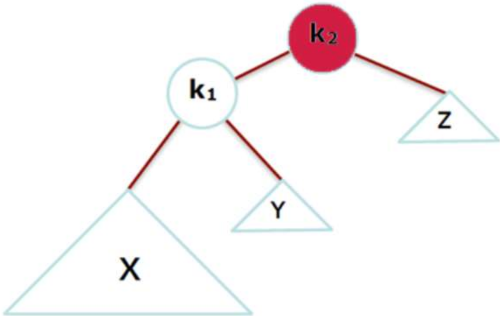
```
int AvlTree::getHeight(Node *nd)
{
    if (nd == nullptr)
    {
        return -1;
    }
    return nd->height;
}

int AvlTree::getBalance(Node *nd)
{
    if (nd == nullptr)
    {
        return 0;
    }
    return getHeight(nd->left) - getHeight(nd->right);
}
```

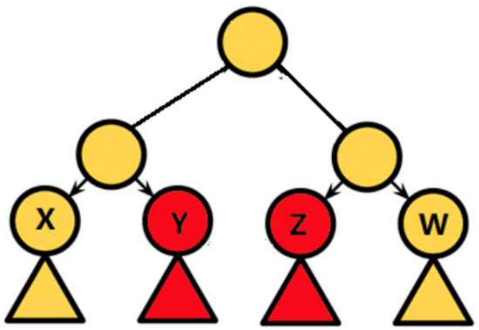
查找

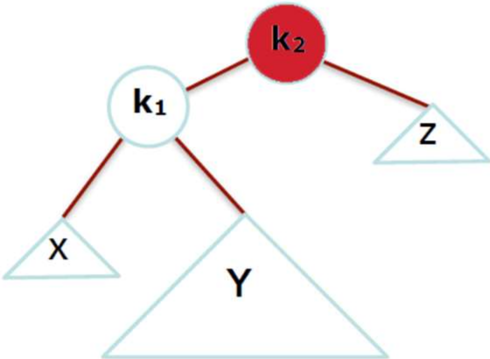
```
Node* AvlTree::InternalSearch(Node *node, int key)
{
    if (node == nullptr)
    {
        return nullptr;
    }

    if (key < node->key)
    {
        return InternalSearch(node->left, key);
    }
    if (key > node->key)
    {
        return InternalSearch(node->right, key);
    }

    return node;
}
```

插入算法





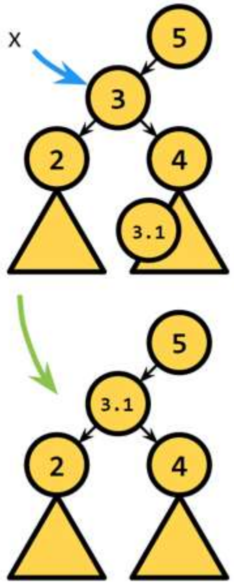





```
Node* AvlTree::InternalInsert(Node *node, int key)
{
    if (node == nullptr)
    {
        return new Node(key);
    }

    if (key < node->key)
    {
        node->left = InternalInsert(node->left, key);
    }
    else if (key > node->key)
    {
        node->right = InternalInsert(node->right, key);
    }
    else
    {
        return node;
    }

    node->height = max(getHeight(node->left), getHeight(node->right)) + 1;

    int balance = getBalance(node);

    // 左左失衡
    if (balance > 1 && getBalance(node->left) >= 0)
    {
        return RotateR(node);
    }
    // 右右失衡
    if (balance < -1 && getBalance(node->right) <= 0)
    {
        return RotateL(node);
    }
    // 左右失衡
    if (balance > 1 && getBalance(node->left) < 0)
    {
        return RotateLR(node);
    }
    // 右左失衡
    if (balance < -1 && getBalance(node->right) > 0)
    {
        return RotateRL(node);
    }
    return node;
}
```

删除算法

```cpp
    node->height = max(getHeight(node->left), getHeight(node->right)) + 1;

    int balance = getBalance(node);

    // 左左失衡
    if (balance > 1 && getBalance(node->left) >= 0)
    {
        return RotateR(node);
    }
    // 右右失衡
    if (balance < -1 && getBalance(node->right) <= 0)
    {
        return RotateL(node);
    }
    // 左右失衡
    if (balance > 1 && getBalance(node->left) < 0)
    {
        return RotateLR(node);
    }
    // 右左失衡
    if (balance < -1 && getBalance(node->right) > 0)
    {
        return RotateRL(node);
    }

    return node;
}
```




```cpp
Node* AvlTree::InternalDelete(Node *node, int key)
{
    if (node == nullptr)
    {
        return node;
    }

    if (key < node->key)
    {
        node->left = InternalDelete(node->left, key);
    }
    else if (key > node->key)
    {
        node->right = InternalDelete(node->right, key);
    }
    else
    {
        if (node->left && node->right)
        {
            // 找到后继结点
            Node * x = node->right;
            while (x->left)
            {
                x = x->left;
            }

            // 后继直接复制
            node->key = x->key;

            // 转化为删除后继
            node->right = InternalDelete(node->right, x->key);
        }
        else
        {
            Node * t = node;
            node = node->left ? node->left : node->right;
            delete t;
            if (node == nullptr)
            {
                return nullptr;
            }
        }
    }
```

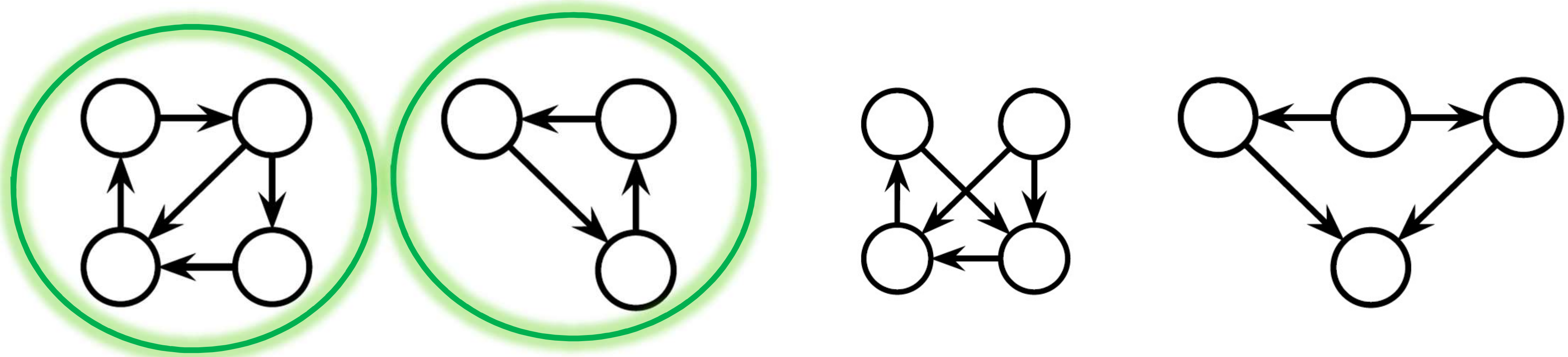
欧拉回路

```
void AvlTree::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 << " ";
    }
}
```

强连通分支
Strongly connected components

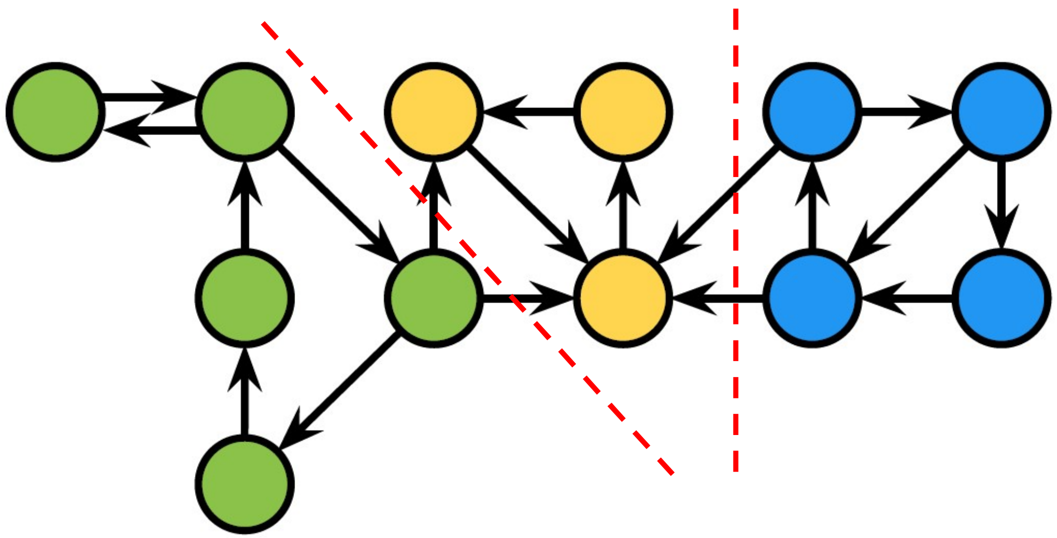
强连通性

- 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，
  - 既有u到v的路径
  - 也有v到u的路径



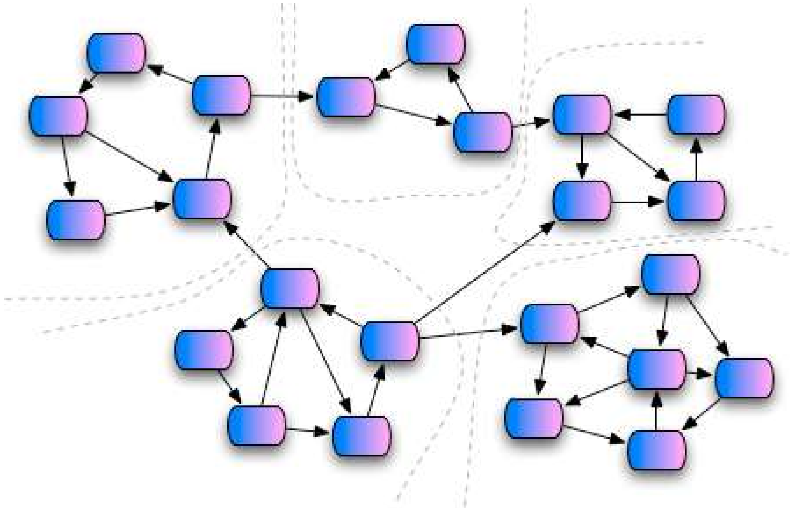
强连通分支

- Strongly Connected Components（缩写为SCC）
- 有向图不见得是强连通的
- 有向图可以分解成强连通分支（为什么？）
- 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
- 然后把互相可达的顶点聚类
- 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$O(n^3)$

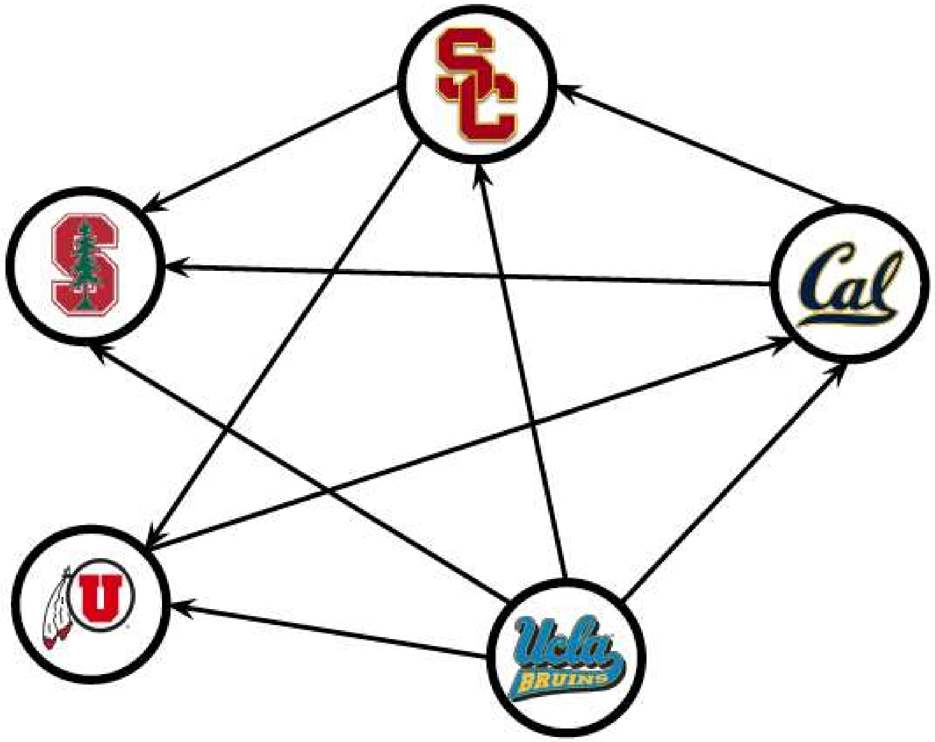
找SCC算法

- 能不能在线性时间内找到有向图的SCC

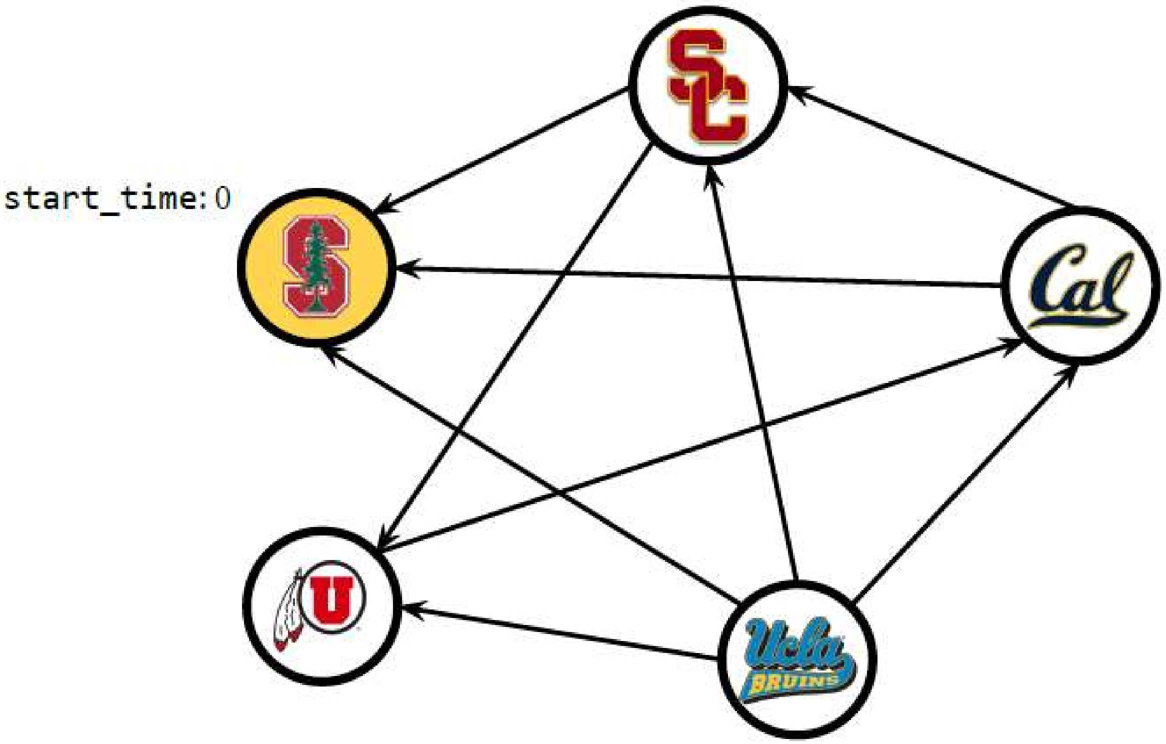
  $O(|V|+|E|)$

- 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

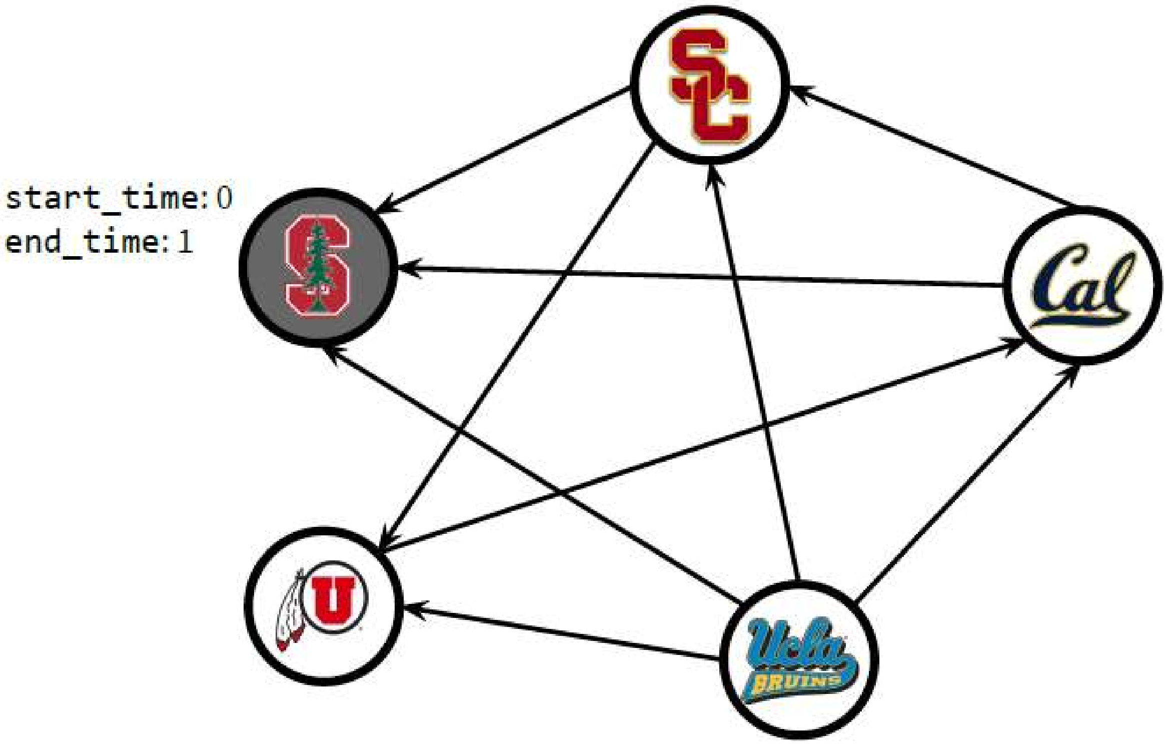
例子：找SCC



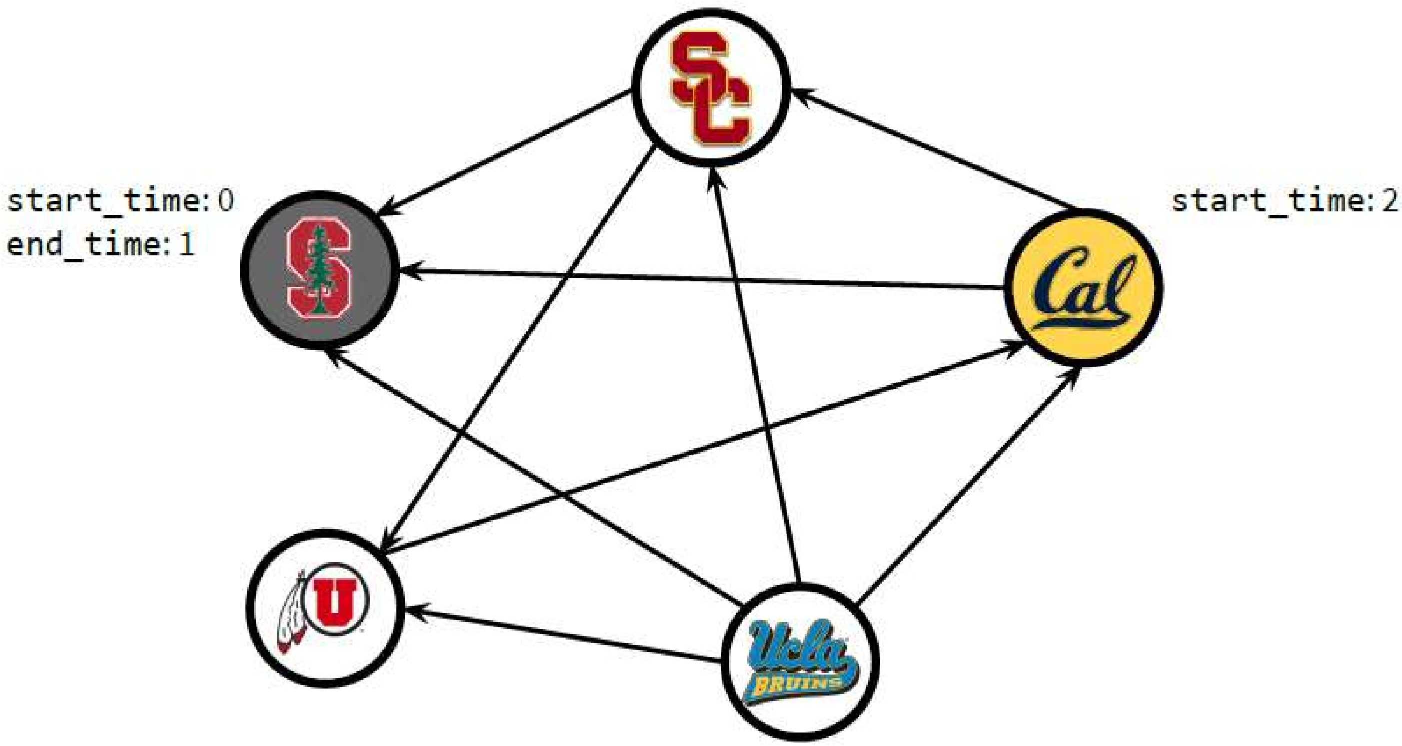
先从一个顶点开始



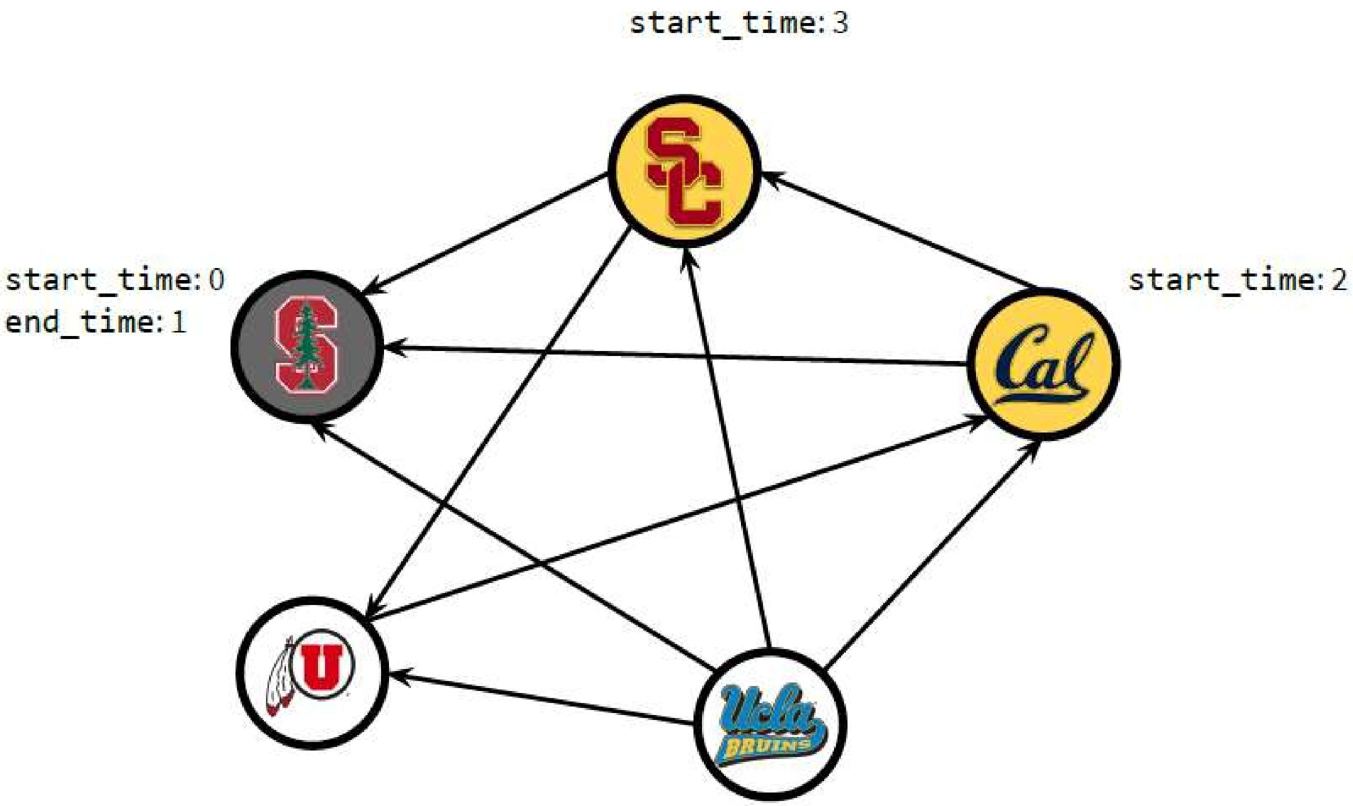
没有邻接顶点，所以完成第一次DFS



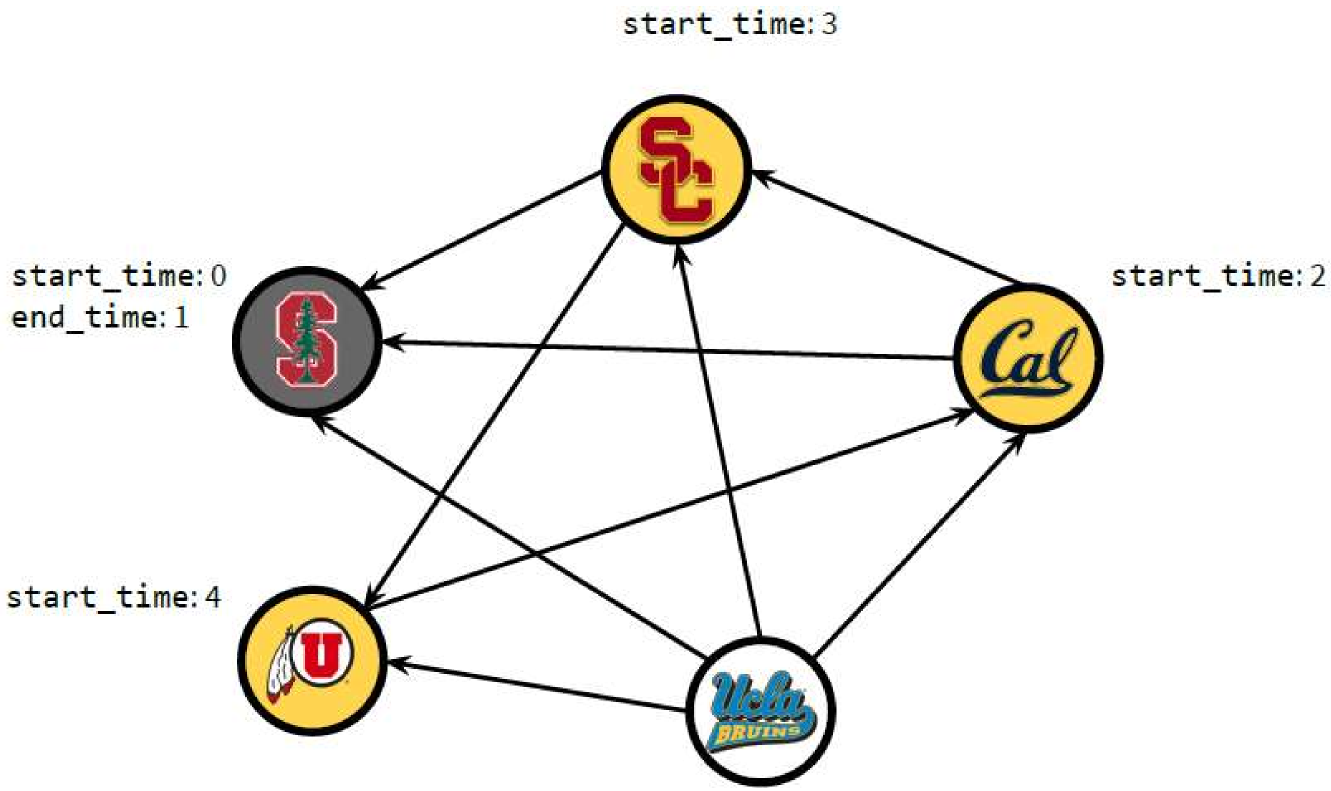
再从剩下未访问顶点选一个开始DFS



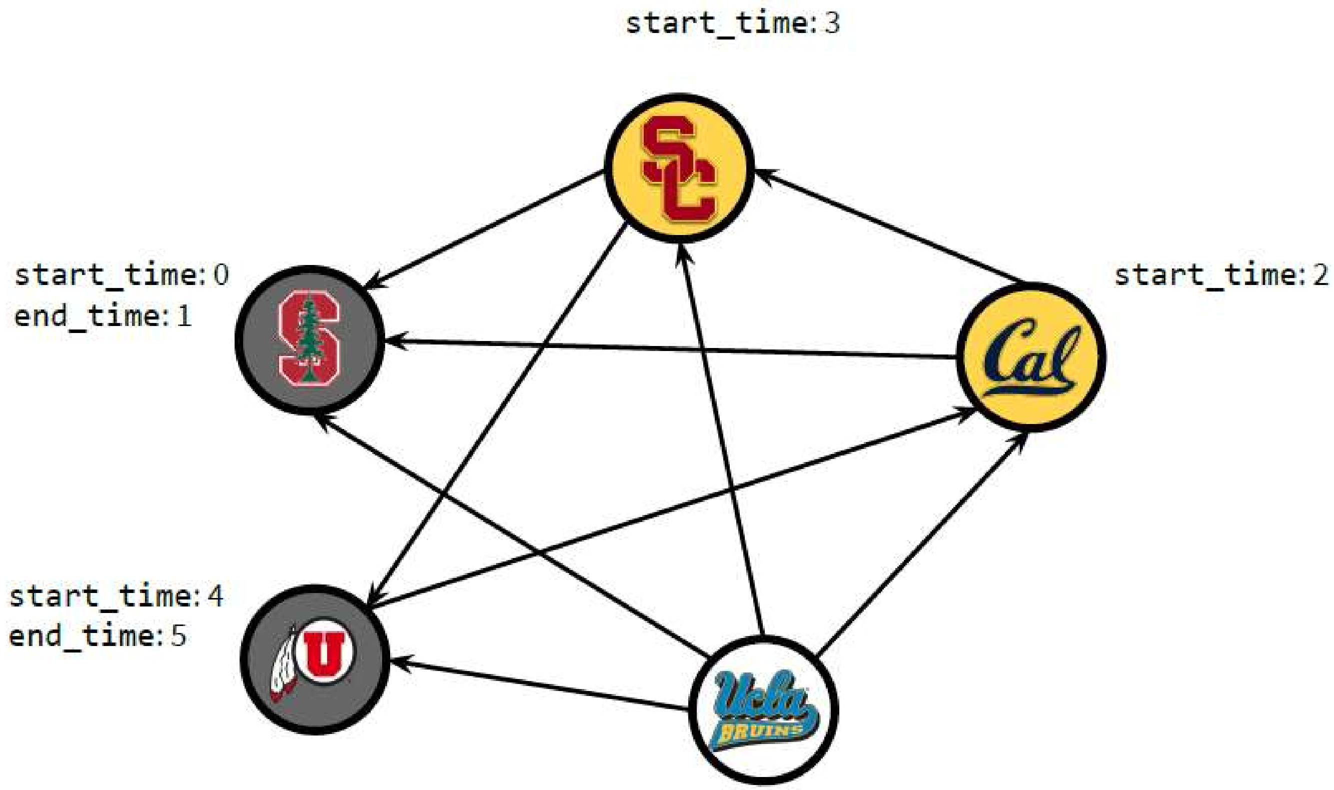
访问其中一个邻接顶点



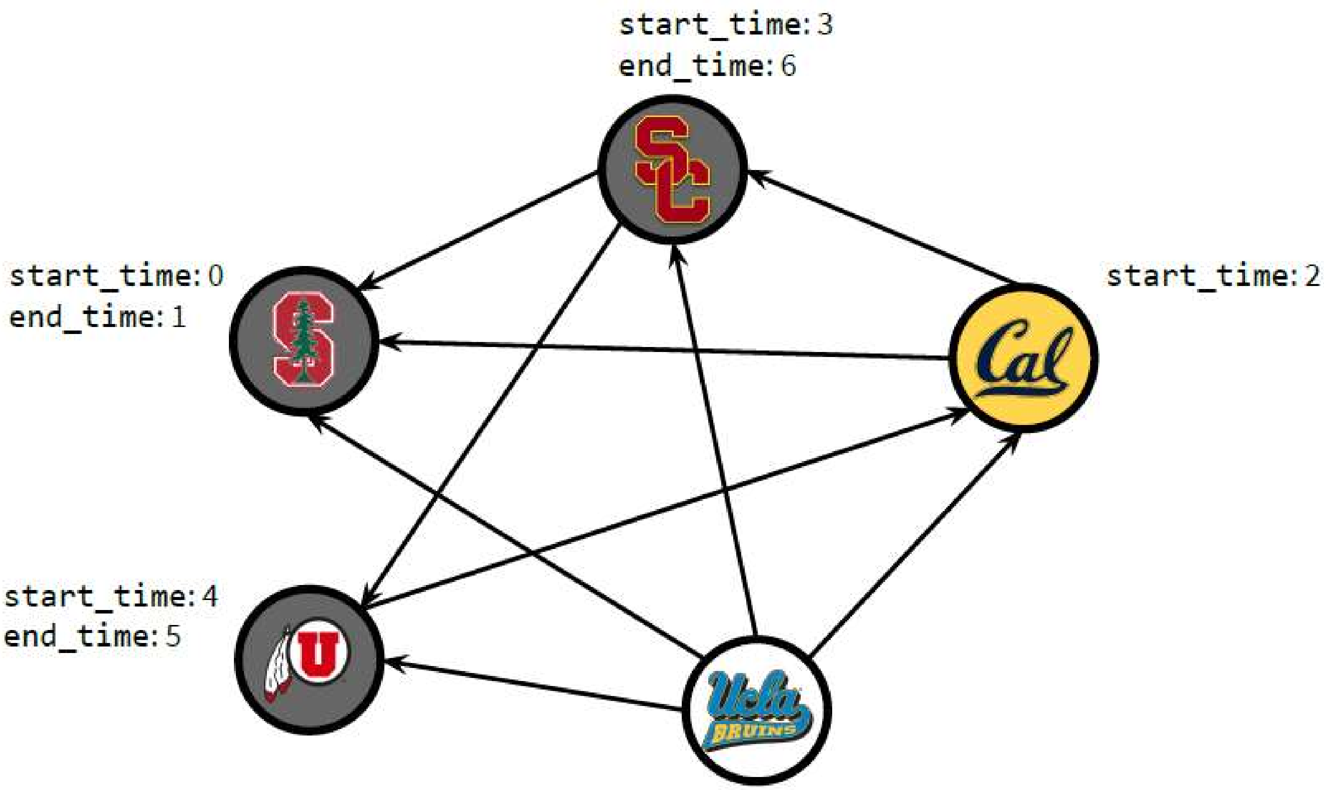
继续访问下一个邻接顶点



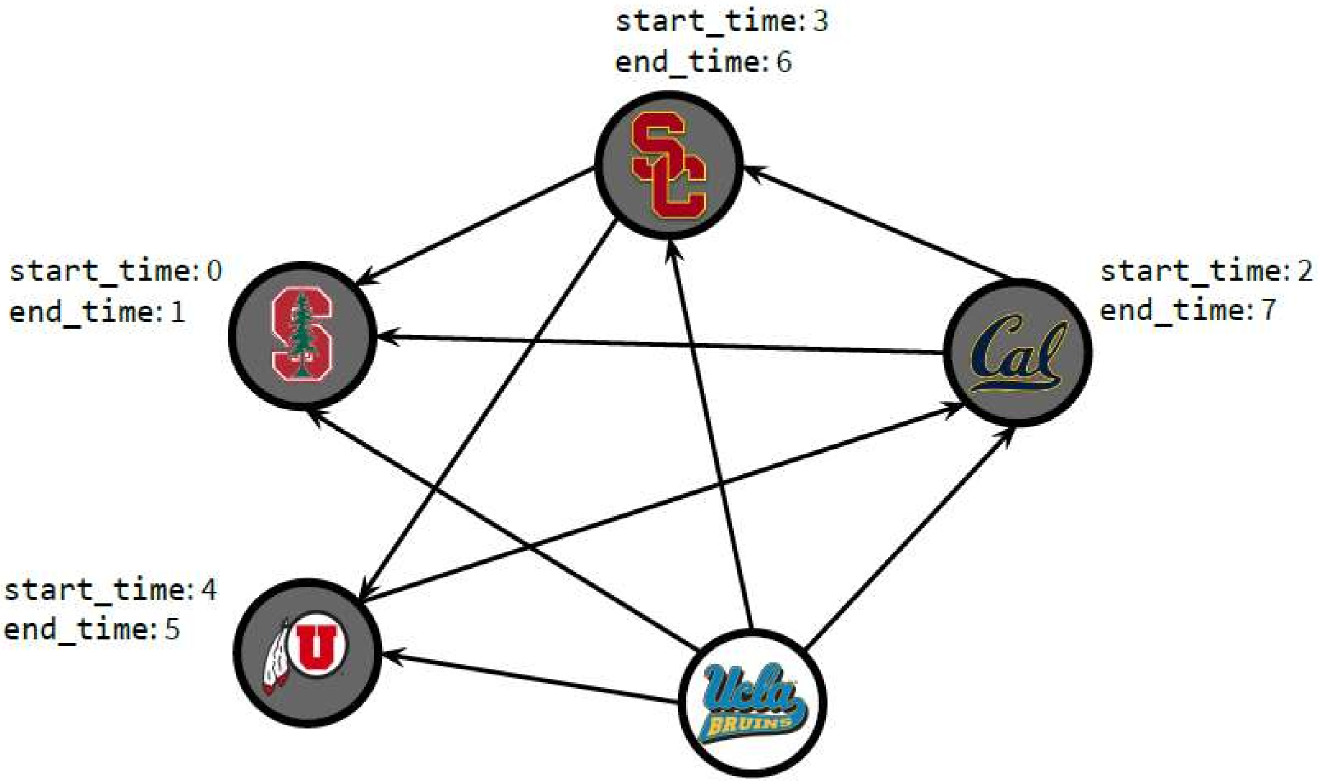
当前访问节点没有未访问邻接顶点



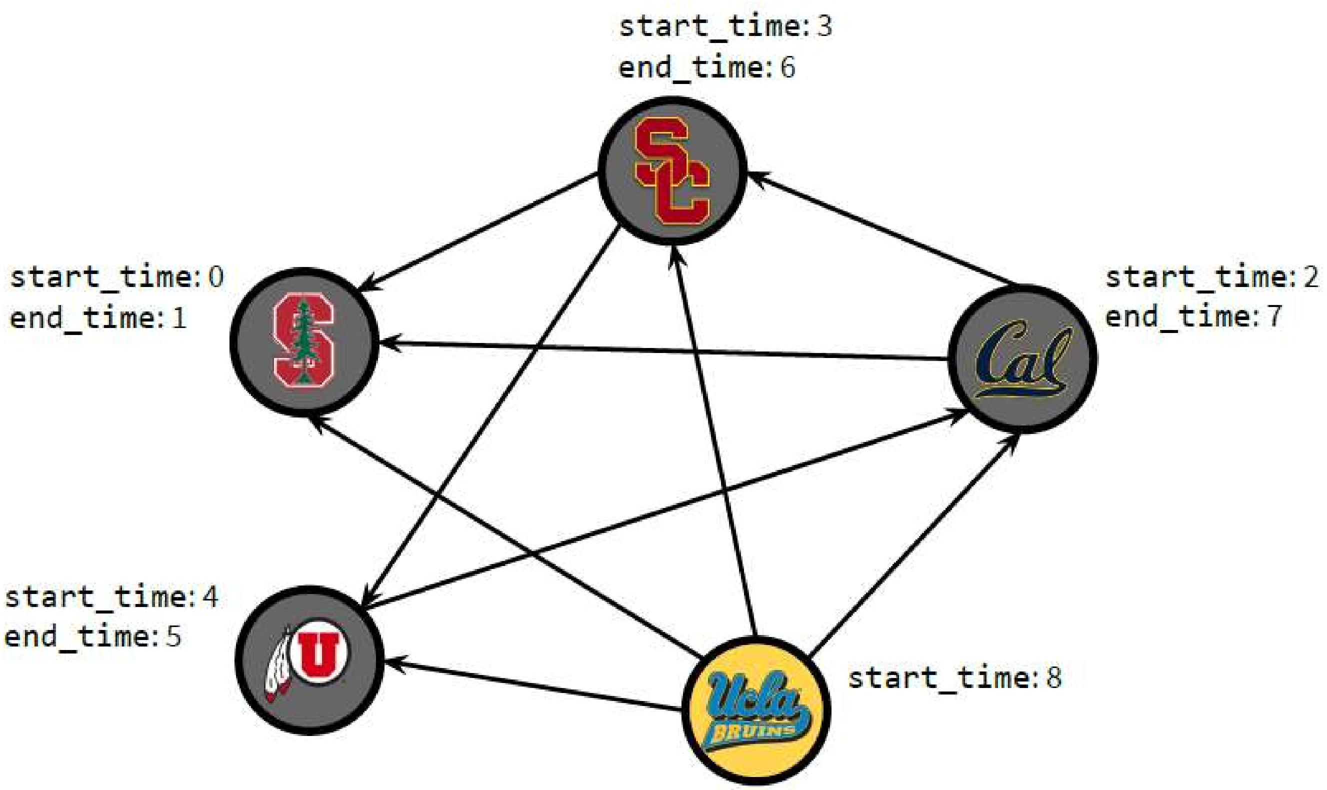
返回后，当前节点也没有未访问邻接顶点



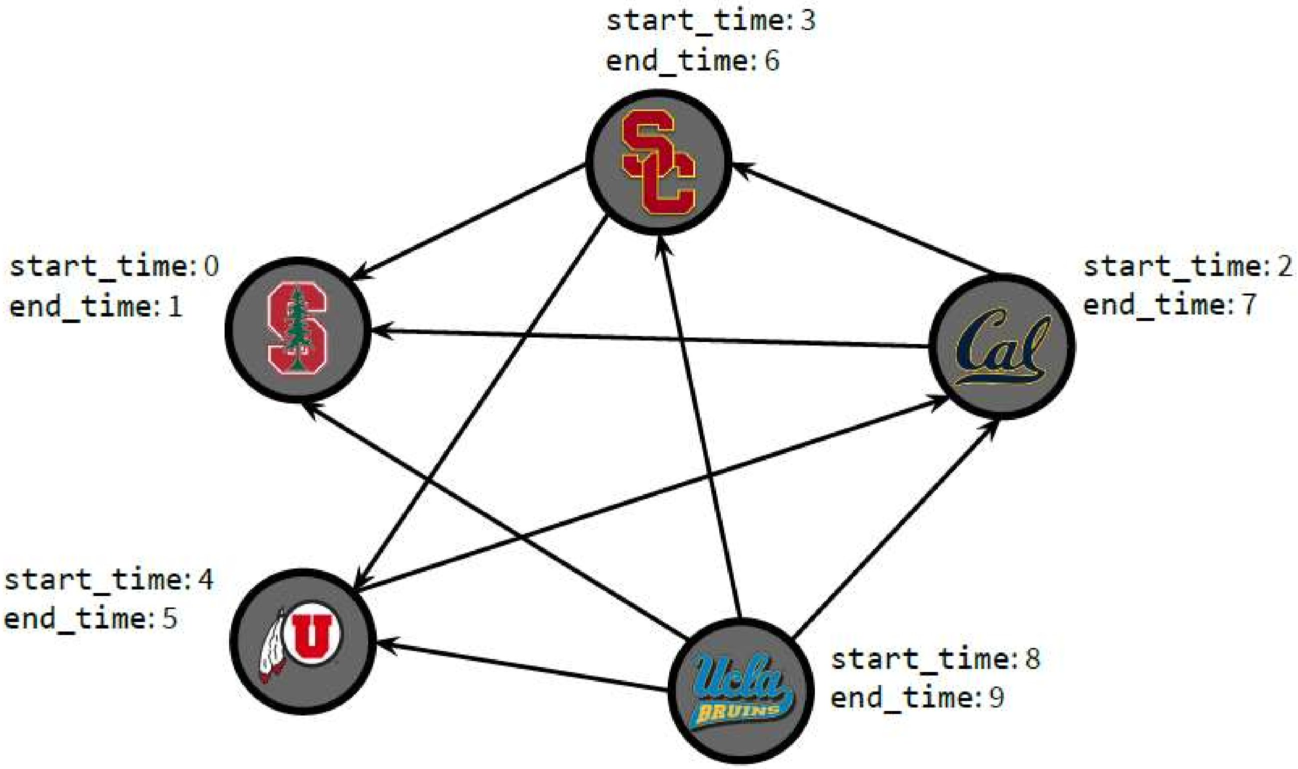
返回后，结束这一次DFS



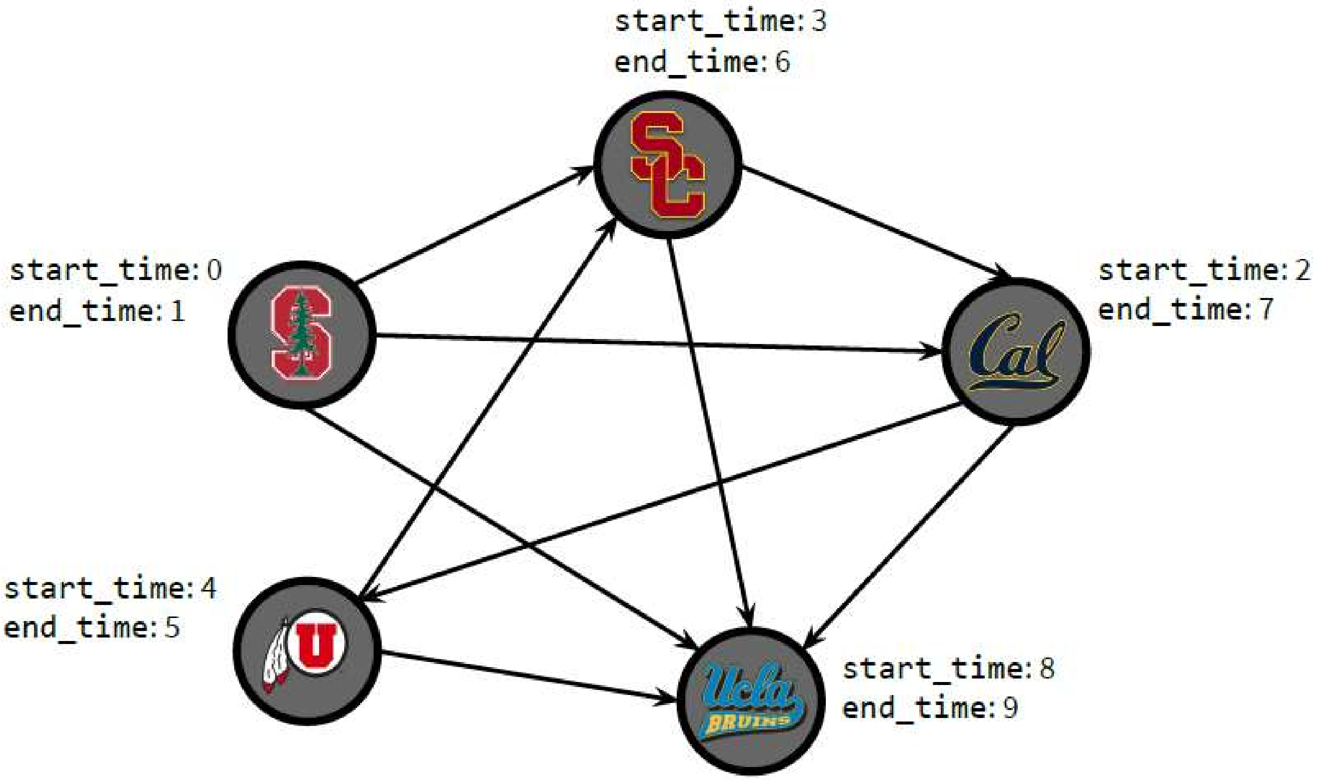
再开始一次DFS


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

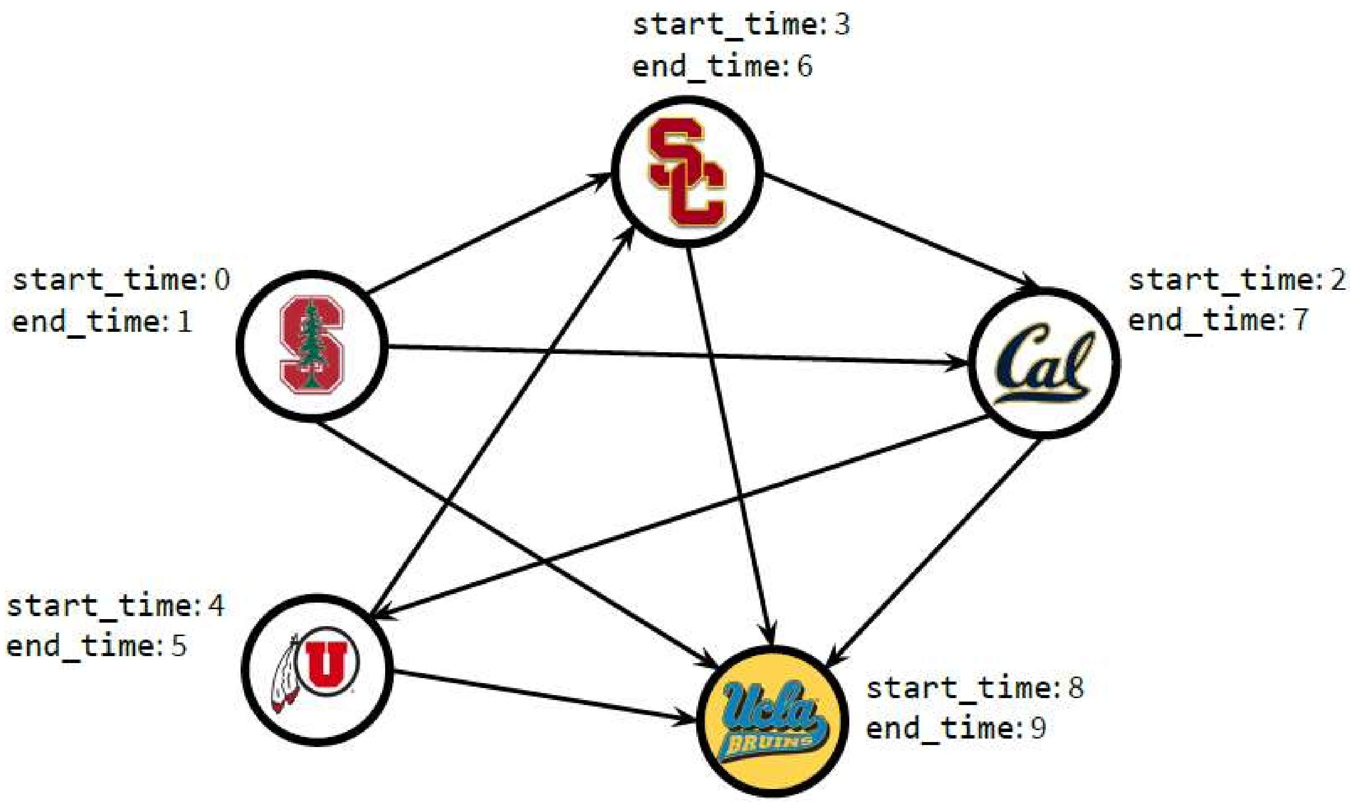
最后一个未访问的顶点结束



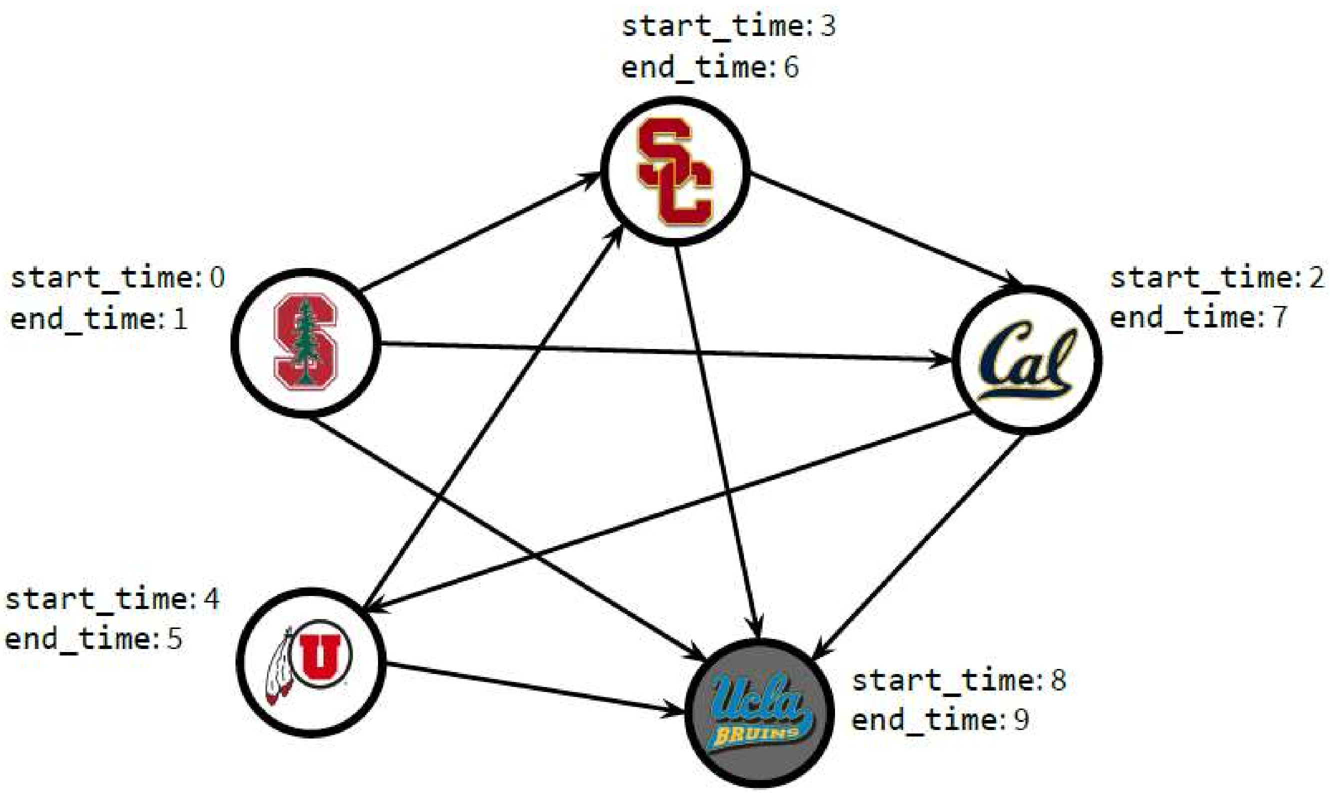
第二步，把所有边反向


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

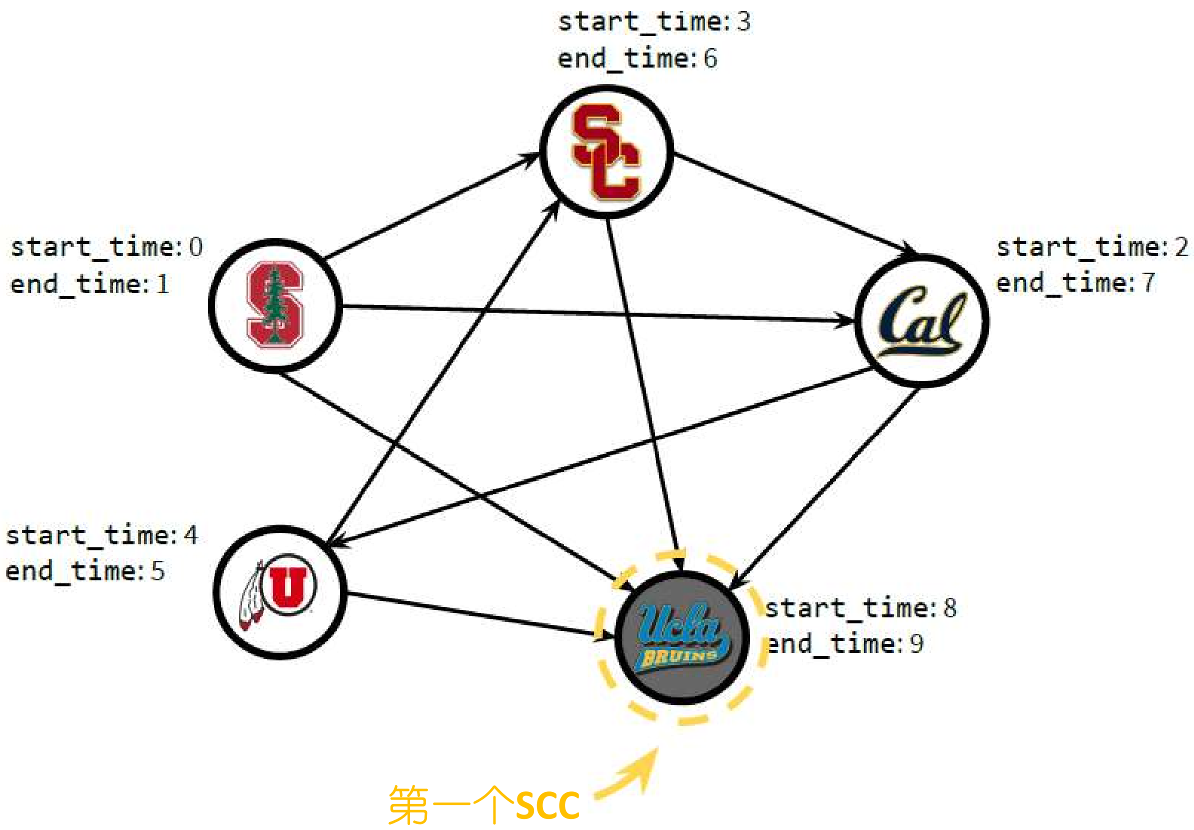
第三步，从结束时间最大的节点开始DFS



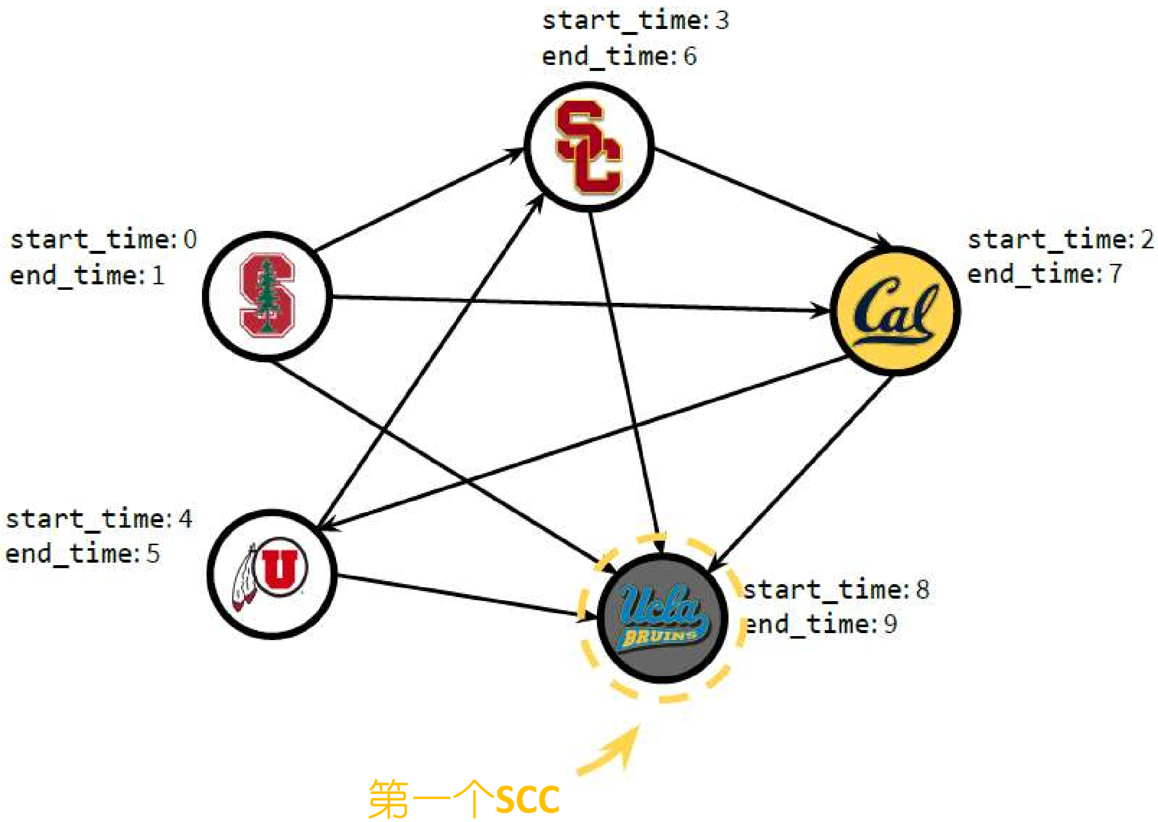
因为没有出边，所以结束访问



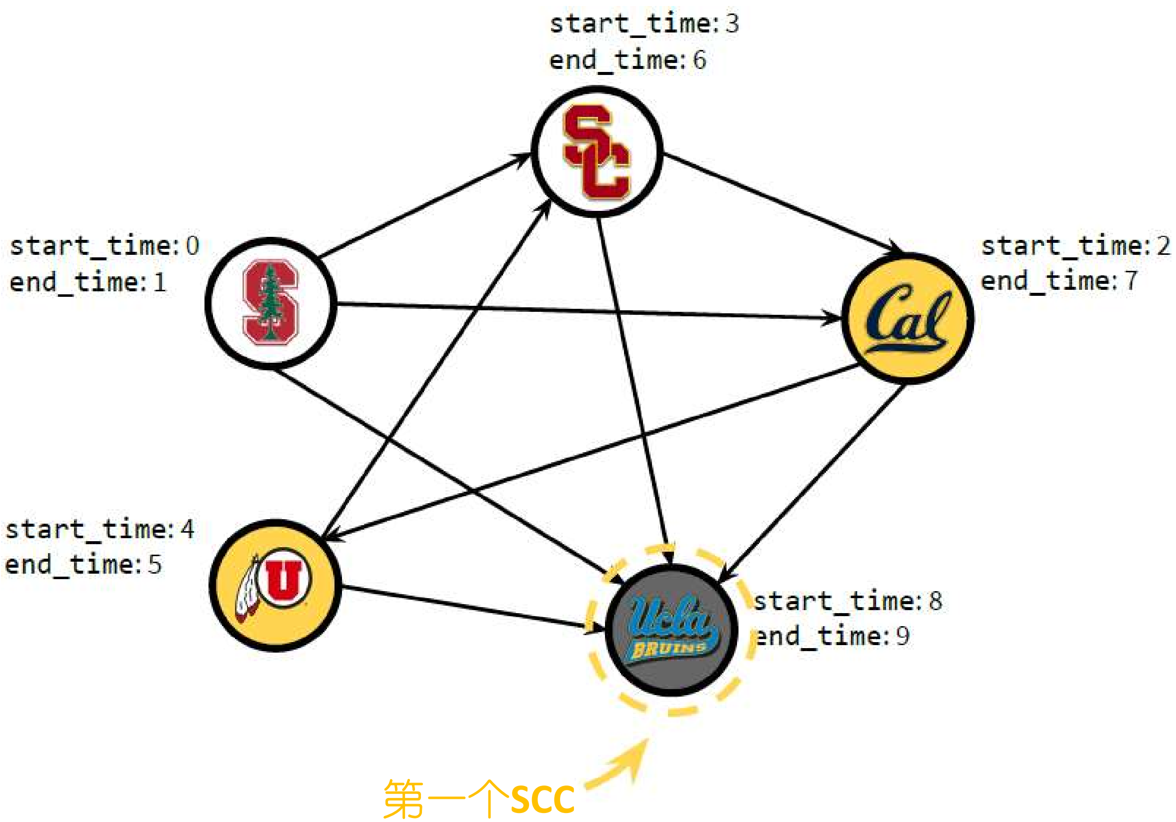
这个DFS树就是一个SCC



再选一个结束时间最大的顶点开始DFS

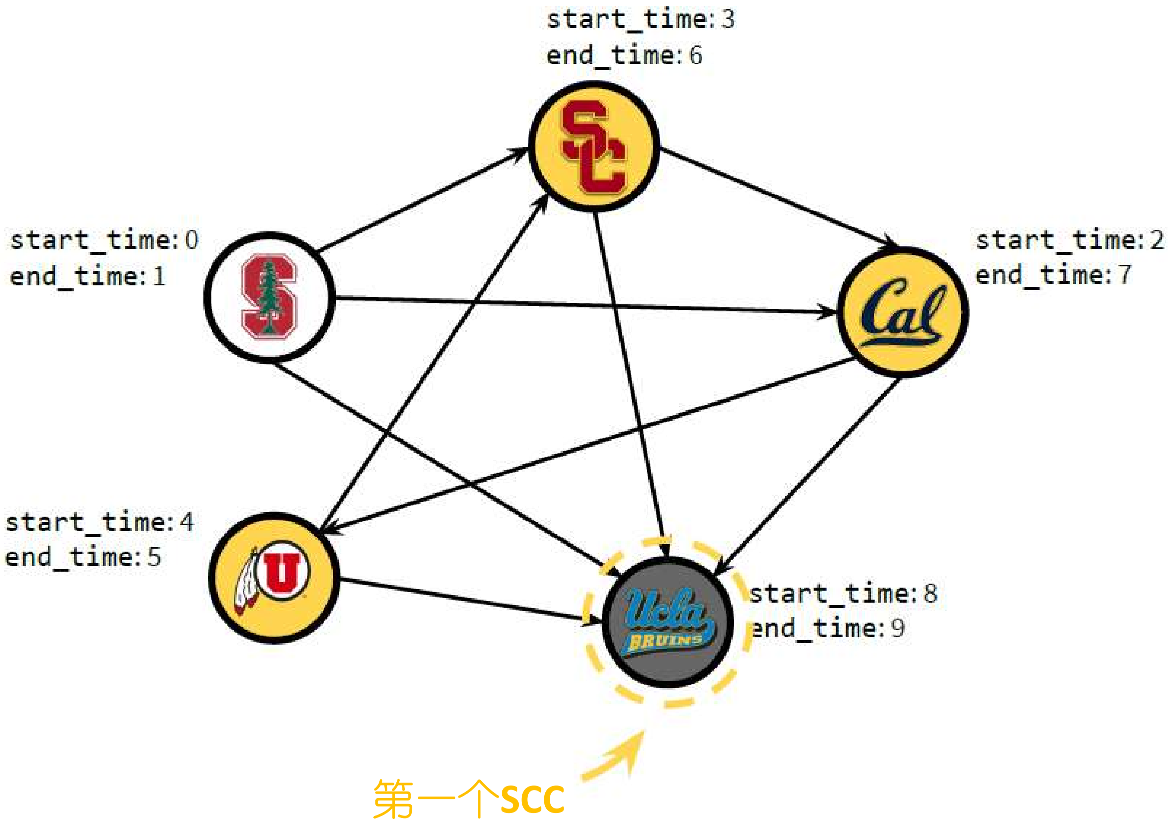


访问其中一个邻接顶点

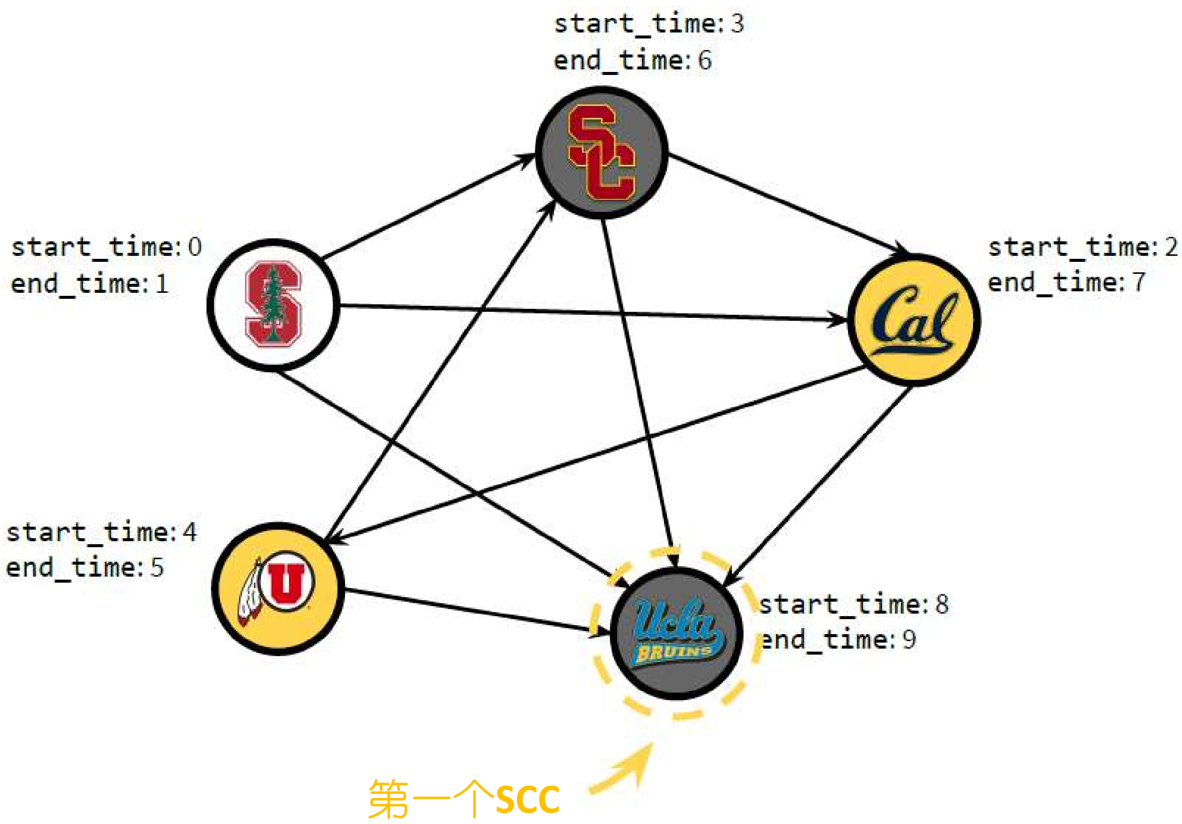


start_time: 3
end_time: 6
start_time: 0
end_time: 1
start_time: 2
end_time: 7
start_time: 4
end_time: 5
start_time: 8
end_time: 9
第一个SCC

继续访问邻接顶点

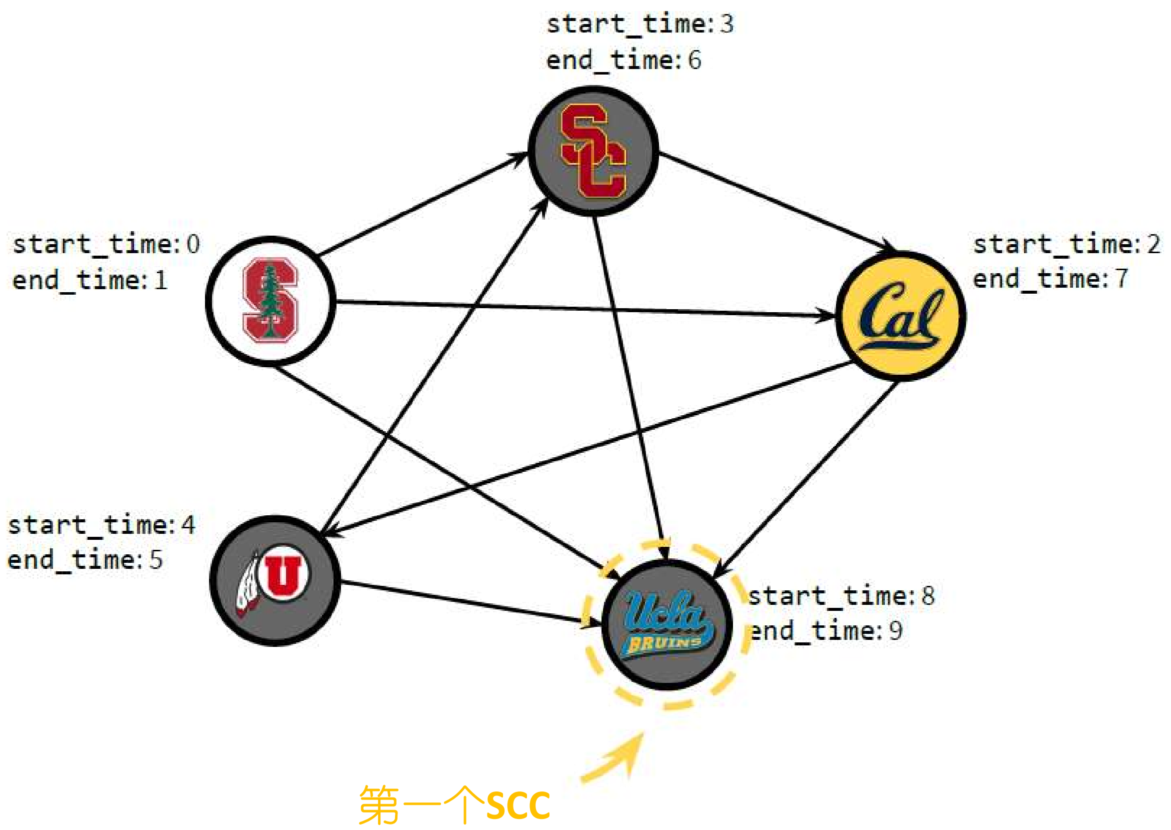

start_time: 3
end_time: 6
start_time: 0
end_time: 1
start_time: 2
end_time: 7
start_time: 4
end_time: 5
start_time: 8
end_time: 9
第一个SCC

当前顶点结束，返回

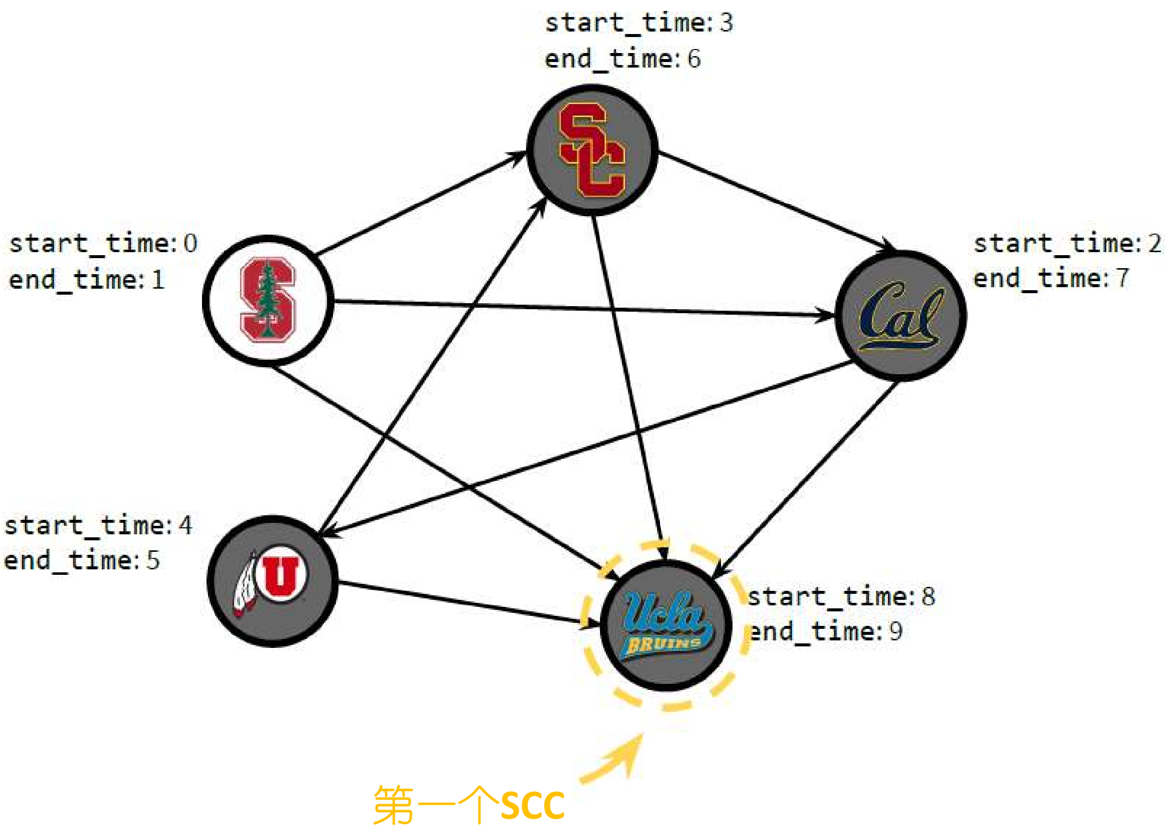


start_time: 3
end_time: 6
start_time: 0
end_time: 1
start_time: 2
end_time: 7
start_time: 4
end_time: 5
start_time: 8
end_time: 9
第一个SCC

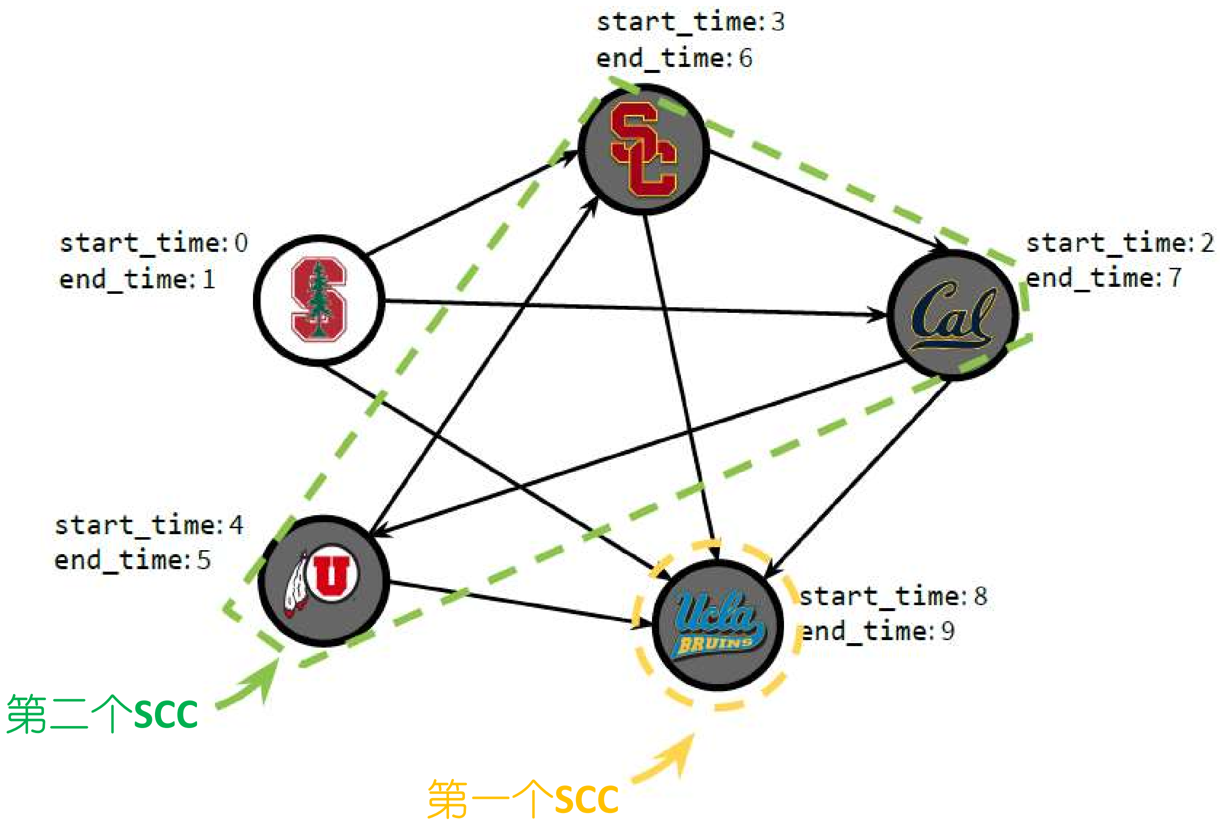
返回后，当前顶点也结束访问



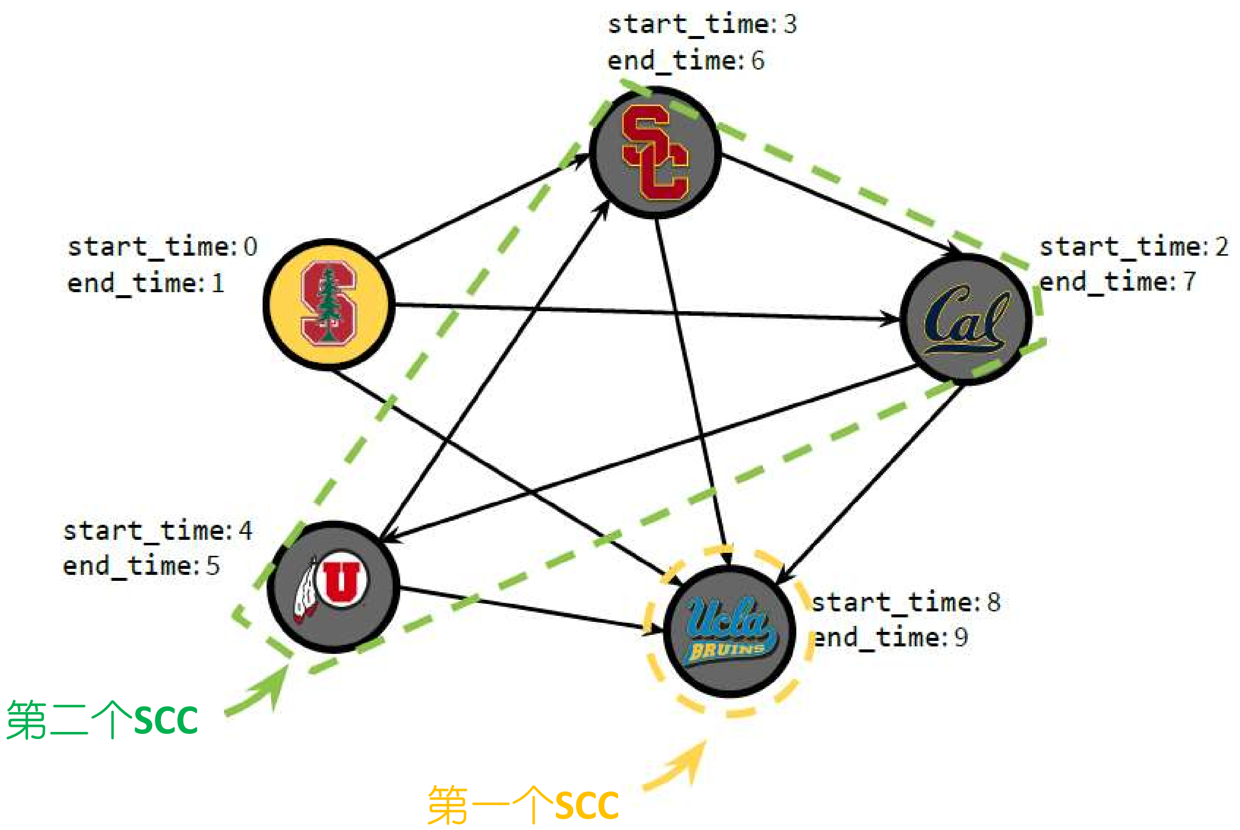
继续返回，第二次DFS结束



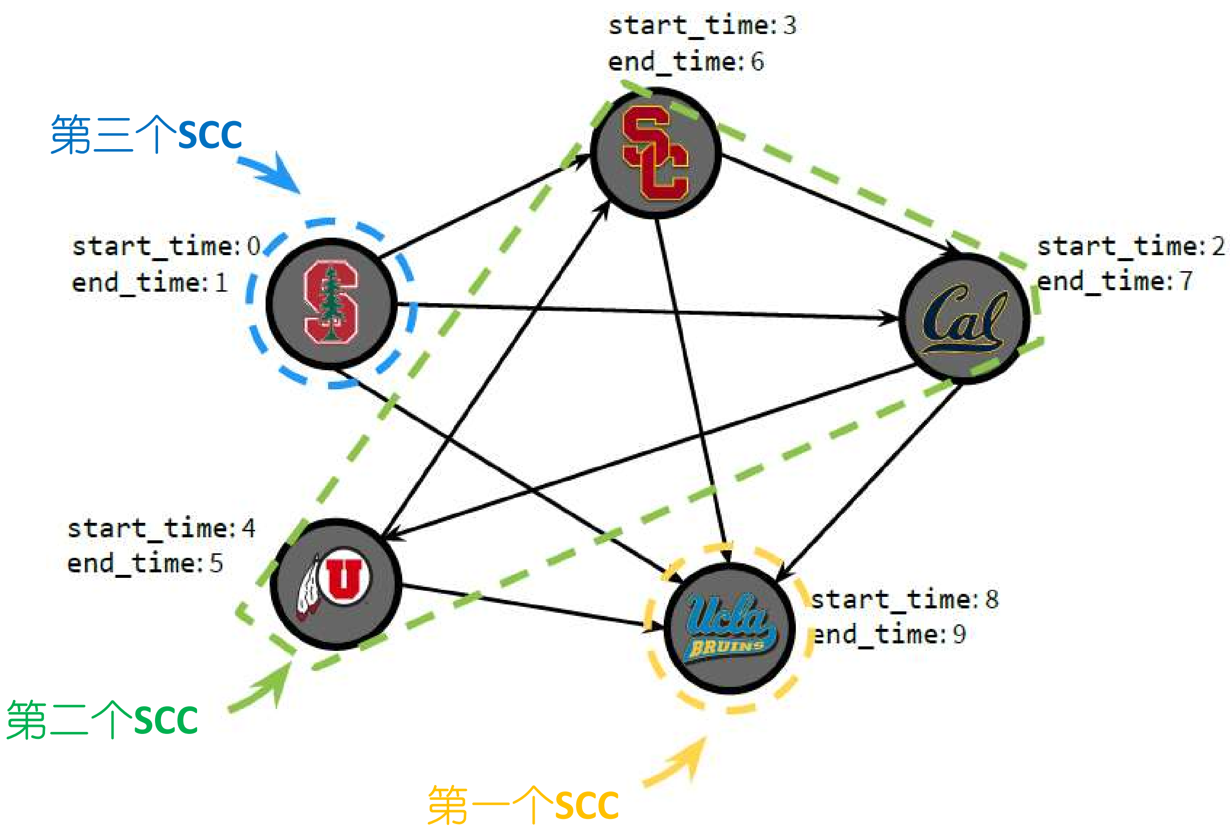
第二个DFS树又是一个SCC



只剩最后一个顶点，它也是一个DFS树



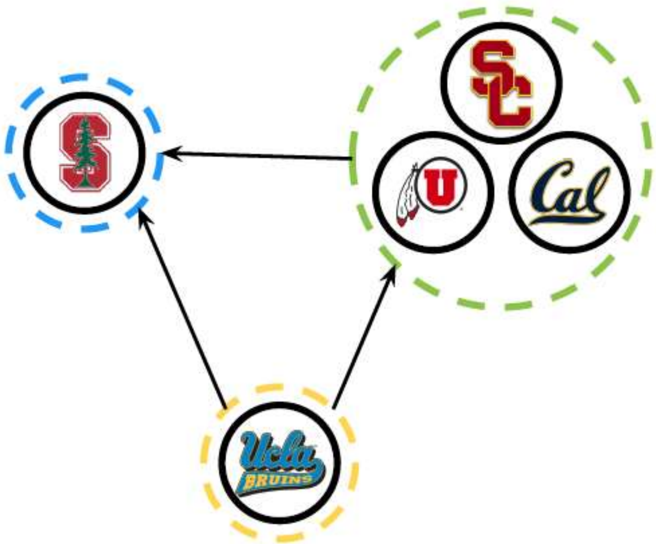
这样就找到三个SCC


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？

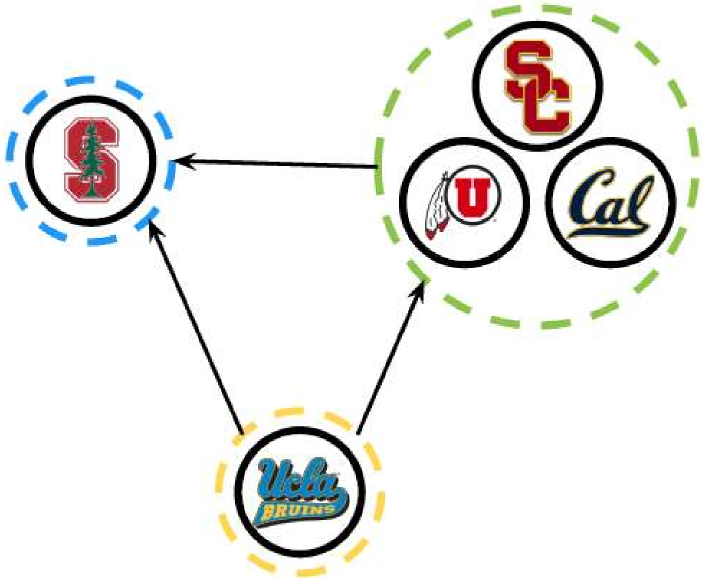
SCC图

- 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



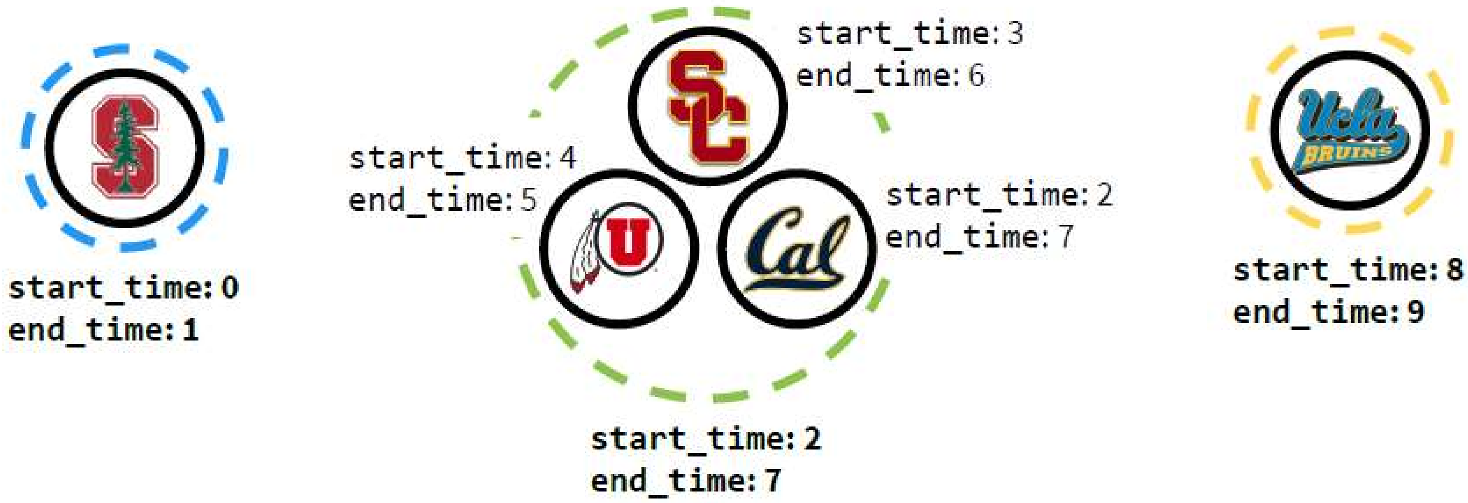
那么，

- SCC图是一个有向无环图
- 思路：如果不是DAG，那么有cycle，环路上的SCC节点可以合并强连通分支



SCC的开始时间和结束时间

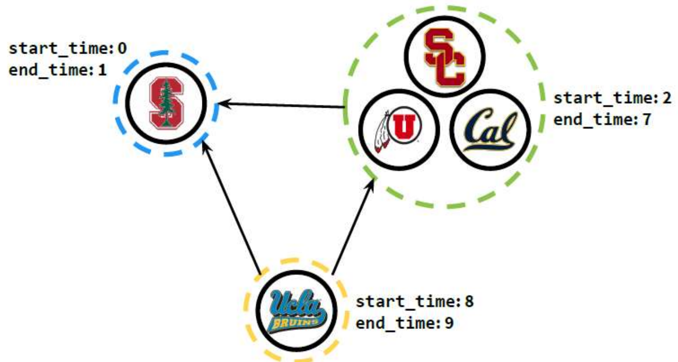
- 定义SCC的开始时间是其中节点开始时间的最小值
- 定义SCC的结束时间是其中节点结束时间的最大值



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

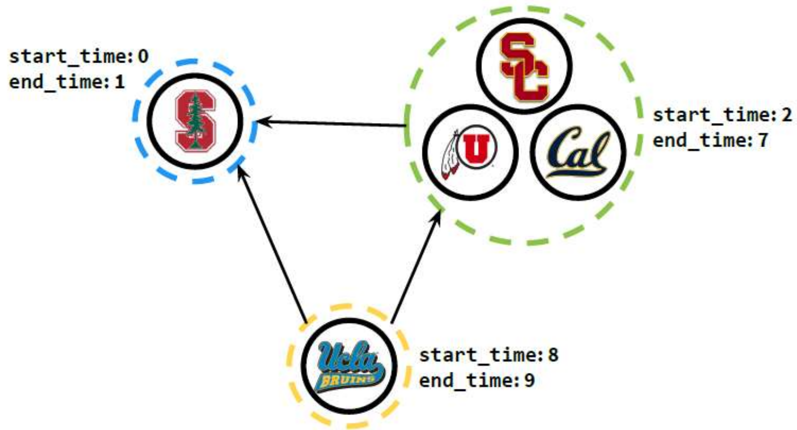
- 观察：结束时间最大的SCC没有入边，
  结束时间最大的顶点一定在这个SCC里面



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

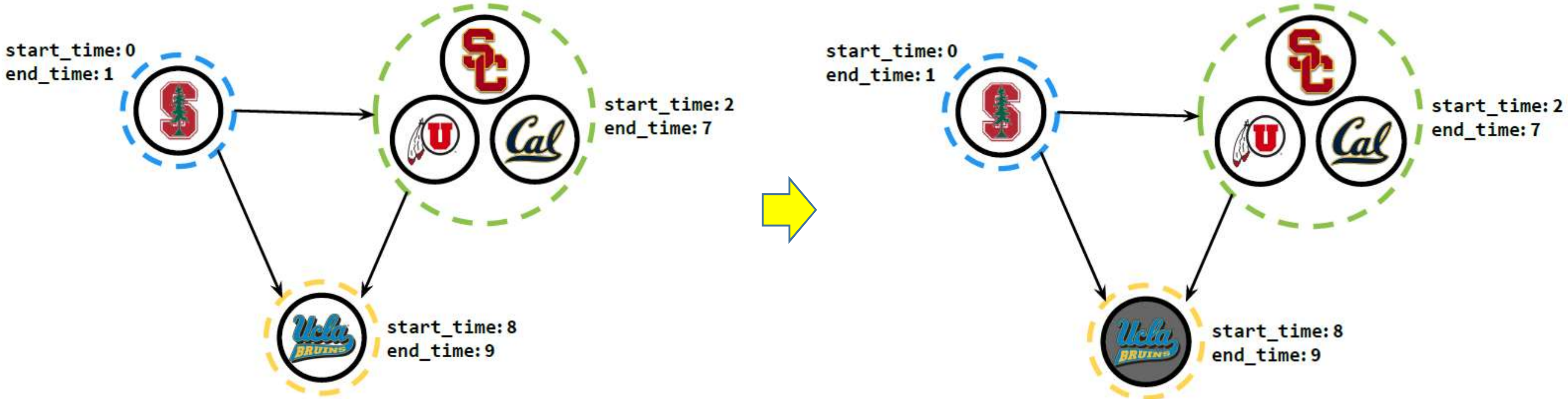
如果不反转方向，先找结束时间最小SCC

- 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（为什么？）



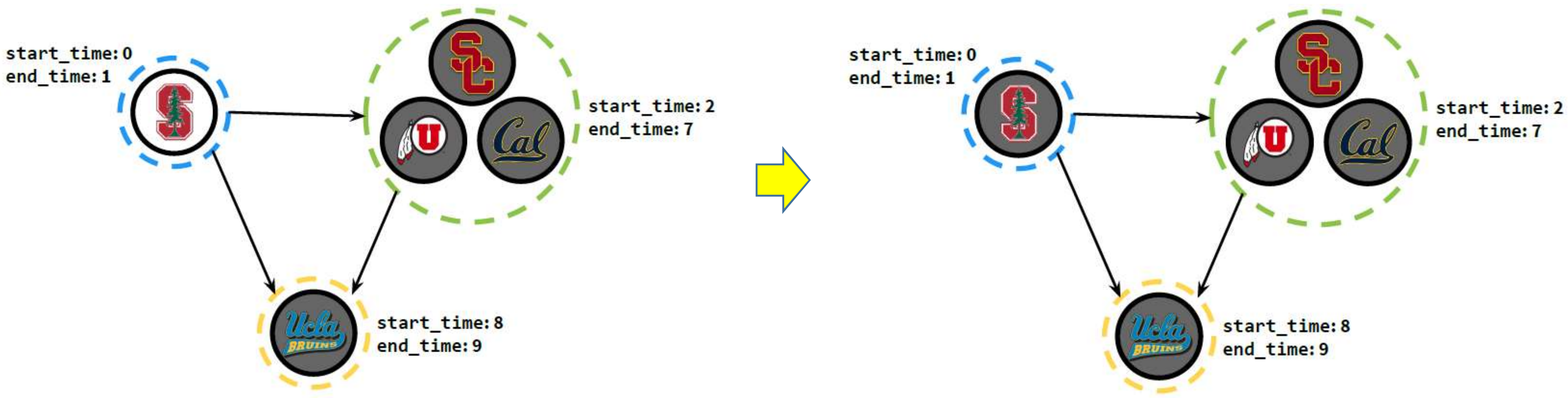
所以必须找结束时间最大的SCC

- 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

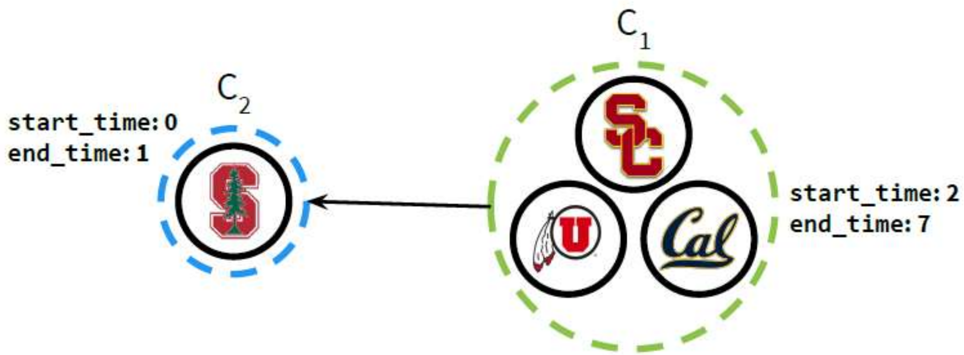
- 忽略找到的SCC中的顶点，剩下的图还是满足之前的性质
- 在剩下的顶点中找结束时间最大的顶点，继续DFS
- 直到所有SCC都找到



一个细节

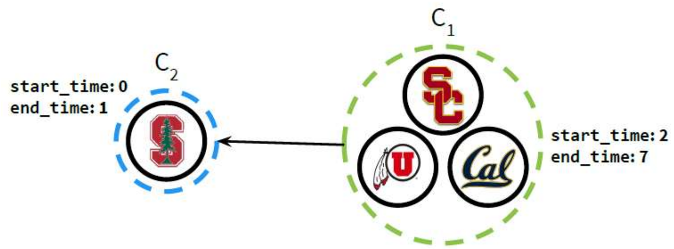
要证明结束时间最大的SCC没有入边，就要说明

- 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

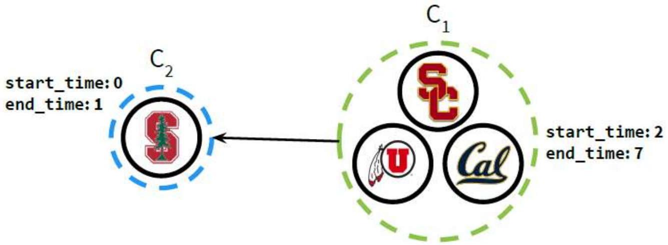


- 那么Y一定在U的DFS子树里，因为U和Y是可达的

- 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

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

Q&A

Thanks!