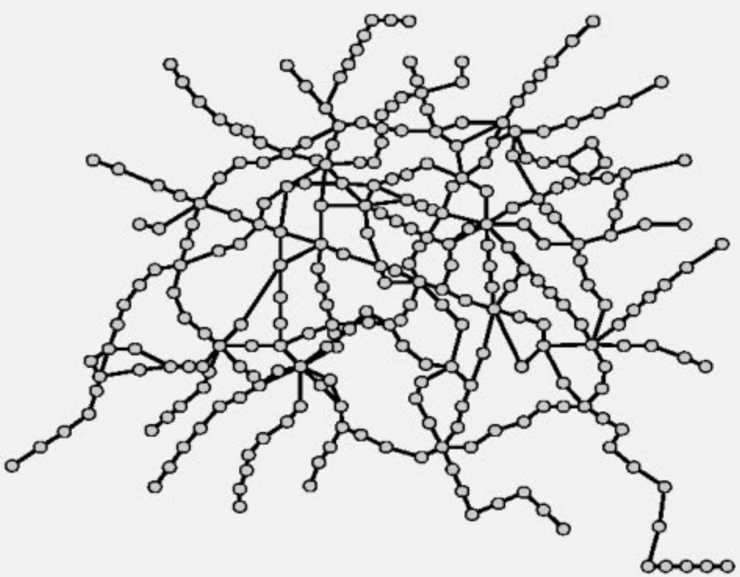
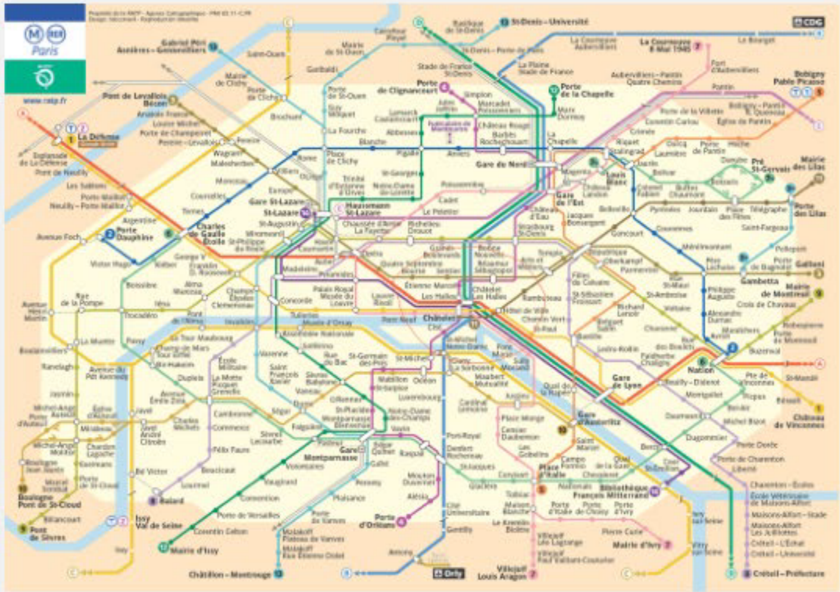
图遍历算法及其性质

赵建华

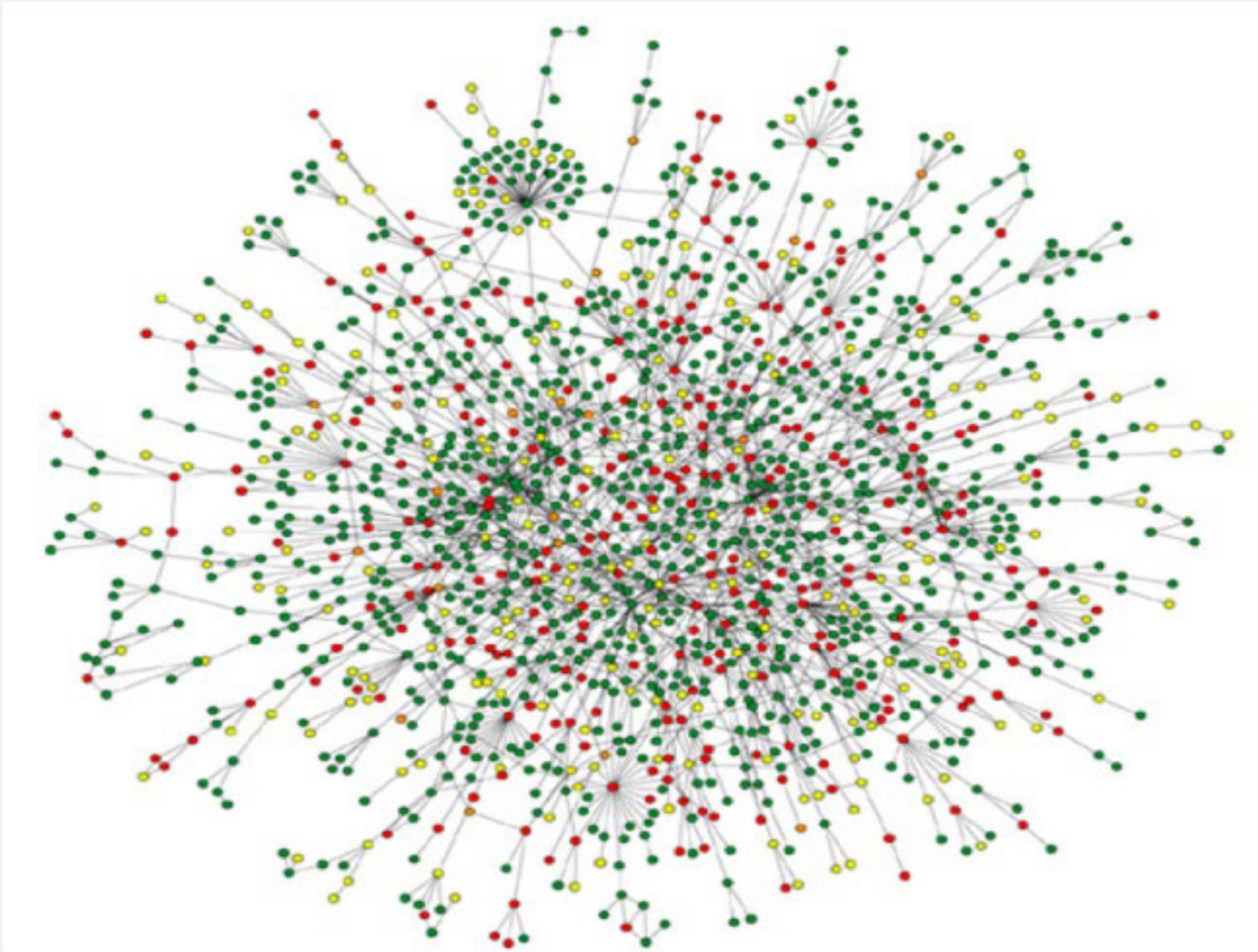
南京大学计算机系

Graph Everywhere

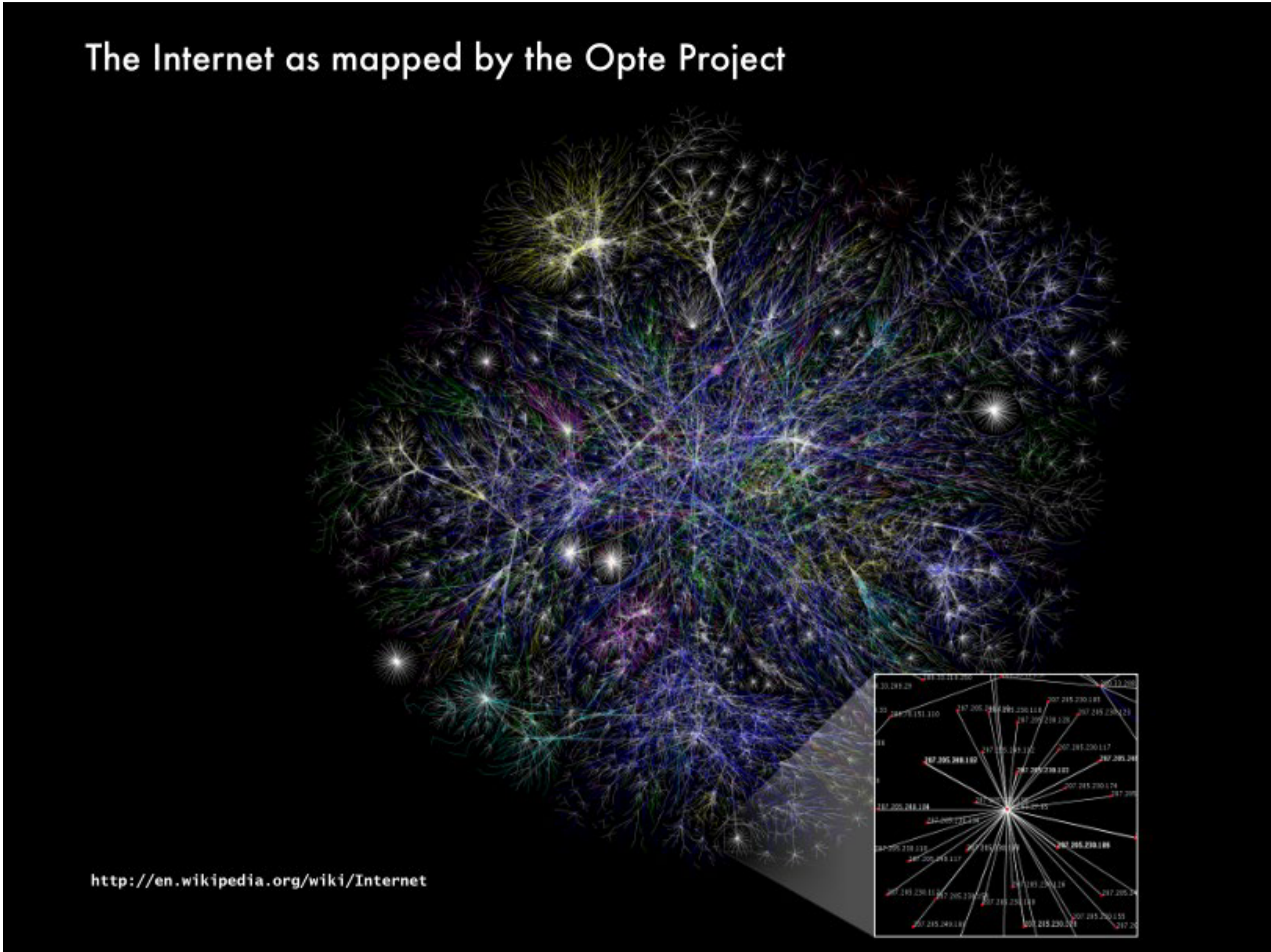




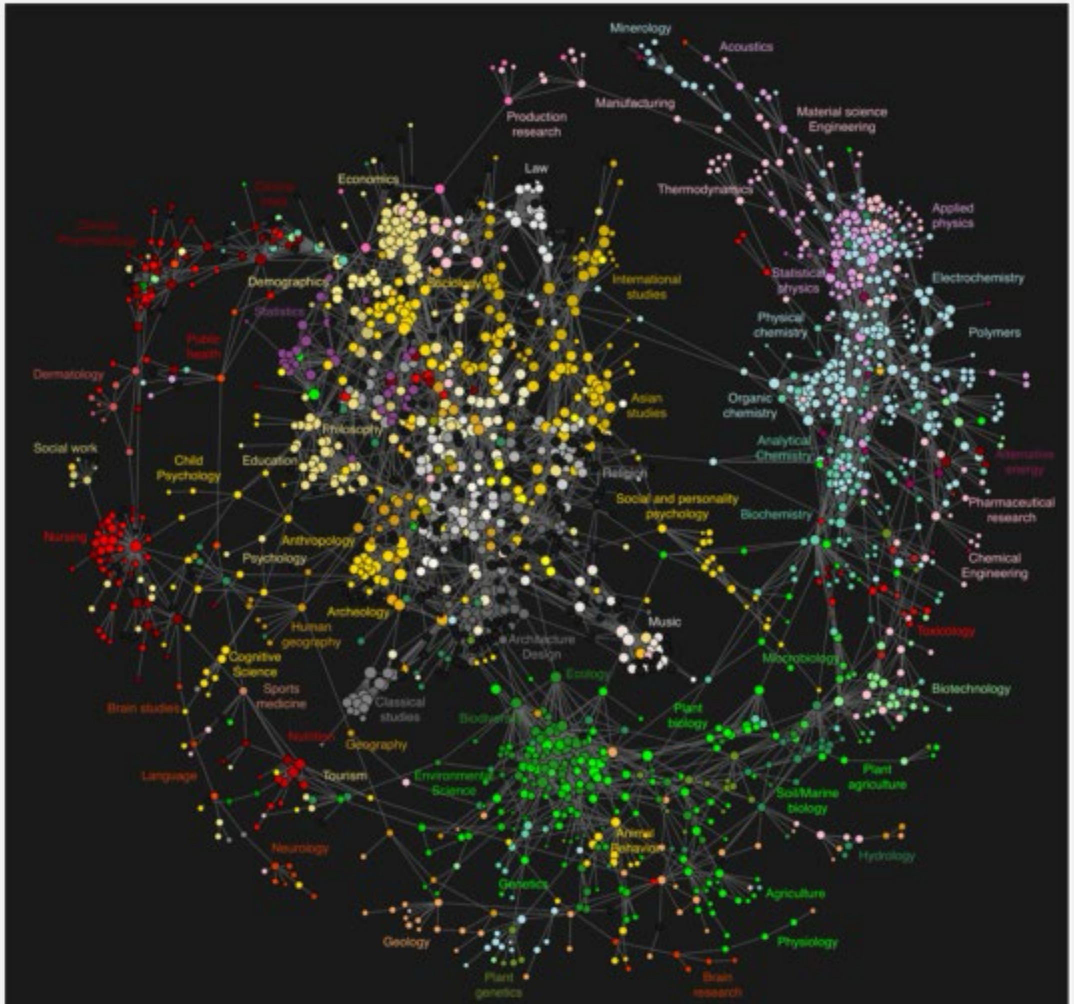
Protein-protein interaction network



Reference: Jeong et al, Nature Review | Genetics


The Internet as mapped by the Opte Project
http://en.wikipedia.org/wiki/Internet

Map of science clickstreams

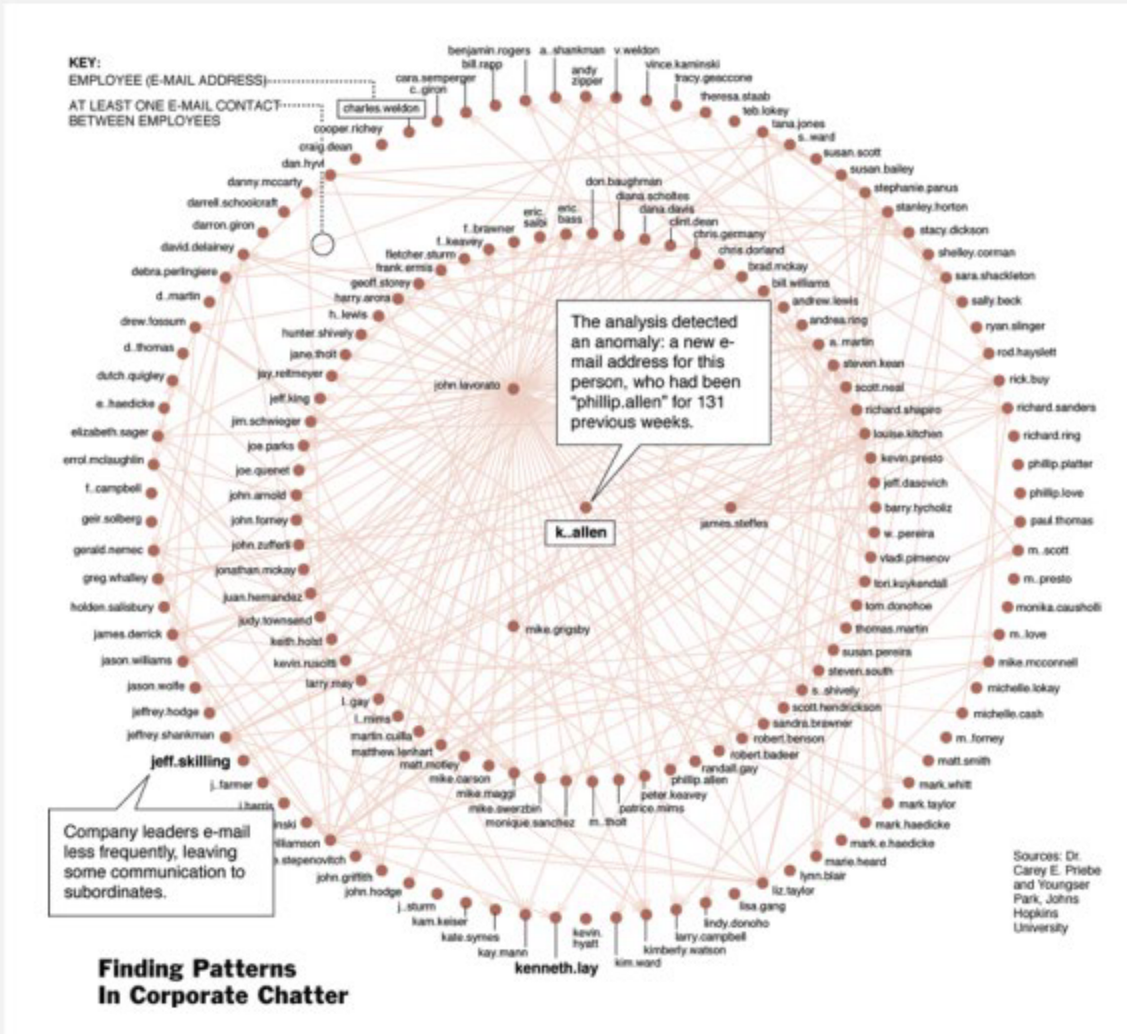


http://www.plosone.org/article/info:doi/10.1371/journal.pone.0004803

10 million Facebook friends

"Visualizing Friendships" by Paul Butler

One week of Enron emails



Graph Basics

- Node
  - Entities of interest
  - $V(G)$
- Edge
  - Relations of interest
  - $E(G) \subseteq V \times V$

Graph Traversals

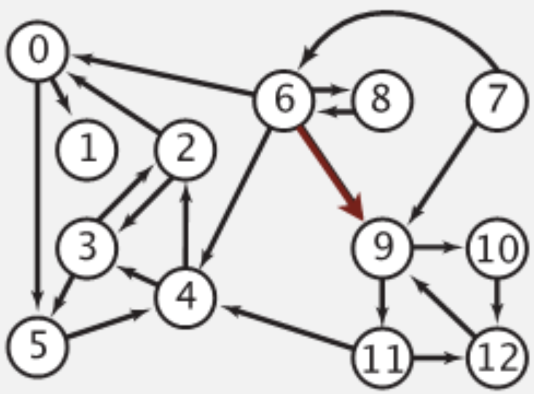
- Depth-First and Breadth-First Search (深度优先搜索和广度优先搜索）
- Finding Connected Components

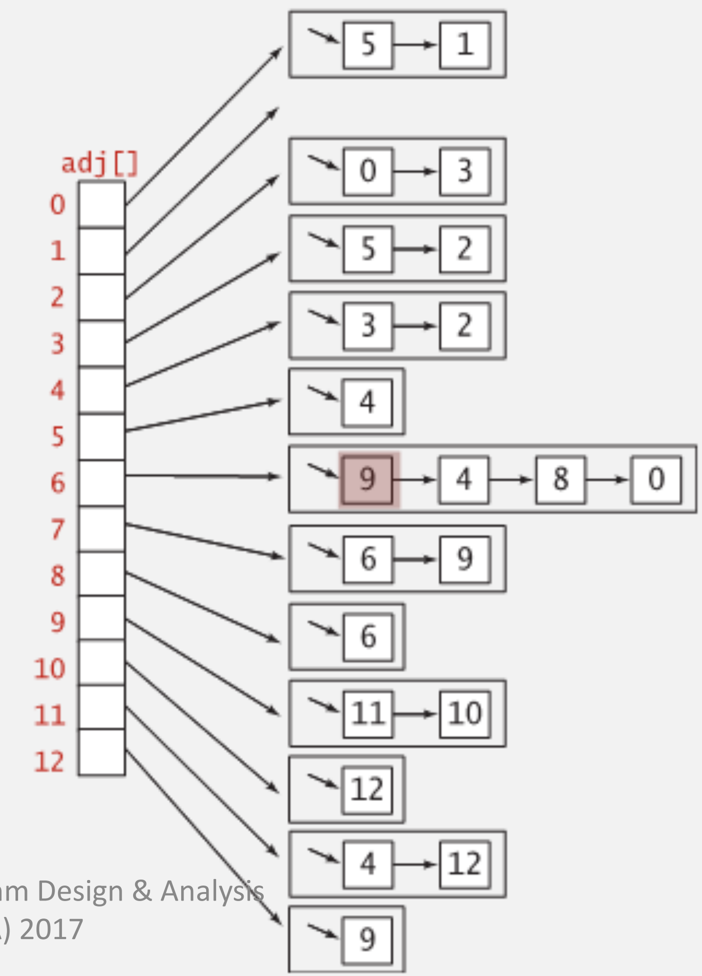
- General Depth-First Search Skeleton
- Depth-First Search Trace

邻接表

Adjacency-lists digraph representation

Maintain vertex-indexed array of lists.





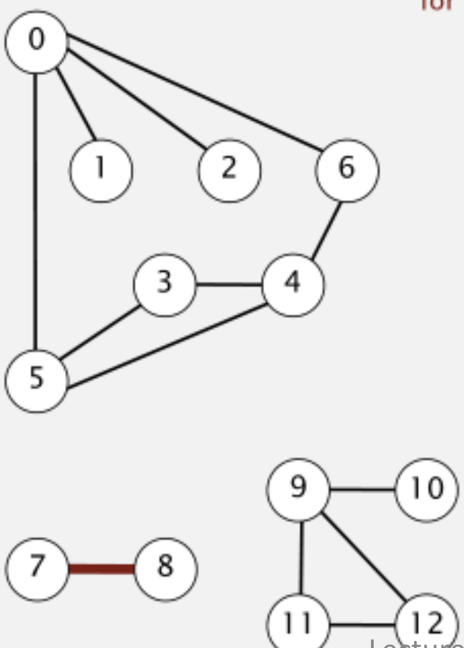
Directed vs. **Undirected** graphs

邻接矩阵

Adjacency-matrix graph representation

Maintain a two-dimensional V-by-V boolean array;

for each edge v–w in graph: `adj[v][w] = adj[w][v] = true.`



two entries
for each edge

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

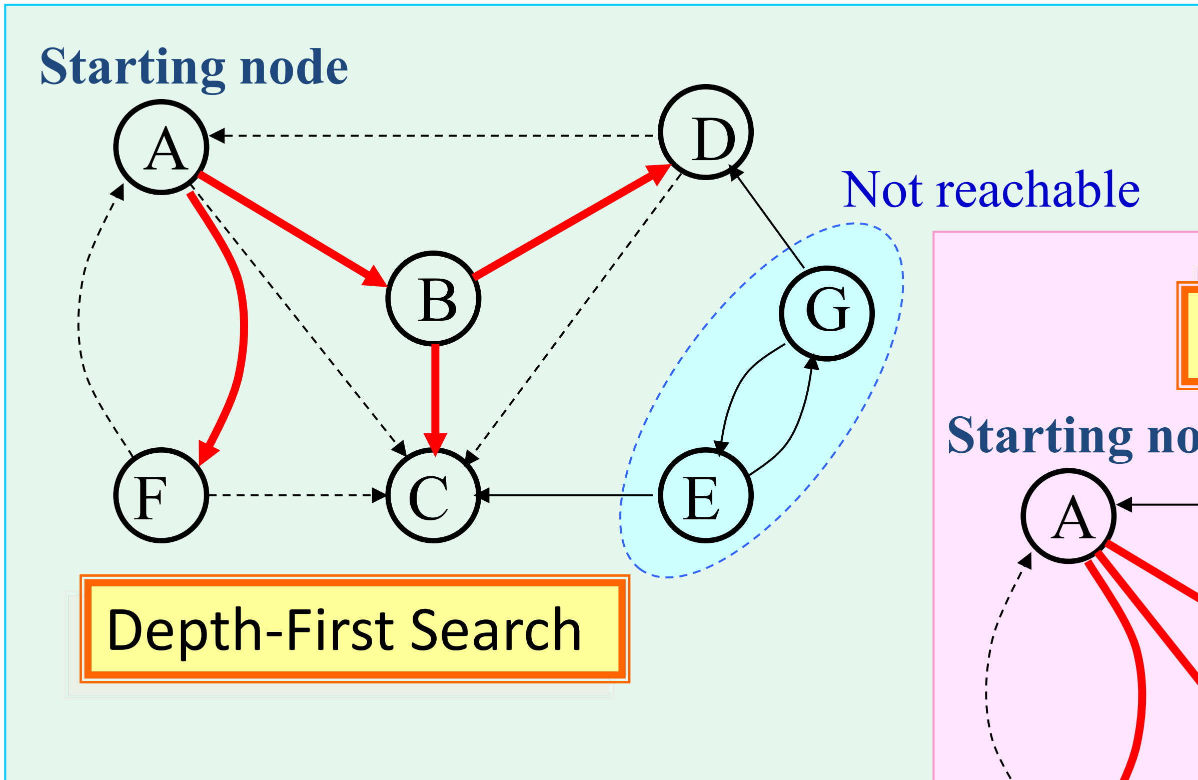
图顶点的可达性（不确定性算法）

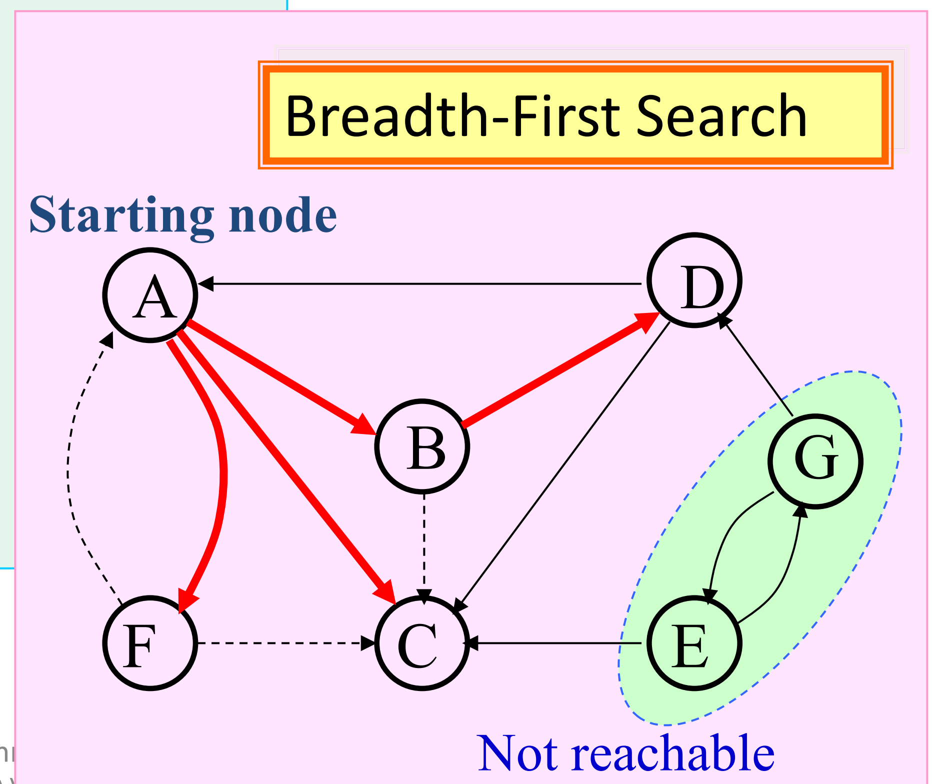
- 问题：给定一个图G和G的一个顶点u，问哪些顶点是从u可达的？
- 闭包形式的定义：
  - 初始值：u本身是可达的
  - 增量：如果顶点v可达，且G中存在边(v,w)，那么w也可达。

```
基本算法：
Reachable = { }
ToBeExplored = {u}
While(ToBeExplored非空)
{
    x = an element in ToBeExplored;
    ToBeExplored = ToBeExprlored – {x};
    Reachable = Reachable + {x};
     S = x的相邻顶点集合 – Reachable - ToBeExplored;
     ToBeExplored = ToBeExprlored + S;
}
```

根据x的选择方式，有不同的遍历计算方法

典型的Graph Traversal方法





- - - - - - - - - *Edges only "checked"*

Outline of Depth-First Search

dfs(G,v)

    Mark v as "discovered".

    For each vertex w that edge vw is in G:

        If w is undiscovered:

            dfs(G,w)

        Otherwise:

            "Check" vw without visiting w.

    Mark v as "finished".

A vertex must in one of the status:

- undiscovered， discovered， Finished
- 可达的顶点将顺序经历上面三个状态

That is: exploring vw, visiting w, exploring from there as much as possible, and backtrack from w to v.

Outline of Breadth-First Search

Bfs(G,*s*)
  Mark *s* as "discovered";
  enqueue(pending,*s*);

  while (pending is nonempty)
    dequeue(pending, *v*);
    For each vertex *w* that edge *vw* is in G:
      If *w* is "undiscovered"
        Mark *w* as "discovered" and
          enqueue(pending, *w*)
    Mark *v* as "finished";

和基本算法的关系：

- Pending等价于ToBeExprlored
- Undiscovered：既不再ToBeExprlored中，也不在Reachable中
- Finished：在Reachable中

特点：
1、总是按照顶点离初始顶点的距离，从近到远进行遍历

Outline of Breadth-First Search

Bfs(G,*s*)
Mark *s* as "discovered";
enqueue(p

while (pe
dequeu

For eac
If *w* i
M

Mark *v*

和基本算法的关系：

- Pending等价于 ToBeExprlored

再
也不在

ble中

离初始
到远

如果图是无限的，对于能够从初始顶点s到达的顶点t，广度优先搜索总是能够找到t。

例如：

- 对于一阶逻辑，我们可以把公式看作图的顶点，并且如果从公式f1能够一步推导出f2，那么从f1到f2就有一条边。
- （这种图不能够用邻接表或者邻接矩阵表示，但是从一个顶点出发能够找到它的所有后继顶点）
- 如果某个公式t能够从s推导得到，那么我们就可以在上面的图中从s出发，以广度优先搜索的方法找到t，表明t是可以证明的。

Finding Connected Components（1）

- Input: a symmetric（对称）digraph G, with n nodes and $2m$ edges(interpreted as an undirected graph), implemented as a array *adjVertices*[1,…*n*] of adjacency lists.
- Output: an array *cc*[1..*n*] of component number for each node v_i
  - 即cc[i]表示了i所在连通子图的编号

基本思想：遍历每个顶点v

1. 对每一个尚未确定所在连通子图的顶点v，通过DFS寻找到和v连通的所有顶点，对于找到的每个顶点w，设置w的连通子图编号为v。
2. 如果顶点v在之前的DFA搜索中已经出现在某个连通子图中就不需要在遍历。

Finding Connected Components（2）

```
void connectedComponents(Intlist[ ] adjVertices, int n,  int[ ] cc)
int[ ] color=new int[n+1];
    <Initialize color array to white for all ertices>
    for (int v=1; v≤n; v++)
        if (color[v]==white) //v尚未被搜索过
            //从v出发搜索v所在的连通子图的顶点，
            //并将这些顶点的连通子图编号标记为v
            ccDFS(adjVertices, color, v, v, cc);
    return
```

红色的v是出发顶点，
蓝色的v是子图的编号

ccDFS: the procedure

```
void ccDFS(IntList[ ] adjVertices, int[ ]
        color, int v, int ccNum, int [ ] cc)
    int w;
    IntList remAdj;

    color[v]=gray; //discovered

    cc[v]=ccNum; //附加的操作

    //通过remAdj来遍历相邻顶点
    remAdj=adjVertices[v];
    while (remAdj≠nil)
        w=first(remAdj);
        if (color[w]==white) //undiscovered
            ccDFS(adjVertices, color, w, ccNum, cc);
        remAdj=rest(remAdj);
    color[v]=black;//finished
    return
```

```
//抽象的DFS框架
dfs(G,v)
    Mark v as "discovered".
    For each vertex w that edge vw is in G:
        If w is undiscovered:
            dfs(G,w)
        Otherwise:
            "Check" vw without visiting w.
    Mark v as "finished".
```

White	:	undiscovered
Grey	:	discovered
Black	:	finished

1、如果G是通过邻接矩阵表示的，ccDFS如何修改？

2、可以通过BFS框架来完成吗？

Analysis of CC Algorithm

- connectedComponents, the wrapper (即主函数)
  - Linear in n (color array initialization + for loop on *adjVertices*)
  - ccDFS所需时间在下面计算
- ccDFS, the depth-first searcher
  - In one execution of ccDFS on v, the number of instructions(rest(*remAdj*)) executed is proportional to the size of *adjVertices*[v].
  - Note: Σ(size of *adjVertices*[v]) is $2m$, and the adjacency lists are traversed **only once**.
- So, the time complexity is in $\Theta(m+n)$
  - Extra space requirements:
    - Color array
    - Activation frame stack for recursion

如果G是通过邻接矩阵表示的，效率如何？

Visits On a Vertex

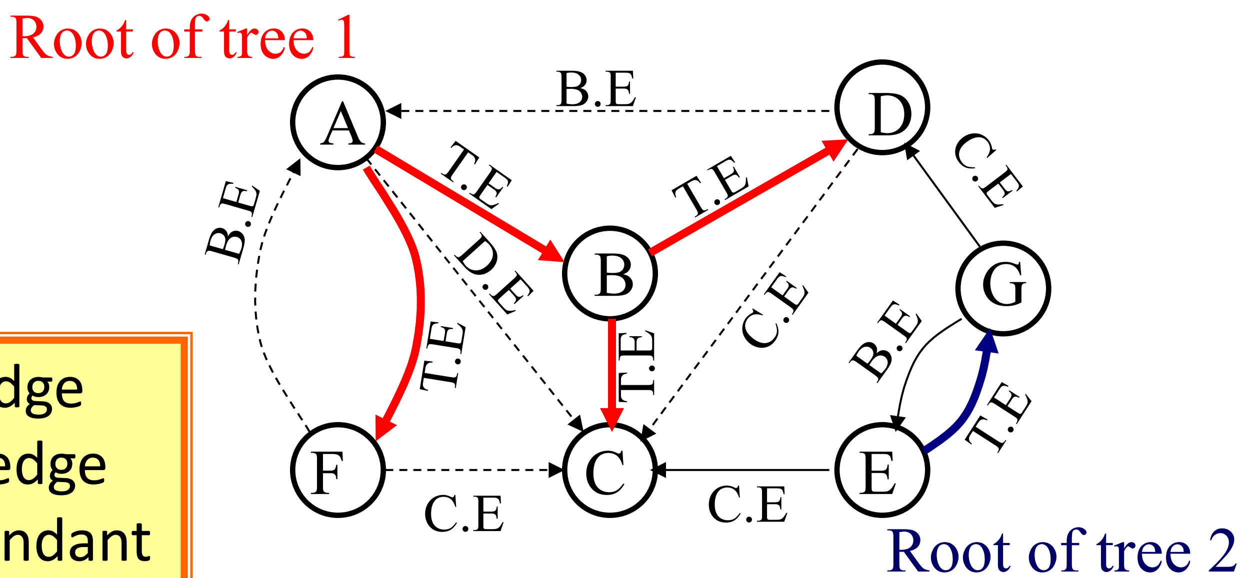
- Classification for the visits on a vertex
  - First visit(exploring): status: white→gray
  - (Possibly) multi-visits by backtracking to: status keeps gray
  - Last visit(no more branch-finished): status: gray→black

- Different operations can be done, during the different visits on a specific vertex
  - On the vertex
  - On (selected) incident edges

对于有些处理，我们需要考虑访问的顺序

Depth-first Search Trees

DFS forest={(DFS tree1), (DFS tree2)}



T.E: tree edge
B.E: back edge
D.E: descendant edge
C.E: cross edge

A finished vertex is never revisited, such as C

Depth-First Search – Generalized Skeleton

Input: Array *adjVertices* for graph G

Output: Return value depends on application.

int dfsSweep(IntList[] *adjVertices*,**int** n, …)

    int ans;

    <Allocate color array and initialize to white>

    For each vertex *v* of G, in some order

        if (color[v]==white)

            int vAns=**dfs(*adjVertices*, color, v, …)**;

            <Process vAns>

        // Continue loop

    return ans;

DFS– Generalized Skeleton

```
int dfs(IntList[] adjVertices, int[] color, int v, …)
    int w;
    IntList remAdj;
    int ans;
    color[v]=gray;
    <Preorder processing of vertex v>
    remAdj=adjVertices[v];
    while (remAdj≠nil)
        w=first(remAdj);
        if (color[w]==white)
            <Exploratory processing for tree edge vw>
            int wAns=dfs(adjVertices, color, w, …);
            < Backtrack processing for tree edge vw , using wAns>
        else
            <Checking for nontree edge vw>
        remAdj=rest(remAdj);
    <Postorder processing of vertex v, including final computation of ans>
    color[v]=black;
    return ans;
```

If partial search is used for a application, tests for termination may be inserted here.

Specialized for connected components:
- parameter added
- preorder processing inserted – cc[v]=ccNum

DFS– Generalized Skeleton

```
int dfs(IntList[] adjVertices, int[] color, int v, …)
    int w;
    IntList remAdj;
    int ans;
    color[v]=gray;
    <Preorder processing of vertex v>
    remAdj=adjVe
    while (remAd
        w=first(rem
        if (color[w]
            <Exploratory processing for tree edge vw>
            int wAns=dfs(adjVertices, color, w, …);
            < Backtrack processing for tree edge vw , using wAns
        else
            <Checking for nontree edge vw>
        remAdj=rest(remAdj);
    <Postorder processing of vertex v, including final computation of ans>
    color[v]=black;
    return ans;
```

If partial search is used for a application, tests for termination may be inserted here.

如何使用这个Skeleton来处理下列问题：

1、如果这个图是一棵树，计算各棵子树的结点数量

2、设置各个顶点在DFS树中的父结点？

nts:

• preorder processing inserted – cc[v]=ccNum

Breadth-First Search - Skeleton

Input: Array *adjVertices* for graph G
Output: Return value depends on application.

void bfsSweep(IntList[] *adjVertices*,**int** n, …)
  int ans;
  <Allocate color array and initialize to white>
  For each vertex *v* of G, in some order
    if (color[v]==white)
      void bfs(*adjVertices*, color, v, …);
  // Continue loop
  return;

Breadth-First Search - Skeleton

```
void bfs(IntList[] adjVertices, int[] color, int v, …)
  int w; IntList remAdj; Queue pending;
  color[v]=gray;    enqueue(pending, v);
  while (pending is nonempty)
    w=dequeue(pending); remAdj=adjVertices[w];
    while (remAdj≠nil)
      x=first(remAdj);
      if (color[x]==white)
        color[x]=gray; enqueue(pending, x);
      remAdj=rest(remAdj);
    <processing of vertex w> //附加处理
    color[w]=black;
  return ;
```

对于使用邻接表表示的图的BFS处理

DFS vs. BFS Search

- Processing opportunities for a node
  - Depth-first: 2
    - At discovering（刚刚开始访问，尚未访问它的后代，但是node所到达的结点可能已经被访问了）
    - At finishing （所有后代结点都访问完毕）
  - Breadth-first: only 1, when de-queued
  - At the second processing opportunity for the DFS, the algorithm can make use of information about the descendants of the current node.

Time Relation on Changing Color

- Keeping the order in which vertices are encountered for the first or last time
  - A global interger time: 0 as the initial value, incremented with each color changing for *any* vertex, and the final value is $2n$
  - Array *discoverTime*: the i th element records the time vertex v_i turns into gray
  - Array *finishTime*: the i th element records the time vertex v_i turns into black
  - The active interval for vertex v, denoted as *active*(v), is the duration while v is gray, that is:

    discoverTime[v], …, *finishTime*[v]
  - Active interval中间的顶点表示了DFS(v)所访问的顶点

```
int dfs(IntList[] adjVertices, int[] color, int v, …)
   int w;
   IntList remAdj;
   int ans;
   color[v]=gray;
   <Preorder processing of vertex v>
   remAdj=adjVertices[v];
   while (remAdj≠nil)
      w=first(remAdj);
      if (color[w]==white)
         <Exploratory processing for tree edge vw>
         int wAns=dfs(adjVertices, color, w, …);
         < Backtrack processing for tree edge vw , using wAns>
      else
         <Checking for nontree edge vw>
      remAdj=rest(remAdj);
   <Postorder processing of vertex v, including final computation of ans>
   color[v]=black;
   return ans;
```

Depth-First Search Trace

General DFS skeleton modified to compute discovery and finishing times and "construct" the depth-first search forest.

int dfsTraceSweep(IntList[ ] *adjVertices*,**int** n, **int**[ ] *discoverTime*, **int**[ ] *finishTime*, **int**[ ] *parent*)

    int ans; **int** *time*=0

    <Allocate color array and initialize to white>

    For each vertex *v* of G, in some order

        if (color[v]==white)

            parent[v]=-1

            int vAns=dfsTrace(*adnVertices*, color, v, *discoverTime*, *finishTime*, *parent*, *time* **);**

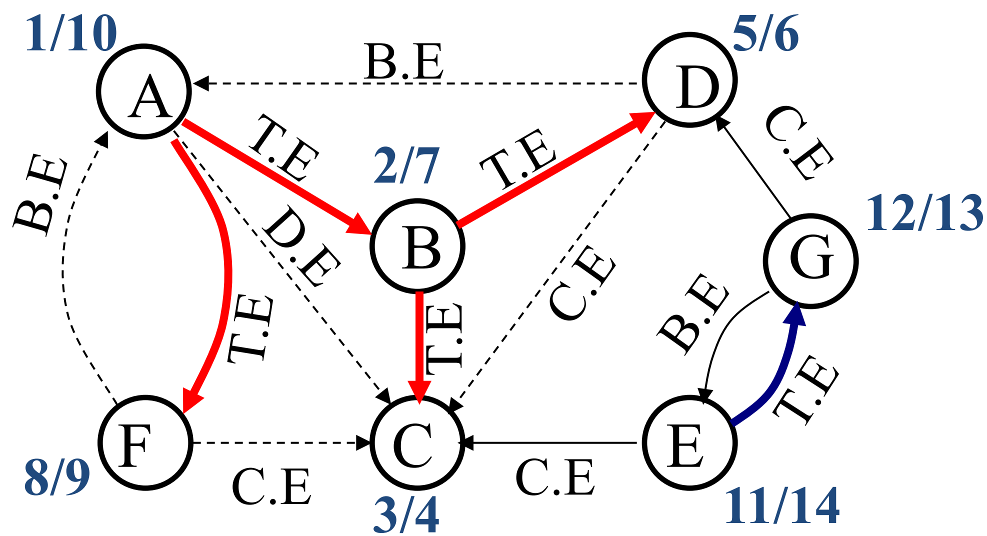
        // Continue loop

    return ans;

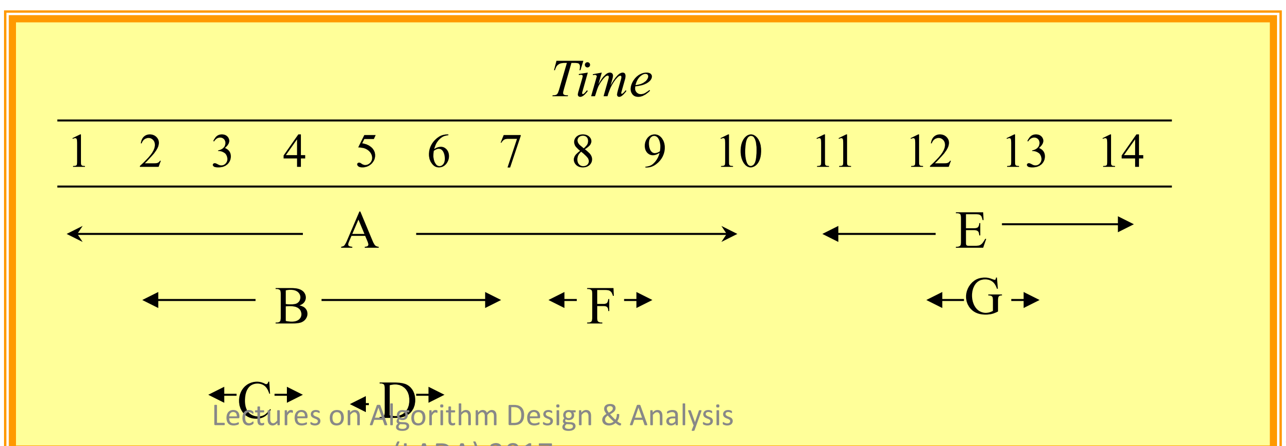
Depth-First Search Trace

```
int dfsTrace(intList[ ] adjVertices, int[ ] color, int v,  int[ ] discoverTime,
                int[ ] finishTime, int[ ] parent  int time)
  int w; IntList remAdj; int ans;
  color[v]=gray; time++; discoverTime[v]=time; //preorder of nodes
  remAdj=adjVertices[v];
  while (remAdj≠nil)
    w=first(remAdj);
    if (color[w]==white)
      parent[w]=v;  (边的处理)
      int wAns=dfsTrace(adjVertices, color, w, discoverTime, finishTime, parent, time);
    else  <Checking for nontree edge vw>
    remAdj=rest(remAdj);
  time++; finishTime[v]=time; color[v]=black; //post order of nodes
  return ans;
```

Active Interval

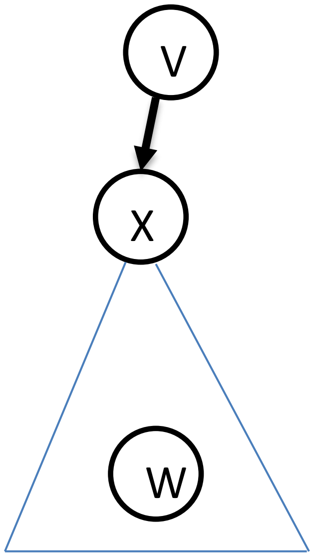


The relations are summarized in the next frame



Properties of Active Intervals(1)

- If w is a descendant of v in the DFS forest, then $active(w) \subseteq active(v)$, and the inclusion is proper if $w \neq v$.
- **Proof:**
  - Define a partial order <: w<v iff. w is a proper descendants of v in its DFS tree. The proof is by induction on <.
  - BASE:
    - If v is minimal. The only descendant of v is itself. Trivial.
  - INDUCTION:
    - Assume that for all x<v, if w is a descendant of x, then active(w)⊆active(x).
    - Let w be any proper descendant of v in the DFS tree, there must be some x such that vx is a tree edge on the tree path to w, so w is a descendant of x. According to dfsTrace, we have $active(x) \subset active(v)$, by inductive hypothesis, $active(w) \subset active(v)$.



Properties of Active Intervals(2)

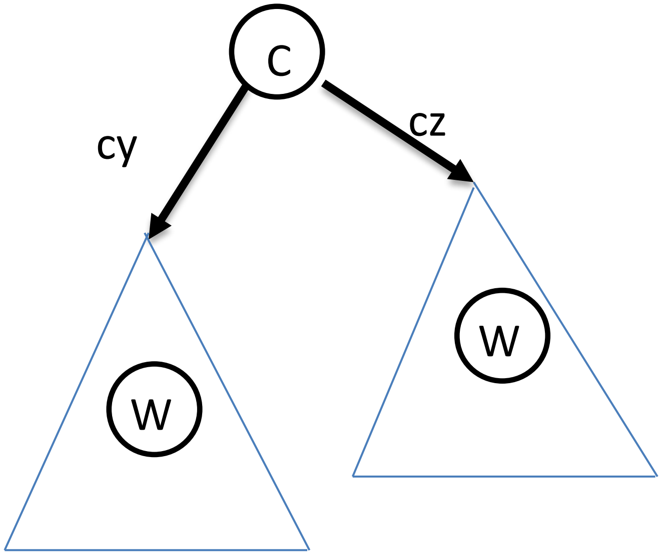
- If $active(w) \subseteq active(v)$, then w is a descendant of v. And if $active(w) \subset active(v)$, then w is a proper descendant of v.

That is: *w* is discovered while *v* is active.

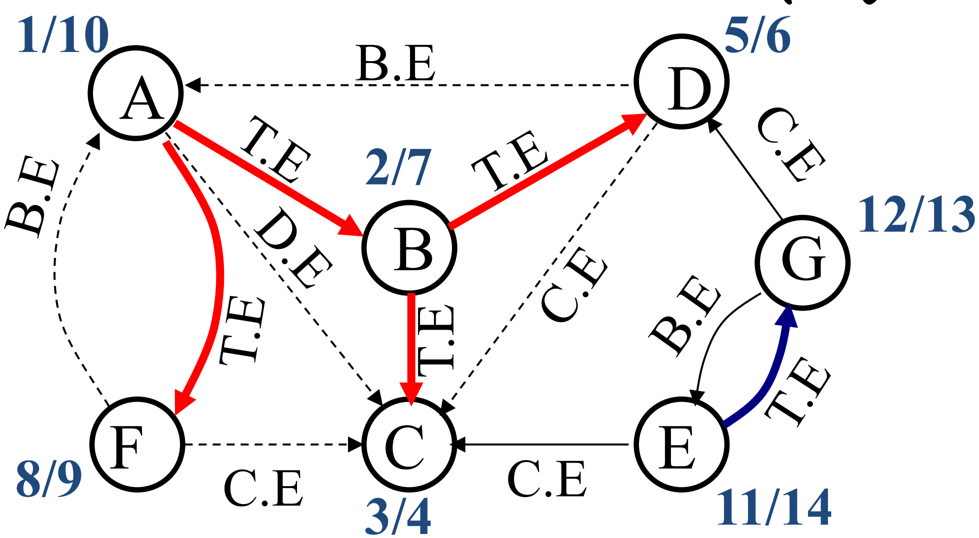
- **Proof:**
  - If w is **not** a descendant of v, there are two cases:
    - v is a proper descendant of w, then $active(v) \subset active(w)$, so, it is impossible that $active(w) \subseteq active(v)$, contradiction.
    - There is no ancestor/descendant relationship between v and w, then $active(w)$ and $active(v)$ are disjoint, contradiction.

Properties of Active Intervals(3)

- If v and w have no ancestor/descendant relationship in the DFS forest, then their **active intervals** are disjoint.
- Proof:
  - If v and w are in different DFS tree, it is trivially true, since the trees are processed one by one.
  - Otherwise, there must be a vertex c, satisfying that there are tree paths c to v, and c to w, without edges in common. Let the leading edges of the two tree path are cy, cz, respectively. According to dfsTrace, *active*(y) and *active*(z) are disjoint.
  - We have *active*(v)$\subseteq$*active*(y), *active*(w)$\subseteq$*active*(z). So, *active*(v) and *active*(w) are disjoint.



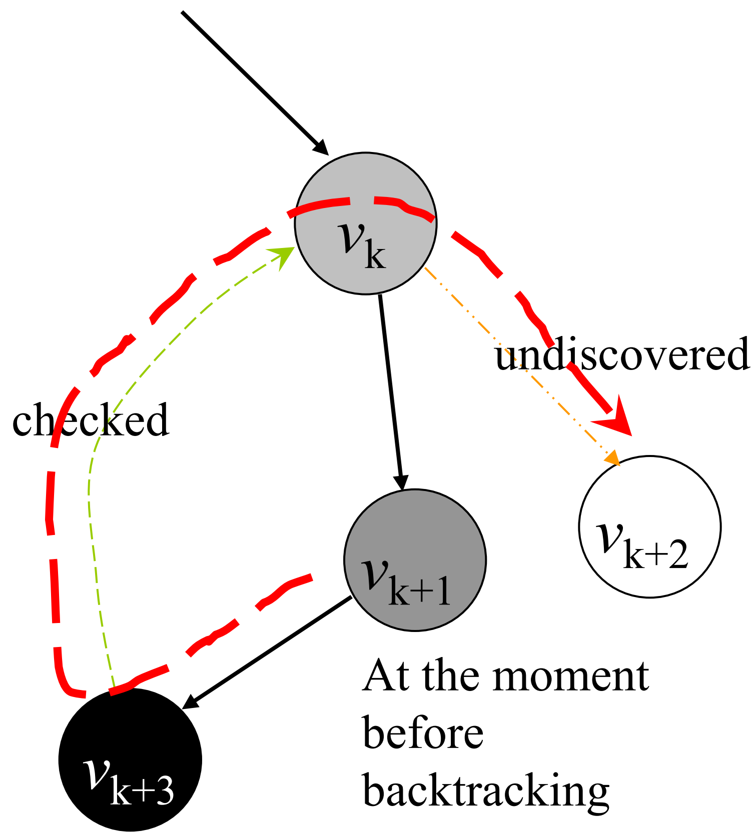
Properties of Active Intervals(4)



- If edge $vw \in E_G$, then
  - vw is a **cross edge** iff. *active*(w) entirely precedes *active*(v).
  - vw is a **descendant edge** iff. there is some third vertex *x*, such that *active*(w)$\subset$*active*(*x*)$\subset$*active*(*v*),
  - vw is a **tree edge** iff. *active*(w)$\subset$*active*(*v*), and there is no third vertex *x*, such that *active*(w)$\subset$*active*(*x*) $\subset$*active*(*v*),
  - vw is a **back edge** iff. *active*(v)$\subset$*active*(w),

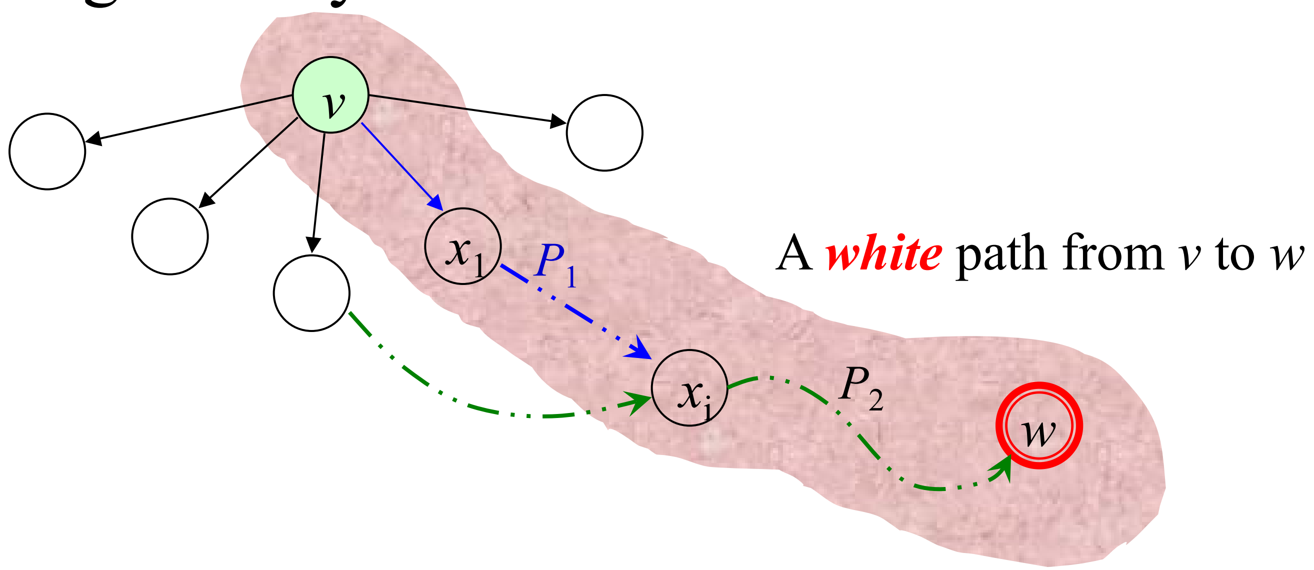
Ancestor and Descendant

- That w is a descendant of v in the DFS forest means that there is a direct path from v to w in some DFS tree.
- The path is also a path in G.
- However, if there is a direct path from v to w **in G**, is w necessarily a descendant of v in ***the*** DFS forest?



DFS Tree Path

- [White Path Theorem] w is a descendant of v in a DFS tree iff. at the time v is discovered(just to be changing color into gray), there is a path in G from v to w consisting entirely of white vertices.



Proof of White Path Theorem

- $\Rightarrow$ All the vertices in the path are descendants of v.
- $\Leftarrow$ All the descendants of v are in some white path
- Proof: by induction on the length *k* of white path from v to w.
  - When k=0, v=w.
  - For k>0, let P=(v, x_1,x_2,…x_k=w). There must be some vertex on P which is discovered during the active interval of v, e.g. x_1, Let x_i is earliest discovered among them. Divide P into P_1 from v to x_i, and P_2 from x_i to w. P_2 is a white path with length less than k, so, by inductive hypothesis, w is a descendant of x_i. Note: $active(x_i) \subseteq active(v)$, so x_i is a descendant of v. By transitivity, w is a descendant of v.

Thank you!

Q & A

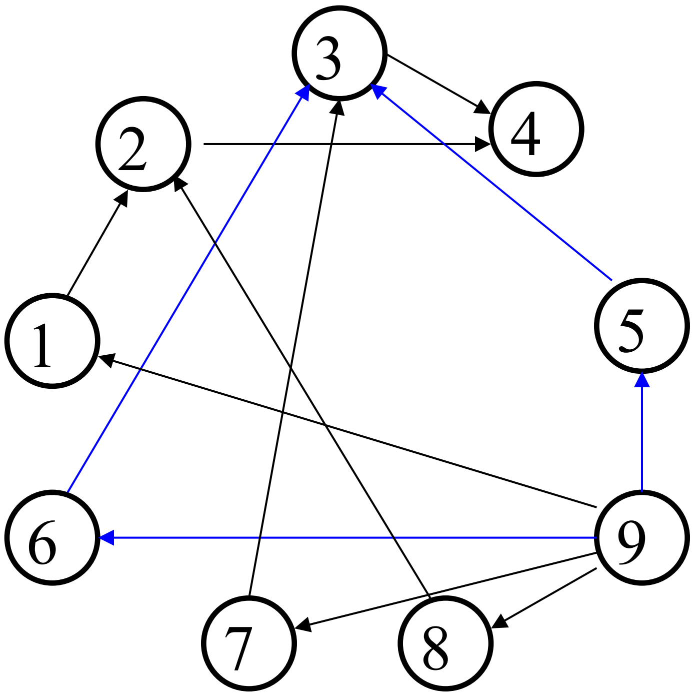
Yu Huang
http://cs.nju.edu.cn/yuhuang

DFS的应用：DAG图和SCC

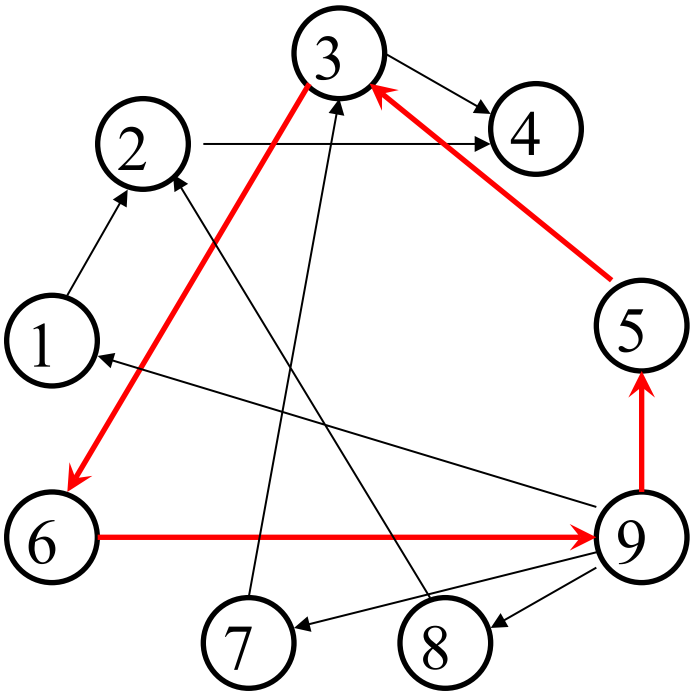
赵建华

南京大学计算机系

Directed Acyclic Graph (DAG)
有向无环图



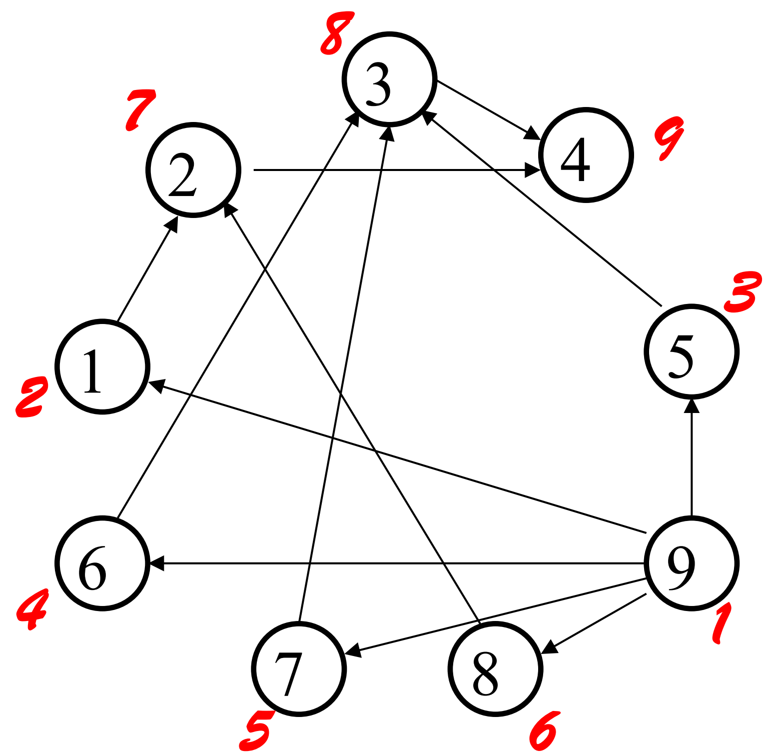
A Directed Acyclic Graph



Not a DAG

Topological Order for G=(V,E)

- Topological number
  - An assignment of distinct integer 1,2,…, n to the vertices of V
  - For every $vw \in E$, the topological number of v is less than that of w.
  - 一个DAG图可以具有多个不同的排序
- Reverse topological order
  - Defined similarly ("greater than")



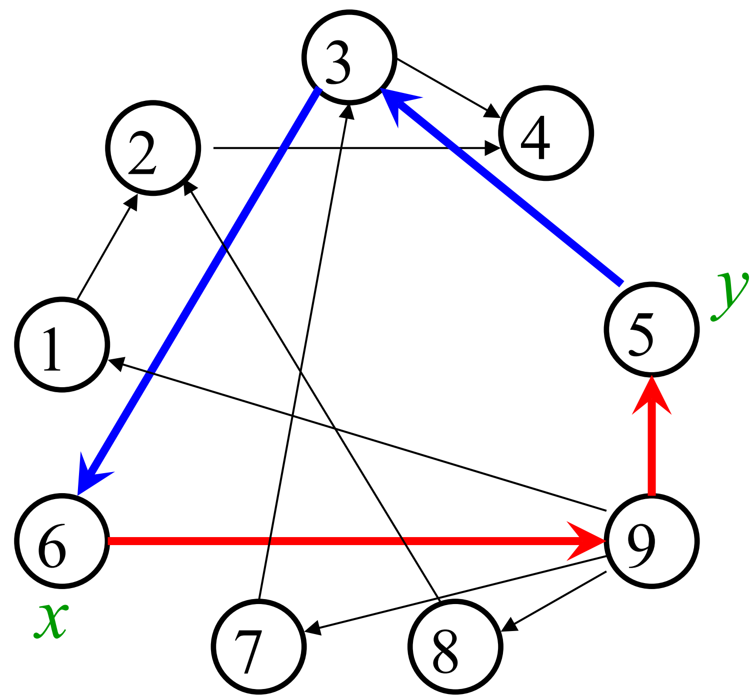
Existence of Topological Order
– a Negative Result

- If a directed graph G has a cycle, then G has no topological order
- Proof
  - [By contradiction]

yx-path

xy-path

For any given topological order, all the vertices on both paths must be in increasing order. Contradiction results for any assignments for x and y.



DFS框架的性质（回顾）

- 图的DFS过程中有两次处理顶点的机会：
  - 顶点状态由undiscovered（white）变成discovered（black）时进行处理（Preorder）
  - 顶点状态由discovered（gray）变成finished（black）时进行处理，（Postorder）
    - 此时顶点的后继顶点的状态要么是finished（black）状态，要么是discovered（gray）状态
    - 如果邻接顶点是discovered（gray）状态，那么此邻接顶点和当前顶点形成一个回路

使用DFS框架处理问题

- 如果一个问题能够转化为如下的形式，那么可以考虑在DFS框架下进行处理
  - 问题的目标是求解各个顶点v对应的值F(v)
  - 如果v的邻接顶点是$u_1,u_2,\ldots,u_k$，那么F(v)可以根据$F(u_1)$, $F(u_2),\ldots,F(u_k)$得到

- 处理的方法大致如下：
  - 按照PostOrder进行处理（也就是顶点由discovered变成finished时进行处理）
  - 函数增加参数来记录各个顶点对应的F值，同时可能需要一些新参数来记录相关信息
  - 在PostOrder进行处理，计算F(v)

- 如果PostOrder处理的时间复杂性是O(1)或者O(k)的(注意所有结点的k值加起来就是边的数量)，那么算法的复杂性就是O(m+n)的，其中m是顶点数，n是边数。

Reverse Topological Ordering

- 问题可以转换成为
  - 给每个顶点v赋予逆拓扑编号，满足每个顶点的编号不同，且v的编号大于v的邻接顶点的编号
  - 通过全局变量topoNum实现：每次赋予一个顶点编号，就将topoNum加一。因为v的邻接顶点先被赋值，自然v的编号是唯一的，且大于邻接顶点的编号

- Specialized parameters
  - Array *topo*, keeps the topological number assigned to each vertex.
  - Global variable *topoNum* to provide the integer to be used for topological number assignments
- Output
  - Array *topo* as filled.

Reverse Topological Ordering

```
int topoNum=0

void dfsTopoSweep(IntList[ ] adjVertices,int n, int[ ] topo)
    <Allocate color array and initialize to white>
    For each vertex v of G, in some order
        if  (color[v]==white)
            dfsTopo(adjVertices, color, v, topo);
        // Continue loop
    return;
```

Reverse Topological Ordering

```
void dfsTopo(IntList[] adjVertices, int[] color, int v, int[ ] topo)
    int w; IntList remAdj; color[v]=gray;  remAdj=adjVertices[v];
    while (remAdj≠nil)
        w=first(remAdj);
        if (color[w]==white)
            dfsTopo(adjVertices, color, w, topo, topoNum);
        remAdj=rest(remAdj);
        topoNum++; topo[v]=topoNum
    color[v]=black;
    return;
```

Obviously, in $\Theta(m+n)$

Filling *topo* is a post-order processing, so, the earlier discovered vertex has relatively greater topo number

思考：
如果在过程中判断图中是否存在环路？

Correctness of the Algorithm

- If G is a DAG with n vertices, the procedure *dfsTopoSweep* computes a reverse topological order for G in the array *topo*.
- Proof
  - The procedure dfsTopo is called exactly once for a vertex, so, the numbers in *topo* must be distinct in the range 1,2,…n.
  - For any edge vw, vw can't be a back edge(otherwise, a cycle is formed). For any other edge types, we have *finishTime*(v)>*finishTime*(w), so, *topo*(w) is assigned earlier than *topo*(v). Note that *topoNum* is incremented monotonically, so, *topo*(v)>*topo*(w).

Existence of Topological Order

- In fact, the proof of correctness of topological ordering has proved that: DAG always has a topological order.

- So, G has a topological ordering, iff. G is a directed acyclic graph.

另一个DAG图的拓扑排序(1)

拓扑排序的性质

- 一个顶点v的入度：G中wv边的数量
- 如果一个DAG图G中的顶点n的入度为0，那么n肯定可以排在第一个；
- 将n放在第一个位置后，从G中删除n得到G’，将G中的顶点拓扑排序后放在n之后，就得到G的拓扑排序

```
Topo(G)
{
    如果G是空图，返回< >;
    找出一个入度为0的节点v; （如果找不到表示图不是DAG图）
    G’ = G - v; //G’是从G中删除v以及相关边后得到的图
    return <n> + Topo(G’)
}
```

上面这个算法中，我们只需要在图中寻找入度为0的顶点，对于图中的边如何指向并不关心。

因此，我们不需要真的在G中删除n.

只需要能够计算G’中各个顶点的入度

另一个DAG图的拓扑排序(2)

- 基于入度信息进行拓扑排序：
  - 首先统计G中各个顶点的入度；
  - 根据入度信息找到一个入度为0的v；
  - 对于离开v的每一条出边(v,w)，将w的入度减去1，则得到G'中各个顶点的入度信息。
  - …
- 如何快速找到入度为0的结点？
  - 对于任意不同于v的顶点w，如果w在G中的入度为0，那么w在G'中的入度是多少？
  - 在G中入度不为0，但是在G'中入度为0的顶点在什么时候可以发现？
  - 可以建立一个队列queue，当结点的入度变成0时，加入到这个队列中。寻找入度为0的顶点时直接在queue中取即可

另一个DAG图的拓扑排序(3)

```
topoList =<>; queue =<>;
对于G中的每个顶点i， inDegrees[i] = 0;    //O(m)
对于G中的每条边(v,w), inDegrees[m] ++;  //O(n)
对于inDegrees[i]==0的结点i, queue.add(i); //O(m)
while(!queue.isEmpty()){
    v = queue.deQueue();
    topoList = topoList +<v>;
    for(v的每条出边(v,w))
    {
        inDegrees[w] --;
        if(inDegrees[w] == 0)
            queue.add(w);
    }
    return topoList;
}
```

另一个DAG图的拓扑排序(3)

```
topoList =<>; queue =<>;
对于G中的每个顶点i，  inDegrees[i] = 0;     //O(m)
对于G中的每条边(v,w), inDegrees[m] ++;  //O(n)
对于inDegrees[i]==0的结点i, queue.add(i); //O(m)
while(!queue.isEmpty()){
    v = queue.deQueue();
    topoList = topoList +<v>
    for(v的每条出边(v,w))
    {
        inDegrees[w] --;
        if(inDegrees[w] ==
            queue.add(v
    }
    return topoList;
}
```

实际上，topoList，queue和结点入度信息可以放置到同一个数组中，具体方法自己思考或者查资料。

提示：

1、queue中的顶点（入读为0的顶点），入度非零的顶点，以及topoList中的顶点是整个顶点集合的一个分划

2、顶点总是先在入度非零的顶点集合中，然后进入queue，最后进入topoList。

3、topoList和queue中的顶点互异，可以用链表表示

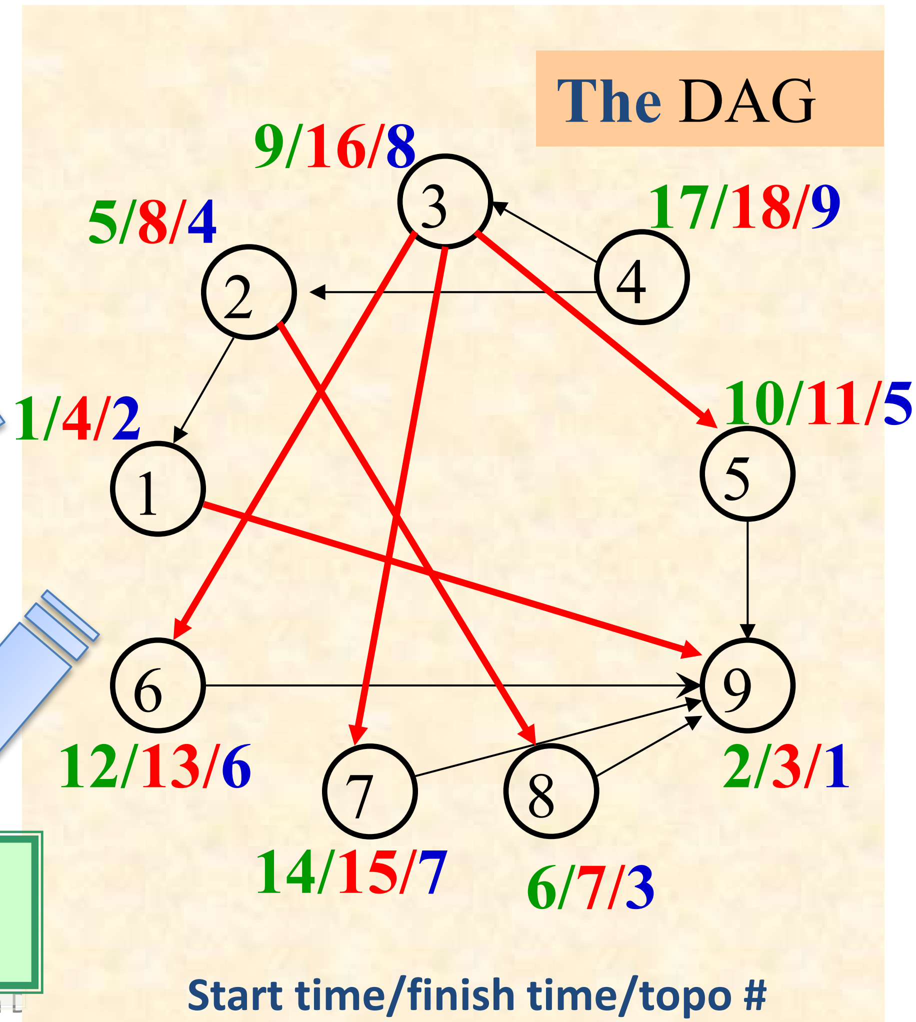
4、算法总是从queue中获取待处理的顶点

Task Scheduling

- Problem:
  - Scheduling a project consisting of a set of interdependent tasks to be done by one person.
- Solution:
  - Establishing a dependency graph, the vertices are tasks, and edge vw is included iff. the execution of v depends on the completion of w,
  - Making task scheduling according to the topological order of the graph(if existing).

Task Scheduling: an Example

Tasks(No.)	Depends on
choose clothes(1)	9
dress(2)	1,8
eat breakfast(3)	5,6,7
leave(4)	2,3
make coffee(5)	9
make toast(6)	9
pour juice(7)	9
shower(8)	9
wake up(9)	-



A reverse topological order

9, 1, 8, 2, 5, 6, 7, 3, 4

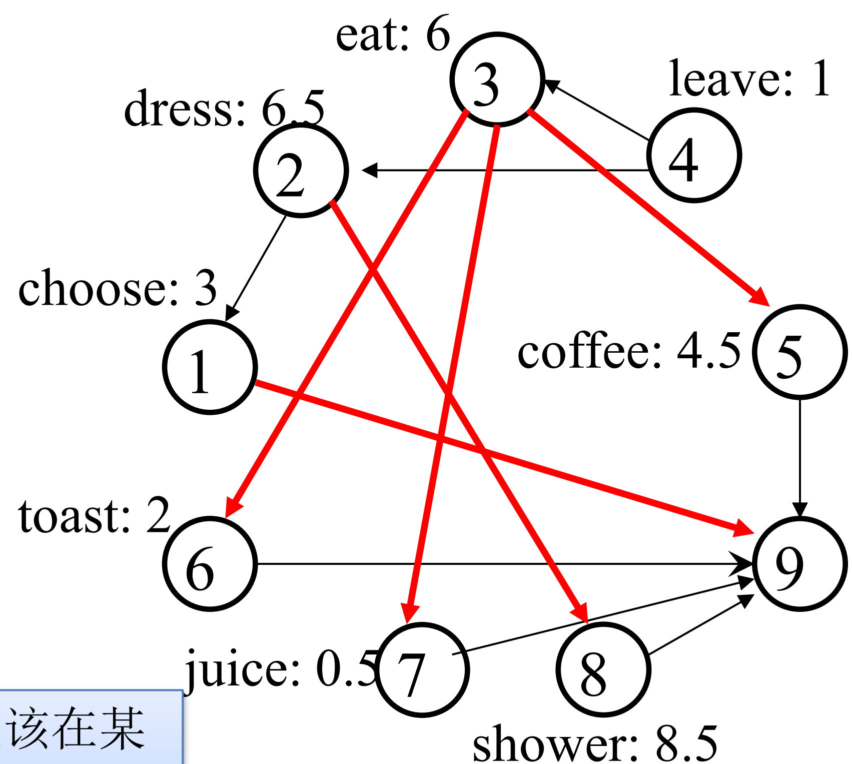
Project Optimization Problem

Assuming:

1. Parallel executions of tasks (v_i) are possible except for prohibited by interdependency.
2. Each task takes a given amount of time

Question:

What is the shortest time to finish the project?

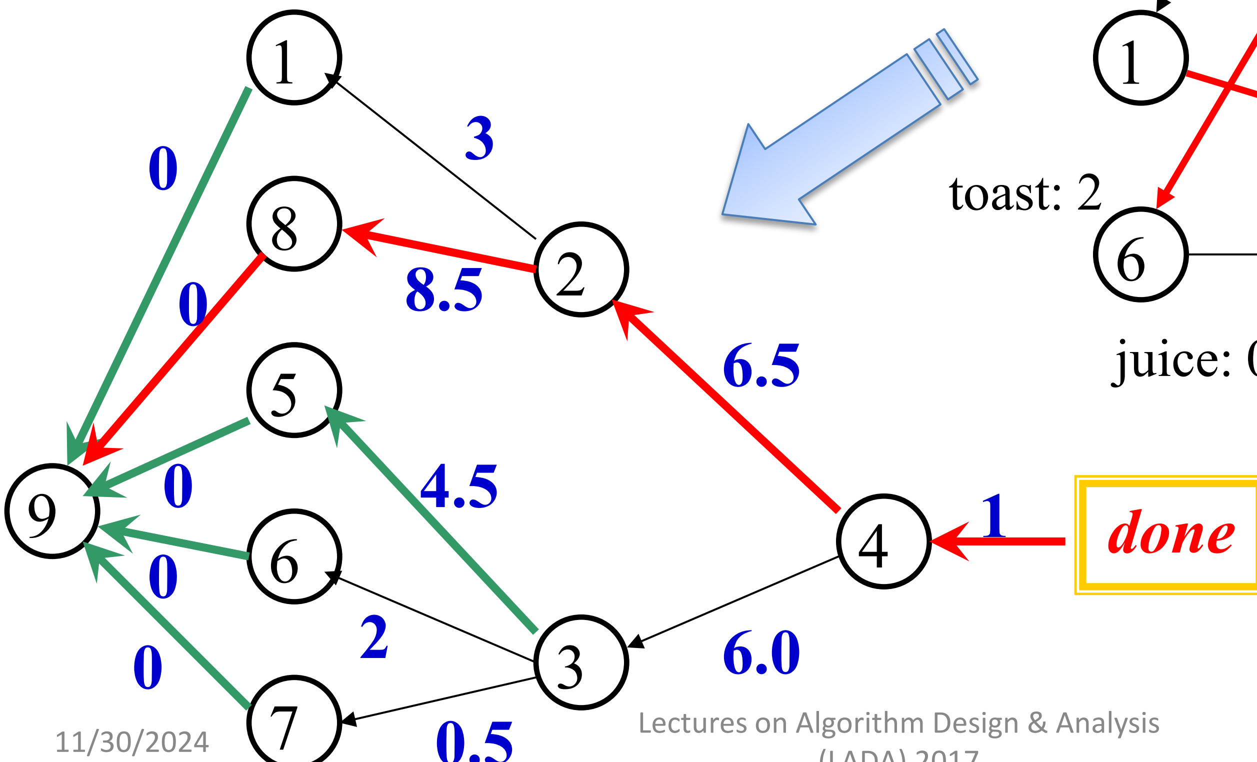


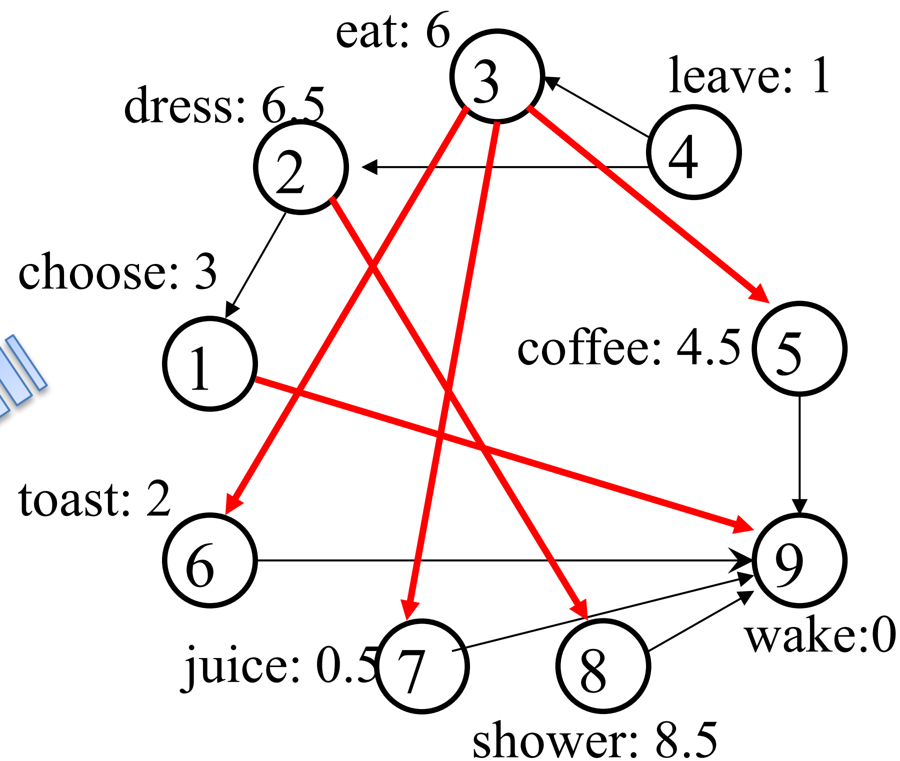
1、显然，要最快完成整个Project，那么应该在某个task可以开始时立刻开始；
2、但是，如果中间某个task需要更多时间了，或者需要适当推迟，会有什么影响？

DAG with Weights

Critical Path

Critical Subpath





左图中，每个结点实际上表示一个时间点，即‘开始做这件事情的时间’

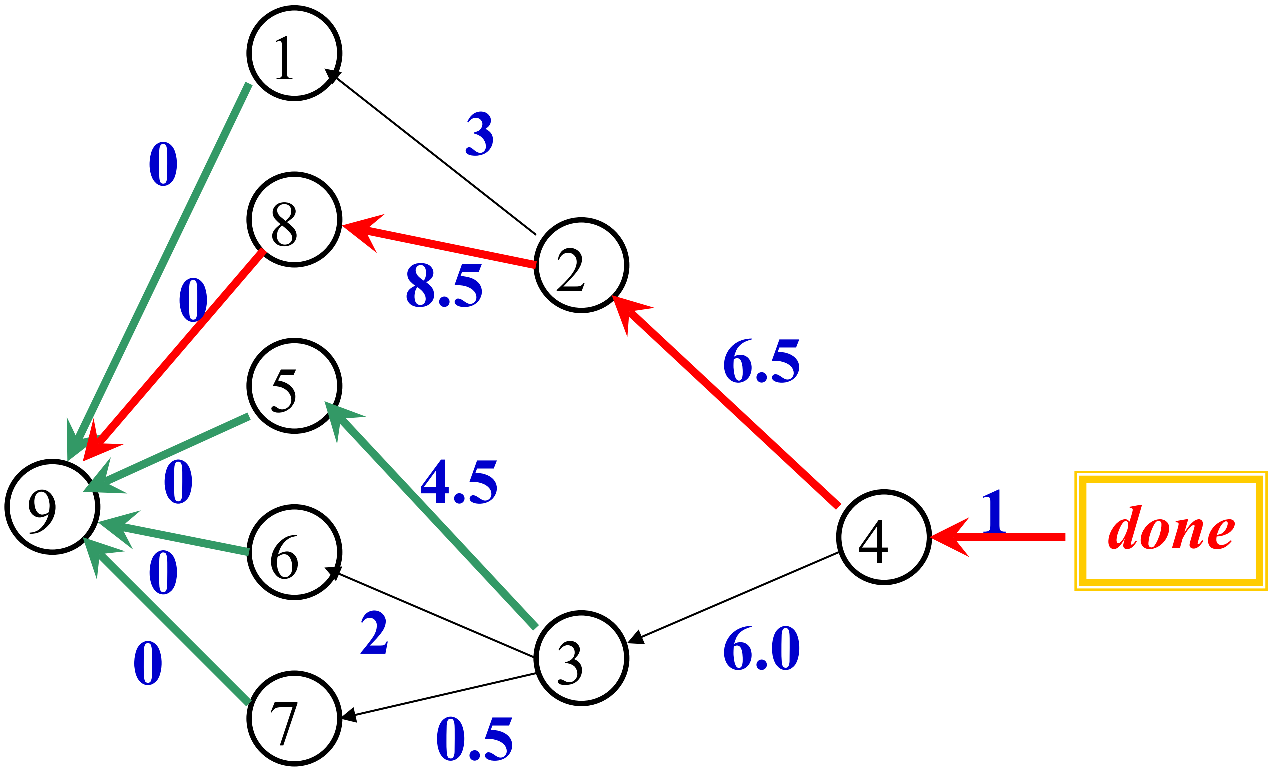
最早完成时间

- 每个活动的最早结束时间（earliest finish time, eft）等于该活动的最早开始时间(earliest start time, est)加上该活动所需时间（标记在边上）。
$$eft(v) = est(v) + duration(v)$$
- 每个活动v开始的时间不能早于它依赖的活动w的最早结束时间
$$est(v) = \max_{vw \in E(G)} eft(w)$$
- 各个结点的值可以按照逆拓扑序进行计算，对于每个结点，先计算est，再计算eft

- 因此，我们可以通过DFS遍历DAG图，并且
  - 对于结点v，每次处理完一个后继就跟新est(v)和eft(v)
  - 对结点进行postorder处理

- 进一步的问题：假如我们加快某些活动的进程，能不能使得整个Project提早完成？
  - 这些task称为关键活动，这些活动所在的路径称为关键路径。

Project Optimization Problem



- Observation
  - In a critical path, v_{i-1}, is a critical dependency of v_i, i.e. any delay in v_{i-1} will result in delay in v_i. (ETF(v_{i-1}) == Earlist(v_i))
  - Reducing the time of a off-critical-path task is of no help for reducing the total time for the project.
- The problem can be transform into: Find the critical path in a **DAG**

Critical Path in a Task Graph

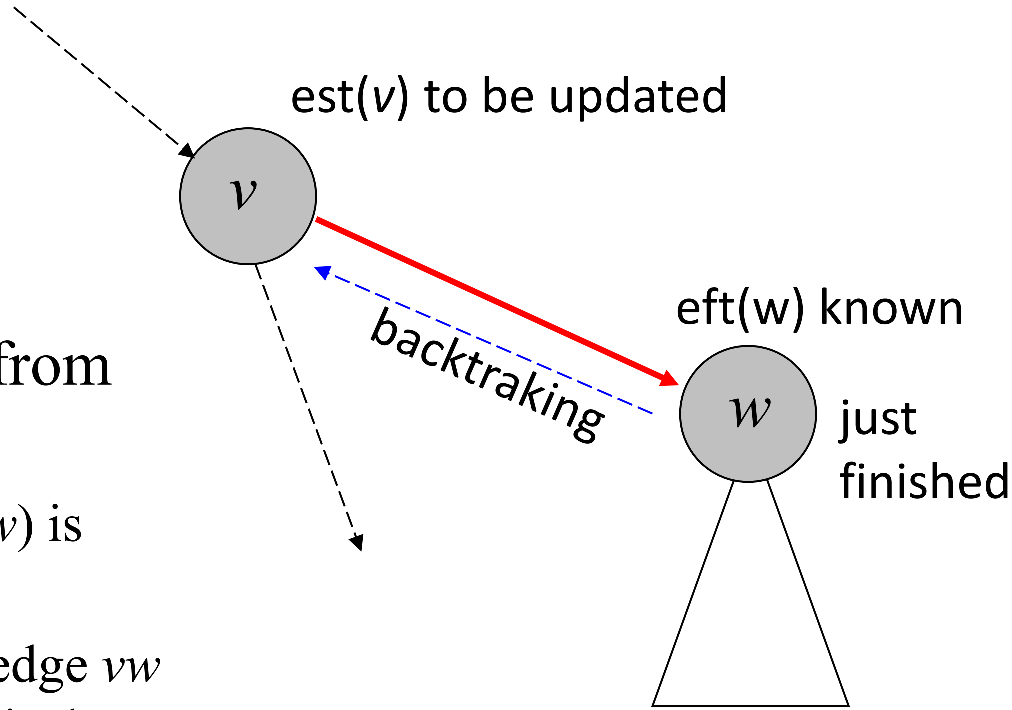
- Earliest start time(est) for a task v
  - If v has no dependencies, the *est* is 0
  - If v has dependencies, the *est* is the maximum of the earliest finish time of its dependencies.
- Earliest finish time(eft) for a task v
  - For any task: *eft* = *est* + *duration*
- Critical path in a project is a sequence of tasks: $v_0, v_1, \ldots, v_k$, satisfying:
  - v_0 has no dependencies;
  - For any $v_i (i=1,2,\ldots,k)$, v_{i-1} is a dependency of v_i, such that *est* of v_i equals *eft* of v_{i-1};
  - *eft* of v_k, is maximum for all tasks in the project.

Critical Path Finding - DFS

- Specialized parameters
  - Array *duration*, keeps the execution time of each vertex. (Input)
  - Array *eft*, keeps the earliest finished time of each vertex. (Output)
  - Array *critDep*, keeps the critical dependency of each vertex.
- Output
  - Array *topo*, *critDep*, *eft* as filled.

- Critical path is built by tracing the output.

Critical Path – Case 1



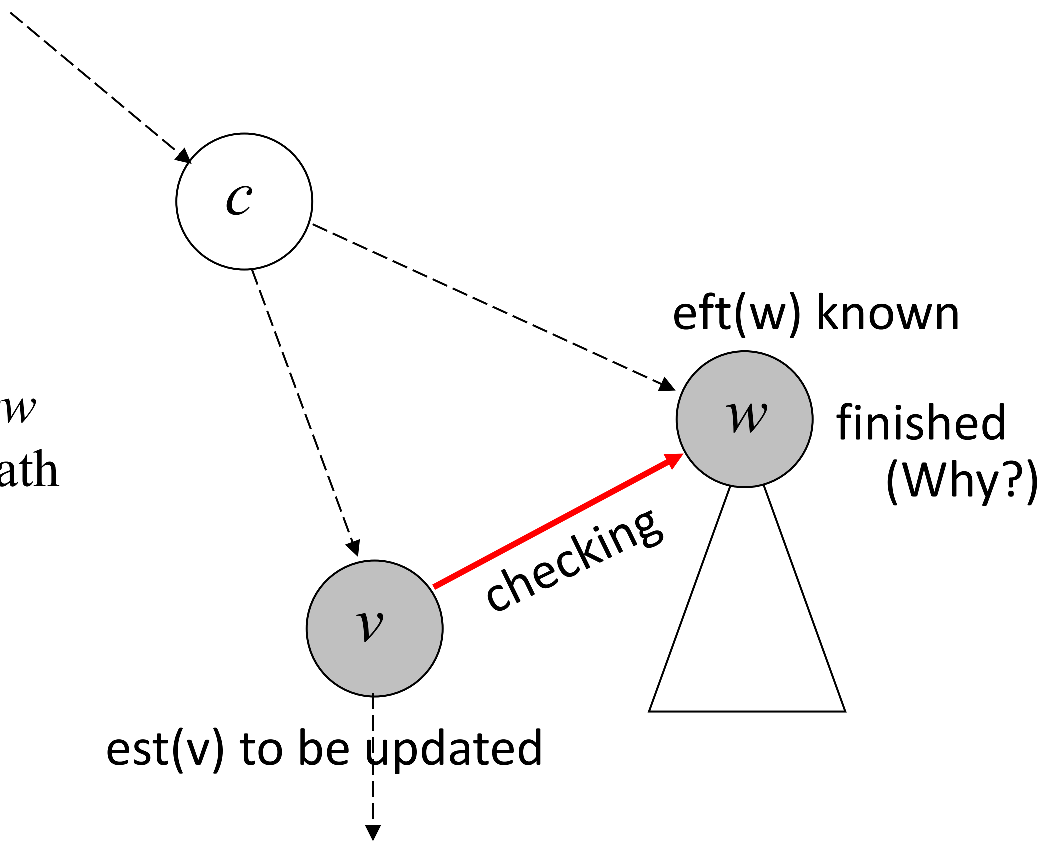
Upon backtracking from w:

- est(v) is updated if eft(w) is larger than est(v)
- and the path including edge vw is recognized as the critical path for tast v
- and the eft(v) is updated accordingly

Critical Path – Case 2

Checking w:

- est(v) is updated if eft(w) is larger than est(v)
- and the path including edge vw is recognized as the critical path for task v
- and the eft(v) is updated accordingly



Critical Path by DFS

```
void dfsCritSweep(IntList[ ] adjVertices,int n, int[ ] duration,
int[ ] critDep, int[ ] eft)
    <Allocate color array and initialize to white>
    For each vertex v of G, in some order
        if  (color[v]==white)
            dfsCrit(adjVertices, color, v, duration, critDep, eft);
        // Continue loop
    return;
```

Critical Path by DFS

```
//要求输入的图是一个DAG图
void dfsCrit(.. adjVertices,  .. color, .. v, int[ ] duration, int[ ] critDep, int[ ] eft)
    int w; IntList remAdj; int est=0;
    color[v]=gray; critDep[v]=-1; remAdj=adjVertices[v];
    while (remAdj≠nil) w=first(remAdj);
    {
        if (color[w]==white)
            dfsCrit(adjVertices, color, w, duration, critDep, efs);
            if (eft[w]≥est)
                {est=eft[w]; critDep[v]=w;}
        else//checking for nontree edge
            if (eft[w]≥est)
                {est=eft[w]; critDep[v]=w;}
        remAdj=rest(remAdj);
    }
    eft[v]=est+duration[v]; color[v]=black;
    return;
```

也可以按照Postorder顺序，在此处遍历v的所有后继，并计算est和eft[v]的值

进一步的问题

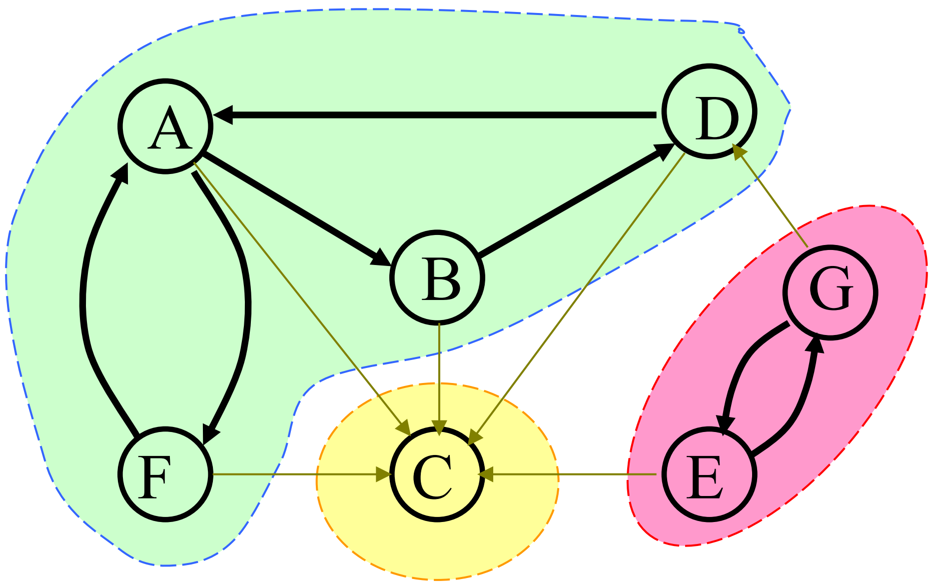
- 关键路径上的task不可拖延，但是其它task可以有所拖延。
- 问题：在不影响整个Project进度的前提下，这些路径最多拖延多久？
- 等价问题：
  - 最后的活动（done）结束时间不变的情况下，它的各个前驱活动最晚什么时候完成？
  - 如果某个活动v最晚开始时间是Latest(v)，它的前驱最晚可以多久开始？

Analysis of Critical Path Algorithm

- Correctness:
  - When *eft*[*w*] is accessed in the while-loop, the w must not be gray(otherwise, there is a cycle), so, it must be black, with *eft* initialized.
  - According to DFS, each entry in the *eft* array is assigned a value exactly once. The value satisfies the definition of *eft*.
- Complexity
  - Simply same as DFS, that is $\mathbf{\Theta(n+m)}$.

SCC: Strongly Connected Component

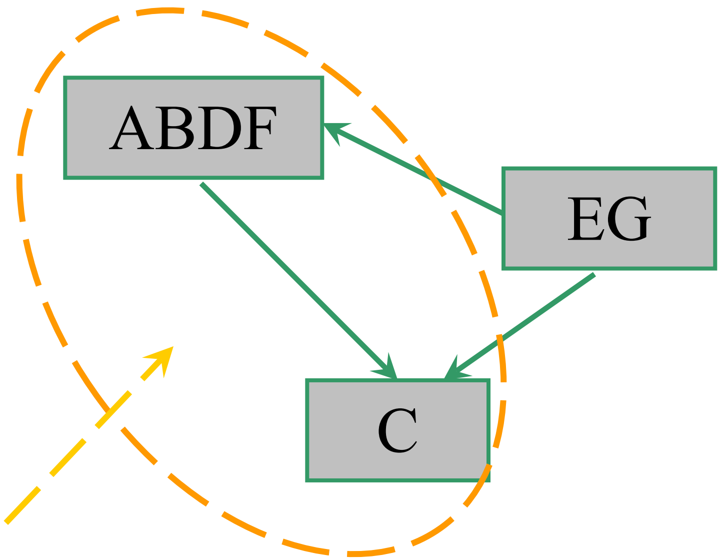
有向图的一个SCC中的两个顶点v和w，都存在从v到w的路径。



Graph G

3 Strongly Connected Components

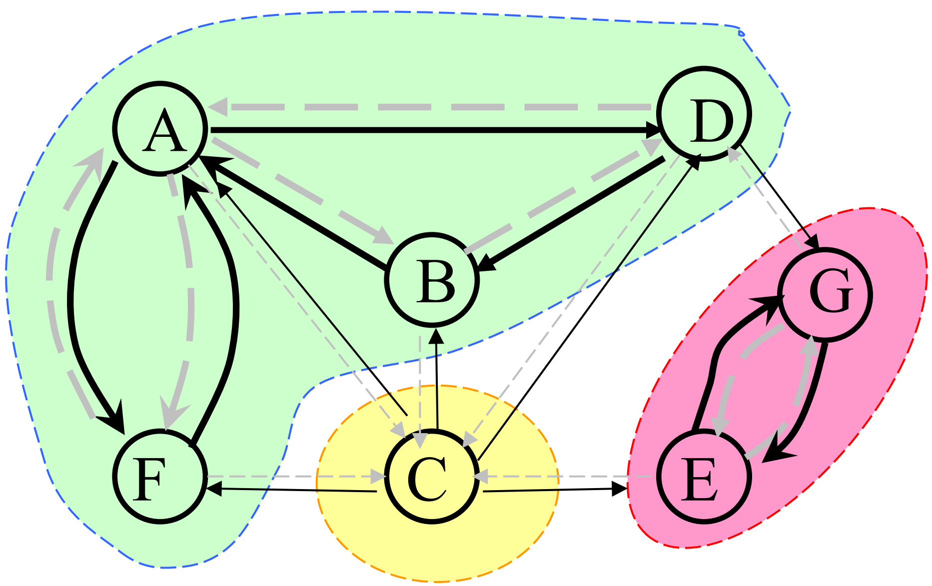
Condensation Graph G↓



It's acyclic, ***Why?***

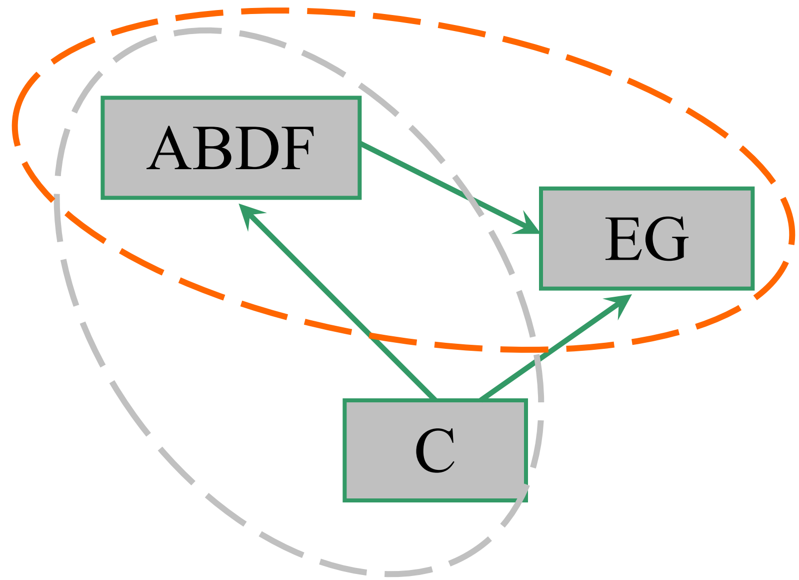
Note: two SCC in one DFS tree

Transpose Graph



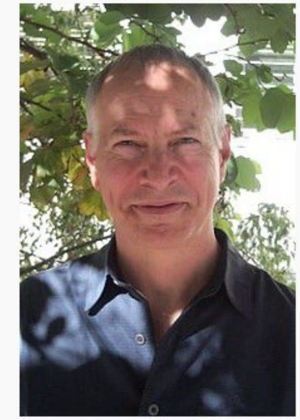
Tranpose Graph G^T
Connected Components unchanged according to vertices

Condensation Graph G↓

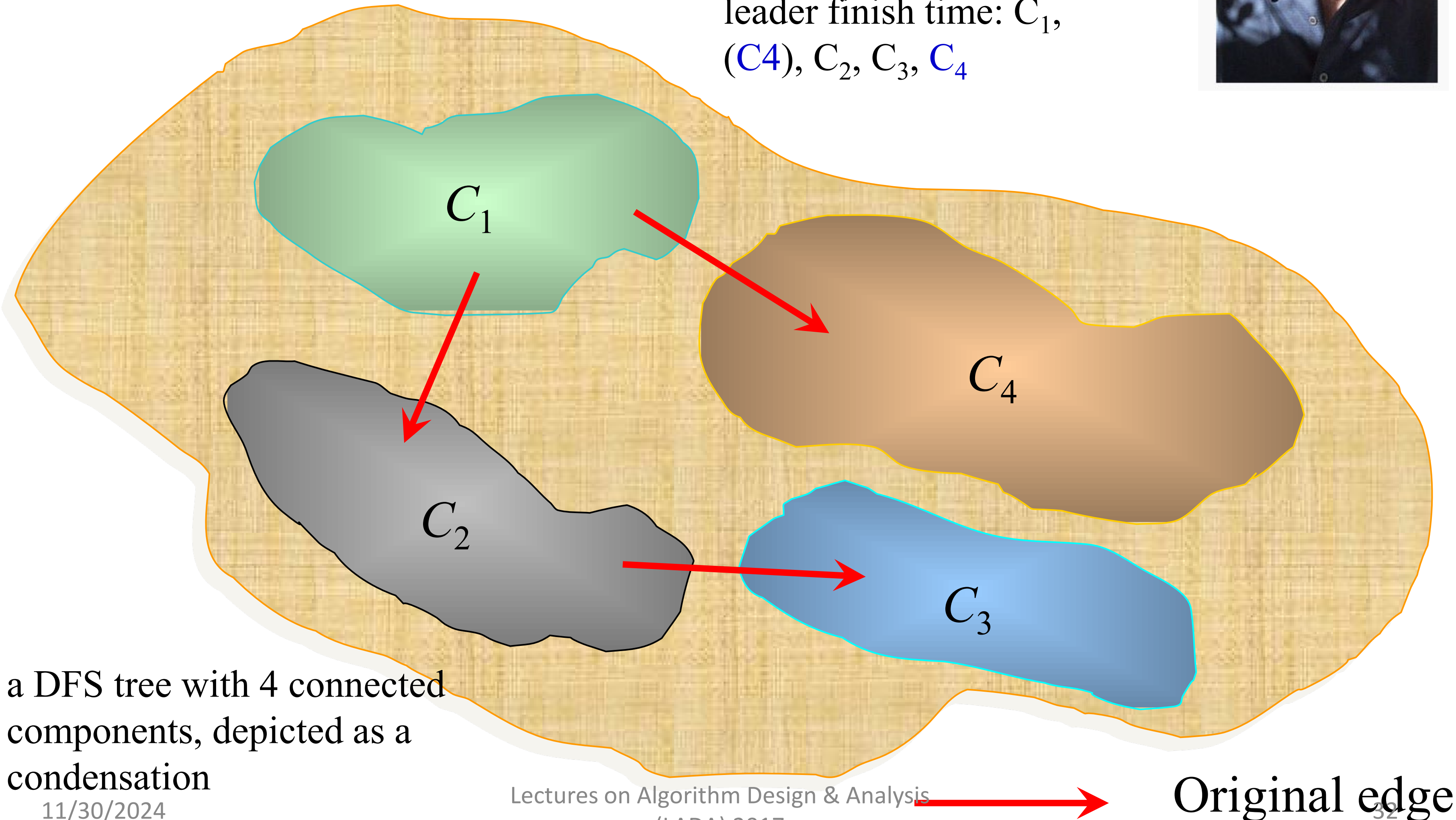


But, DFS tree changed

Basic Idea - G


Robert Endre Tarjan

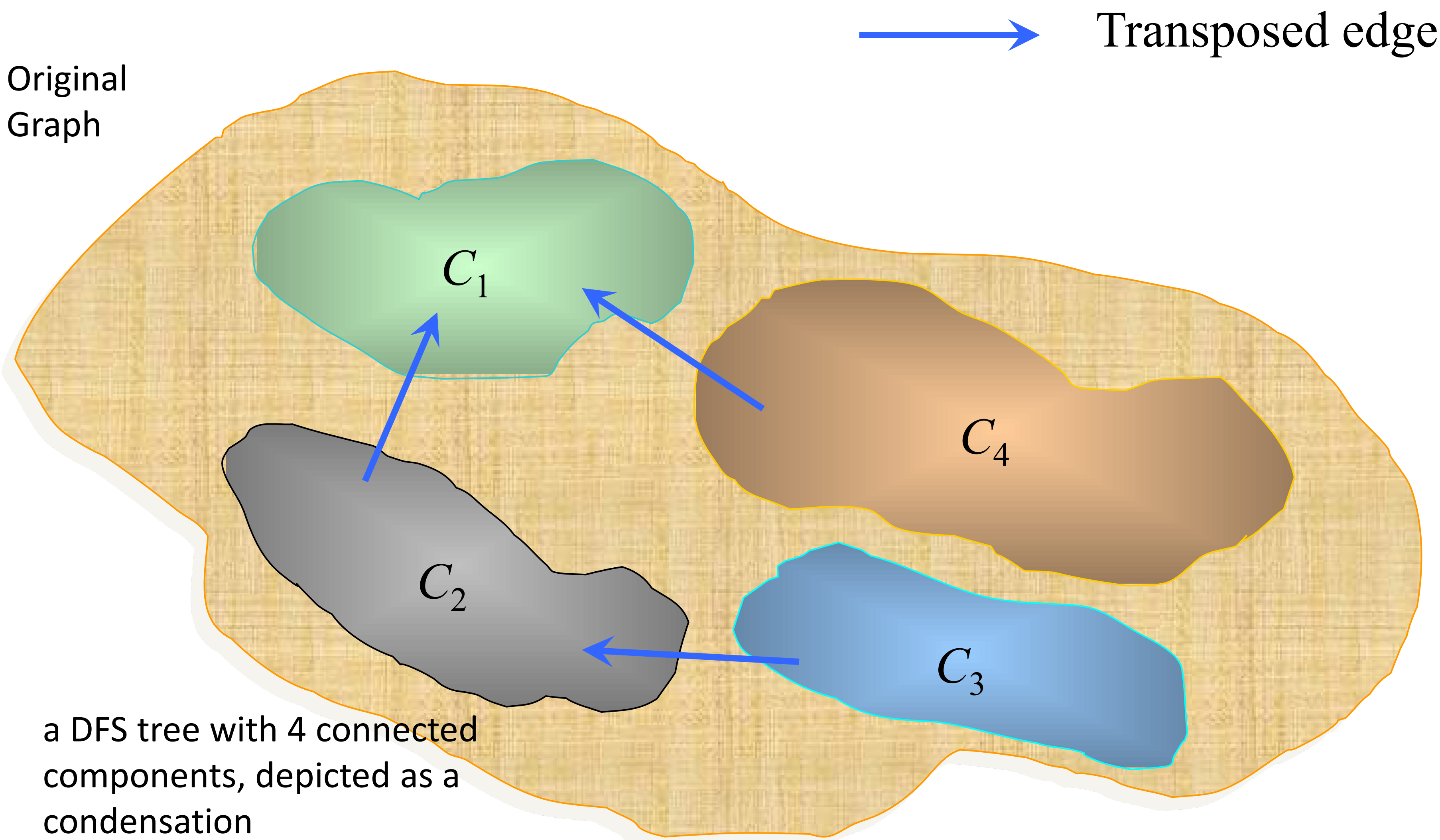
Reverse topo order for leader finish time: C_1, ($C4$), C_2, C_3, C_4



a DFS tree with 4 connected components, depicted as a condensation

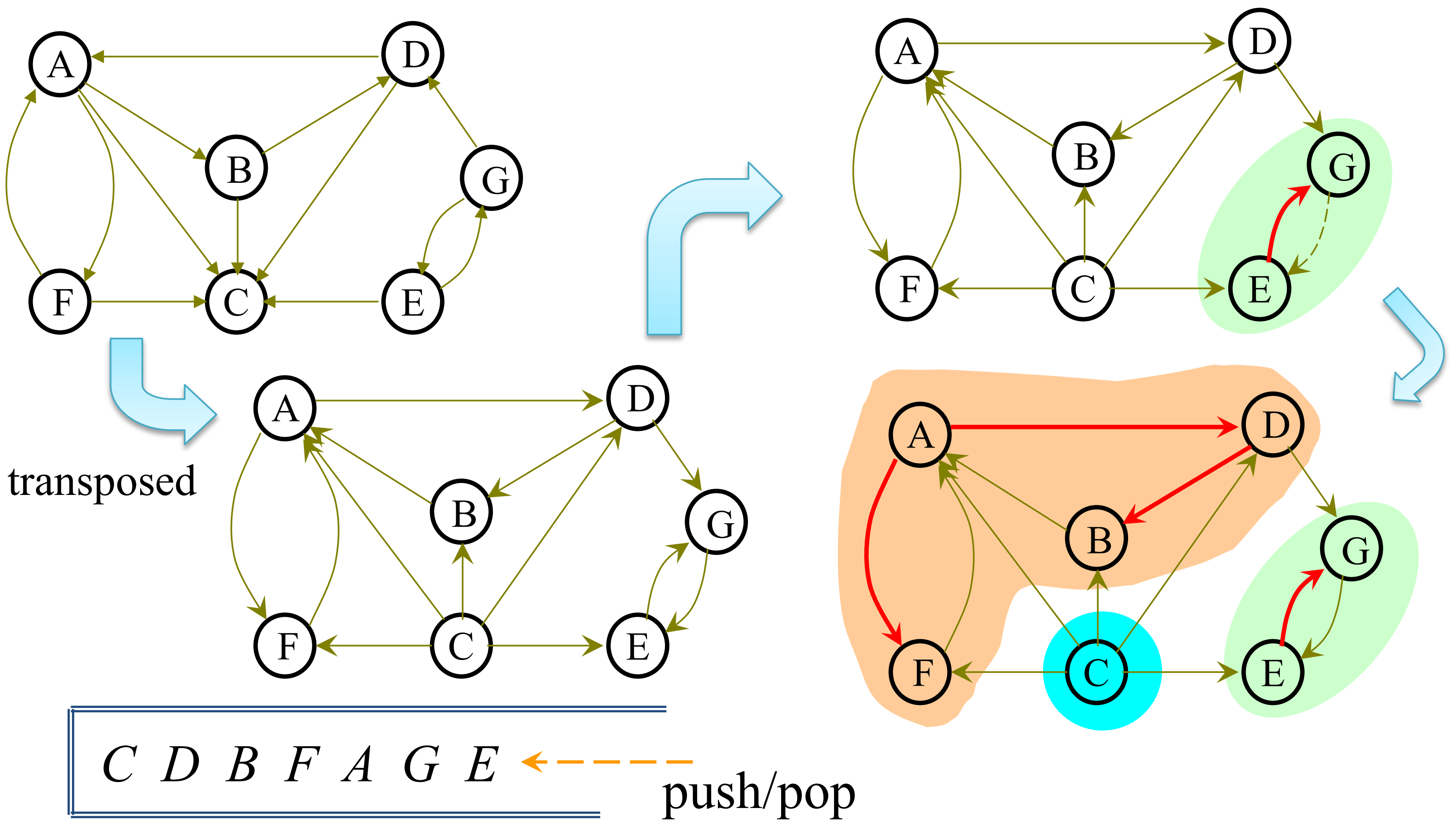
Basic Idea - G^T

Original Graph



a DFS tree with 4 connected components, depicted as a condensation

SCC - An Example



Leader of a Strong Component

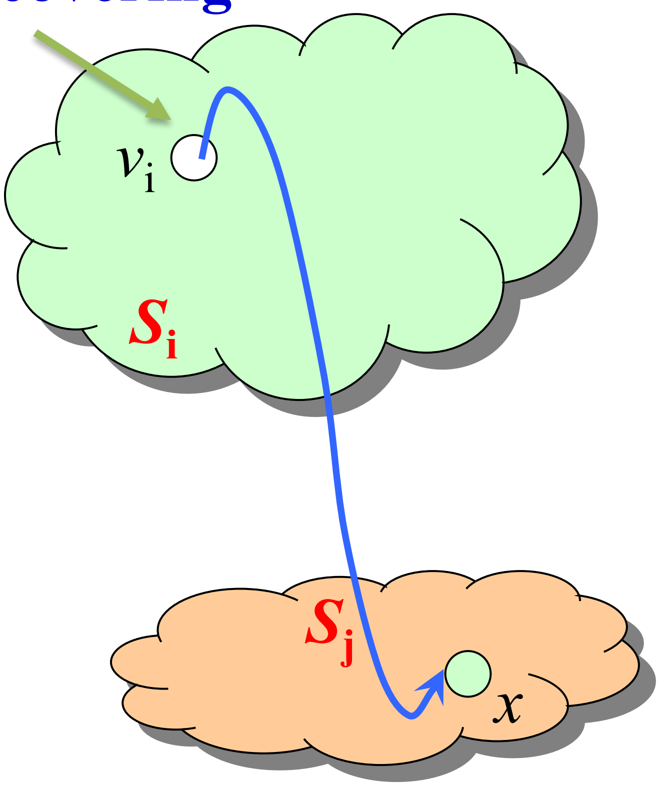
- For a DFS, the first vertex discovered in a strong component S_i is called the **leader** of S_i .
- Each DFS tree of a digraph G contains **only complete** strong components of G, one or more.
  – Proof: Applying White Path Theorem whenever the leader of S_i (i=1,2,…p) is discovered, starting with all vertices being white.
- The leader of S_i is the last vertex to finish among all vertices of S_i. (since all of them in the same DFS tree)

Path between SCCs

At the time a leader v_i is discovered in a DFS search, there is no path from v_i to any gray vertex, say x.

The leader of S_i

At discovering



For any vertex x in other SCC that there is a path from vi to x, *x* can't be gray:

Because there is a path from each gray node to v_i.
If x is gray, x should in same SCC with v_i
So,

1、either, *x* is black (v_i finishes later than x)

2、or $v_i x$-path is a White Path (v_i finishes later than x)

(For 2, consider the [possible] last non-white vertex z on the $v_i x$-path)

一个SCC中的顶点的出边只能到达先被finish的SCC中的顶点！

Active Intervals

- If there is an edge from S_i to S_j, then it is **impossible** that the active interval of v_j is **entirely after** that of v_i. (Note: for leader v_i only)
  - There is no path from a leader of a strong component to any gray vertex.
  - If there is a path from the leader v of a strong component to any x in a different strong component, v finishes later than x.

Strong Component Algorithm: Outline

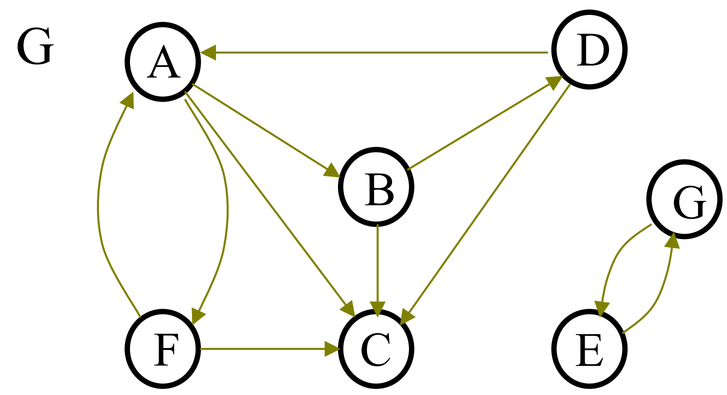
void strongComponents(IntList[] *adjVertices*, **int** *n*, **int**[] *scc*)
//*Phase* 1
1. IntStack *finishStack*=create(*n*);
2. Perform a depth-first search on *G*, using the DFS skeleton. At postorder processing for vertex *v*, insert the statement:
   push(*finishStack*, *v*)

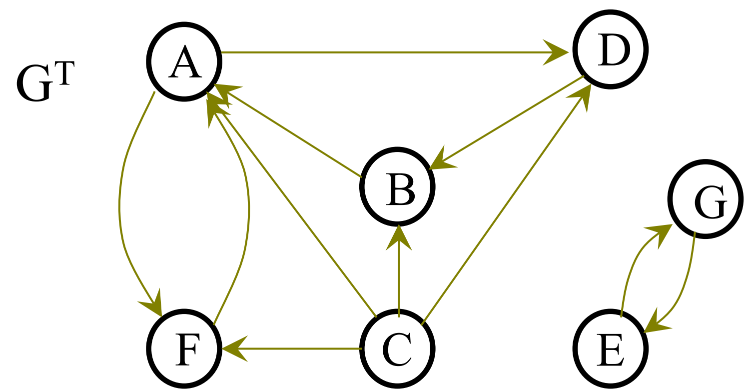
//Phase结束后：
// 一个SCC的顶点中，leader最靠近栈顶
// 后finish的SCC的leader更靠近栈顶
// 不存在从先finish的SCC到达后finish的SCC的边
// 即：在G^T中没有从后finish的SCC到先finish的
// SCC的边

//*Phase* 2
3. Compute G^T, the transpose graph, represented as array *adjTrans* of adjacency list. （G^T如何计算？）
4. **dfsTsweep(*adjTrans*, *n*, *finishStack*, *scc*)**;

return

Note: *G* and G^T have the same SCC sets





- 从A开始遍历，各SCC完成的顺序是：{C} {BDFA} {GE}
- 从E开始遍历，各SCC完成的顺序是：{GE} {C} {BDFA}

Strong Component Algorithm: Core

void dfsTsweep(IntList[] *adjTrans*, **int** *n*, IntStack *finishStack*, **int**[] *scc*)
<Allocate *color* array and initialize to white>
while (finishStack is not empty)
int v=top(*finishStack*);
pop(*finishStack*);
if (color[*v*]==white)
dfsT(*adjTrans*, *color*, *v*, *v*, *scc*);
return;

在本次针对转置图的dfs中：

1. 针对G的DFS中后finish的SCC Leader在本次DFS中先被Discover；
2. 同一个SCC中，Leader先被访问；
3. 在G^T中不存在后Finish的SCC到达先finish的边，因此dfsT不会跨越不同的SCC！

void dfsT(IntList[] *adjTrans*, **int**[] *color*, **int** *v*, **int** *leader*, **int**[] *scc*)
Use the standard depth-first search skeleton.
At preorder processing for vertex v insert the statement:
scc[v]=leader;
Pass leader and scc into recursive calls.

Correctness of Strong Component Algorithm (1)

- In phase 2, each time a white vertex is popped from *finishStack*, that vertex is the Phase 1 leader of a strong component.
  - The later finished, the earlier popped
  - The leader is the first to get popped in the strong component it belongs to
  - If x popped is not a leader, then some other vertex in **the** strong component has been visited previously. But not a partial strong component can be in a DFS tree, so, x must be in a completed DFS tree, and is not white.

Correctness of Strong Component Algorithm (2)

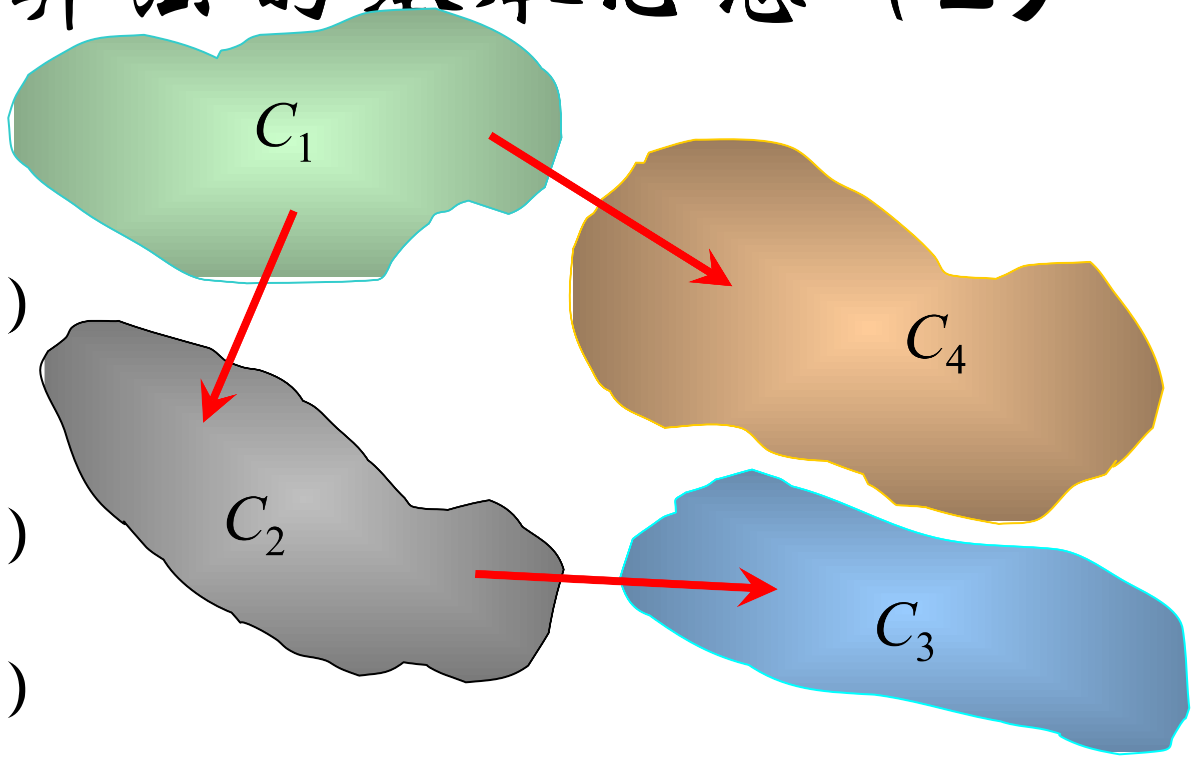
- In phase 2, each depth-first search tree contains exactly one strong component of vertices
  - Only “exactly one” need to be proved
  - Assume that v_i, a phase 1 leader is popped. If another component S_j is reachable from v_i in G^T, there is a path in G from v_j to v_i. So, in phase 1, v_j finished later than v_i, and popped earlier than v_i in phase 2. So, when v_i popped, all vertices in S_j are black. So, S_j are not contained in DFS tree containing $v_i(S_i)$.

Tarjan的SCC算法的基本思想（1）

- 进行DFS遍历时，对结点的访问顺序是：
  - 访问某个SCC S_i的Leader,
  - 访问这个SCC中的某些结点
  - 访问另一个SCC的Leader，递归
    - 遍历该SCC的结点，以及这个SCC相连的其它SCC
    - 返回（回到S_i的结点)
  - 访问另外的SCC的Leader
    - 遍历该SCC的结点，以及…
    - 返回（回到S_i的结点)
  - … …
  - 完成对S_i的遍历，从Leader回溯

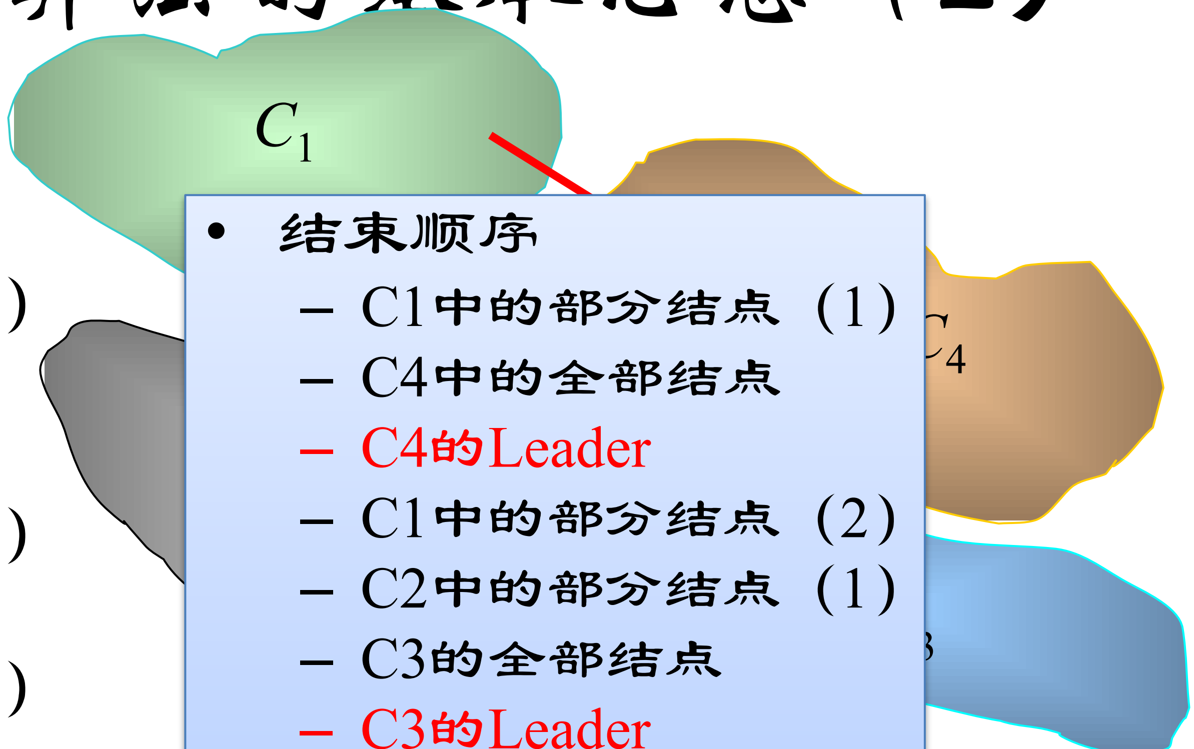
Tarjan的SCC算法的基本思想（2）

- 访问顺序
  - C1的Leader
  - C1的部分结点（1）
  - C4的Leader
  - C4的全部结点
  - C1的部分结点（2）
  - C2的Leader
  - C2的部分结点（1）
  - C3的Leader
  - C3的全部结点
  - C2的部分结点（2）
  - C1的部分结点（3）



Tarjan的SCC算法的基本思想（2）

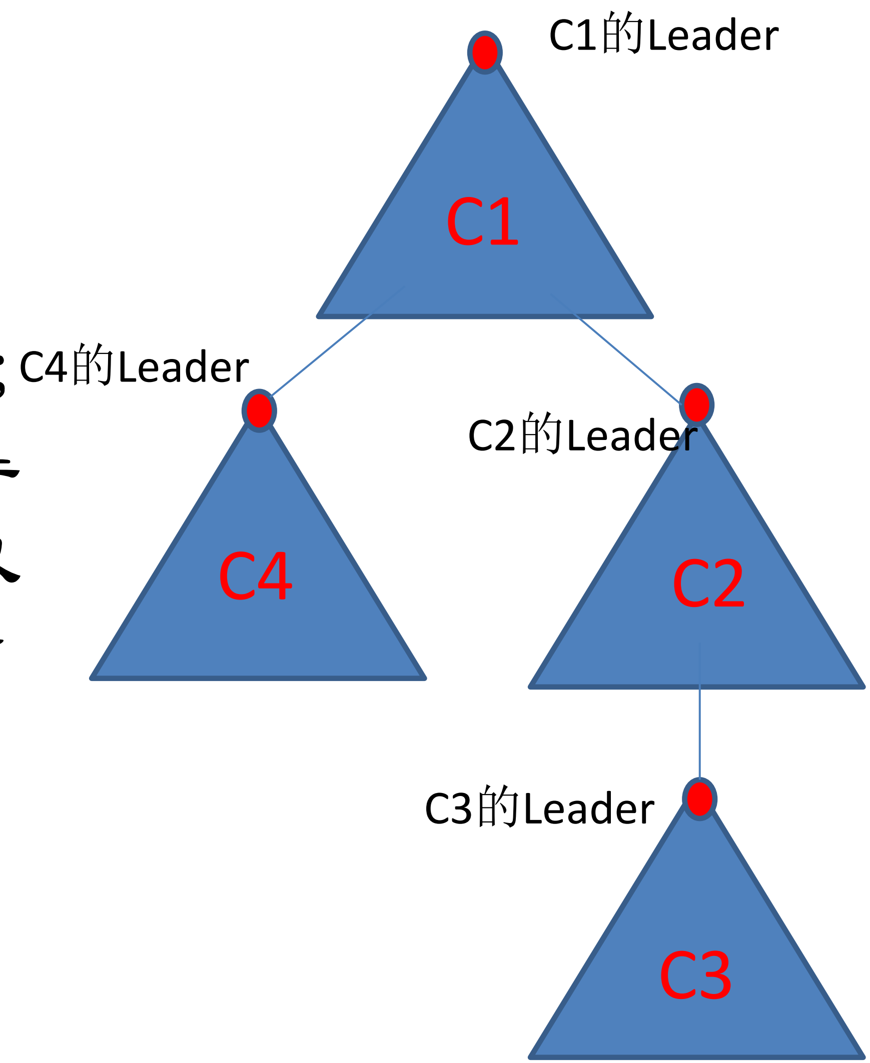
- 访问顺序
  - C1的Leader
  - C1的部分结点（1）
  - C4的Leader
  - C4的全部结点
  - C1的部分结点（2）
  - C2的Leader
  - C2的部分结点（1）
  - C3的Leader
  - C3的全部结点
  - C2的部分结点（2）
  - C1的部分结点（3）



- 结束顺序
  - C1中的部分结点（1）
  - C4中的全部结点
  - C4的Leader
  - C1中的部分结点（2）
  - C2中的部分结点（1）
  - C3的全部结点
  - C3的Leader
  - C2中的部分结点（2）
  - C2的Leader
  - C1的部分结点（3）
  - C1的Leader

Tarjan的SCC算法的基本思想（3）

- 在DFS树中，每个SCC的Leader的后代包括
  - 它所在SCC的全部结点（White Path Theorem）；
  - 其它SCC的结点集合，并且每个SCC的结点集合以该SCC的Leader作为子树的根。



Tarjan的SCC算法的基本思想（4）

- 按照访问顺序（先根序）将结点排列，Leader之后的结点包含了
  - 它所在SCC的全部结点
  - 以及其它的SCC的结点，这些SCC的结点是以其Leader为第一个结点，嵌套在其中的。
- Tarjan算法：按照后根序，当完成某个Leader的访问时，从先根序列的尾部Pop出Leader所在的SCC结点，就得到各个SCC的结点。
  - 前提：我们能够判定某个顶点是否leader!

后根序	先根序
C1中的部分结点（1）	C1的Leader
C4中的全部结点	C1的部分结点（1）
C4的Leader	C4的Leader
C1中的部分结点（2）	C4的全部结点
C2中的部分结点（1）	C1的部分结点（2）
C3的全部结点	C2的Leader
C3的Leader	C2的部分结点（1）
C2中的部分结点（2）	C3的Leader
C2的Leader	C3的全部结点
C1的部分结点（3）	C2的部分结点（2）
C1的Leader	C1的部分结点（3）

Tarjan算法基本框架（1）

input: graph $G = (V, E)$
output: set of strongly connected components (sets of vertices)

index := 0
S := empty array
for each *v* **in** *V* **do if** (*v*.index is undefined)
        then strongconnect(*v*)
        end if
end for

Tarjan算法基本框架（2）

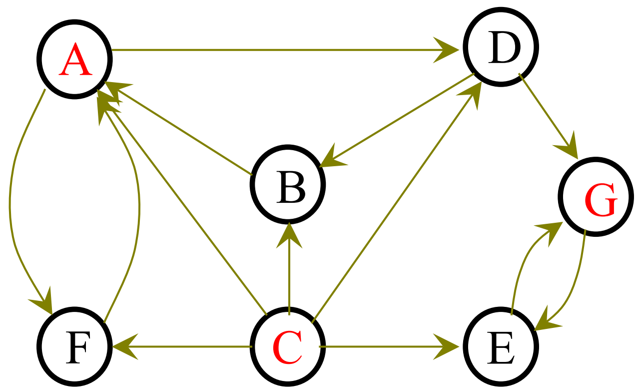
```
function strongconnect(v)
    //设置先根序，lowLink,
    v.index := index        v.lowlink := index        index := index + 1
    //按照先根序压栈；
    S.push(v)               v.onStack := true

    //递归，还需要添加一些代码，为判断v是否Leader结点设置信息
    for each (v, w) in E do
        if (w.index is undefined) then        //white结点
            strongconnect(w)
        … …

    // 后根处理，如果v是leader node, 从先根处理的栈中获得一个SCC（弹出到v位置）
    if (v is a leader node) then //此时从v出发，能够到达的其它SCC的顶点均已经出栈
        repeat
            w := S.pop()
            w.onStack := false
            add w to current strongly connected component
        while (w != v)
        output the current strongly connected component
    end if
end function
```

Tarjan算法的运行

SCCs：
1、ABDF
2、C
3、EG

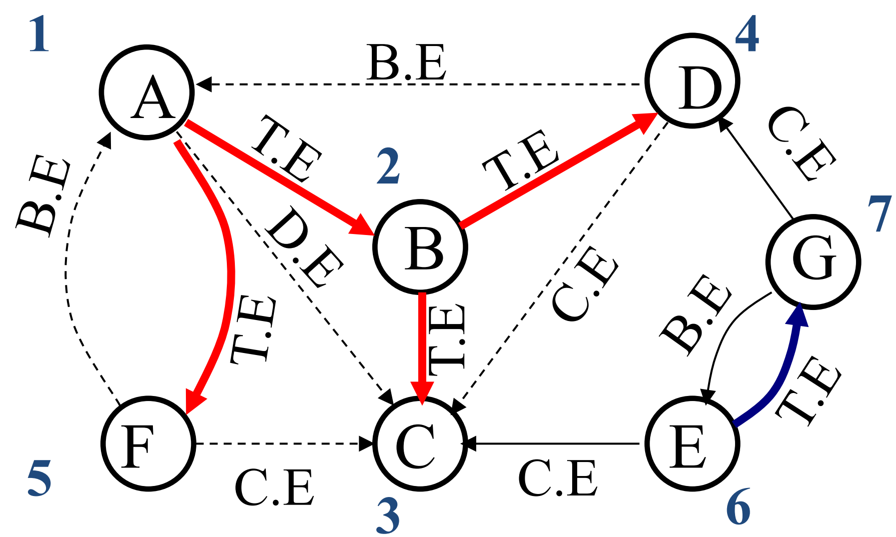


性质：
处理某个SCC中的结点v的时候，栈中的顶部只包含v所在SCC的结点。

- DFS先根序列：ADBGEF，C
- DFS后根序列：BEGDFA，C
  - 当后根访问到G时，G是Leader，栈中是ADBGE，GE是一个SCC，出栈。栈中变成ADB
  - 当后根访问到A时，A是Leader，栈中为ADBF，ADBF是一个SCC，出栈。栈中为空。
  - 继续从C开始进行DFS，得到新的SCC C

Tarjan算法中判断Leader的原理

- 假设从一个SCC的Leader出发，以DFS方式进行遍历该SCC后得到DFS树T。那么T中每个不同于Leader的结点u都必然存在一个后代结点v，使得v的某个相邻结点w满足w.index < u.index，且w在该SCC中。
- 证明
  - 设u所在子树的结点集合是S。考察从S中某个结点出发，到达SCC中不在S中的结点的路径。
  - 这样的路径必然存在，因为u在SCC中，必然有从S中某个结点到达SCC的Leader的路径，而这路径中的某条边会离开S。
  - 这种边可以分成两类：Back Edge和Cross Edge。因为边的目标结点不在S中，这个结点的index必然小于u的index。



- 可以使用后根序遍历SCC并计算SCC中各个结点u的*所有后代结点的相邻结点的index*中的最小值，记录为u.lowIndex。
- 因为在对SCC进行遍历时，Leader的index在SCC的所有结点中最低，因此在SCC中，只有Leader的lowIndex会等于它自己的index

Tarjan算法中判断Leader的方法

```
… … … …
S.push(v)        v.onStack := true    //v按先根序压入栈中，并标记v在栈中

//用于设置和判断v是否Leader信息的代码，实际上是对v所在的SCC的后根序处理，
//记录每个结点的在本SCC中所有后代结点的相邻结点的最低先根序。
for each (v, w) in E do
    if (w.index is undefined) then //white结点
        strongconnect(w)
        v.lowlink := min(v.lowlink, w.lowlink)
    else if (w.onStack) then
        //注意，当w在栈中时，要么w在当前SCC中时，要么w.index >遍历时才计算lowLink
        v.lowlink := min(v.lowlink, w.index)
    end if
end for

// If v is a leader node, pop the stack and generate an SCC
if (v.lowLink == v.index) then
                …
end if
```