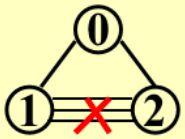
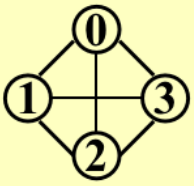
1 Definitions

- **G(V, E)** where G ::= graph, V = V(G) ::= finite nonempty set of vertices, and E = E(G) ::= finite set of edges.
- **Undirected graph:** $(v_i, v_j) = (v_j, v_i)$ ::= the same edge.
- **Directed graph (digraph):** $\langle v_i, v_j \rangle ::= v_i \rightarrow v_j \neq \langle v_j, v_i \rangle$ (tail: v_i, head: v_j)
- **Restrictions :**
  (1) **Self loop** is illegal.
  (2) **Multigraph** is not considered

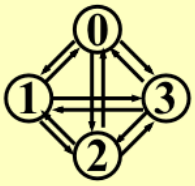


- **Complete graph:** a graph that has the maximum number of edges



$\#\text{ of V} = n \Rightarrow$

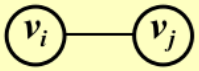
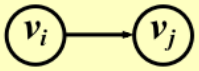
$\#\text{ of E} = C_n^2 = \frac{n(n-1)}{2}$



$\#\text{ of V} = n \Rightarrow$

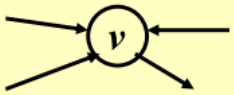
$\#\text{ of E} = P_n^2 = n(n-1)$

- 自己指向自己是不合法的
- 多重指向也是不考虑的
- 无向图 / 有向图

-  v_i and v_j are adjacent ;
  (v_i , v_j) is incident on v_i and v_j
-  v_i is adjacent to v_j ; v_j is adjacent from v_i ;
  < v_i , v_j > is incident on v_i and v_j
- Subgraph G' ⊂ G ::= V(G') ⊆ V(G) && E(G') ⊆ E(G)
- Path (⊂ G) from v_p to v_q ::= { v_p, v_{i1}, v_{i2}, ···, v_{in}, v_q } such that (v_p, v_{i1}), (v_{i1}, v_{i2}), ···, (v_{in}, v_q) or < v_p, v_{i1} >, ···, < v_{in}, v_q > belong to E(G)
- Length of a path ::= number of edges on the path
- Simple path ::= v_{i1}, v_{i2}, ···, v_{in} are distinct
- Cycle ::= simple path with $v_p = v_q$
- v_i and v_j in an undirected G are connected if there is a path from v_i to v_j (and hence there is also a path from v_j to v_i)
- An undirected graph G is connected if every pair of distinct v_i and v_j are connected

- 无向图只考虑相连，而有向图还考虑方向
- 联通图：任意两个节点都能相通（无向图中）

- **(Connected) Component of an undirected G ::= the maximal connected subgraph**
- **A tree ::= a graph that is connected and *acyclic***
- **A DAG ::= a directed acyclic graph**
- **Strongly connected directed graph G ::= for every pair of v_i and v_j in V(G), there exist directed paths from v_i to v_j and from v_j to v_i. If the graph is connected without direction to the edges, then it is said to be weakly connected**
- **Strongly connected component ::= the maximal subgraph that is strongly connected**
- **Degree(v) ::= number of edges incident to v. For a directed G, we have in-degree and out-degree. For example:**



in-degree(v) = 3; out-degree(v) = 1; degree(v) = 4

- **Given G with n vertices and e edges, then**

$$e = \left(\sum_{i=0}^{n-1} d_i \right) \Big/ 2 \quad \text{where } d_i = \text{degree}(v_i)$$

- 最大联通分量：部分联通的最多的子图
- 树其实就是没有cycle的graph
- DAG：有向、非周期
- 强联通图：任意两个点都存在互相的路径（有向图）
- 强联通分量：最大联通的子图
- indegree：需要所有节点；outdegree：当前节点就可
- 边的个数 = （点赞 + 收赞）/2

1.1 Representation of Graphs

1.1.1 Adjacency Matrix

adj_mat [n] [n] is defined for G(V, E) with n vertices, $n \geq 1$:

$$\text{adj\_mat}[i][j] = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ or } < v_i, v_j > \in E(G) \\ 0 & \text{otherwise} \end{cases}$$

Note: If G is undirected, then adj_mat[ ][ ] is symmetric. Thus we can save space by storing only half of the matrix.

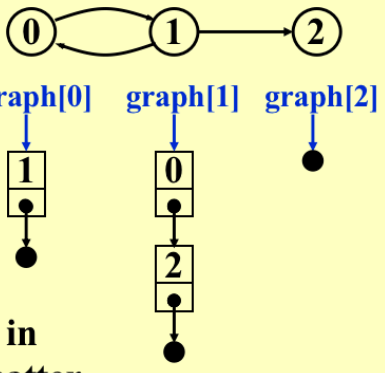
$$\text{degree}(i) = \sum_{j=0}^{n-1} \text{adj\_mat}[i][j] \quad \text{(if G is undirected)}$$

$$+ \sum_{j=0}^{n-1} \text{adj\_mat}[j][i] \quad \text{(if G is directed)}$$

- 如果是无向的，那么只用存一半就行了

1.1.2 Adjacency Lists

〖Example〗



$$\text{adj\_mat}[3][3] = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Note: The order of nodes in each list does not matter.

For undirected G:

$S = n$ heads + $2e$ nodes $= (n+2e)$ ptrs+$2e$ ints

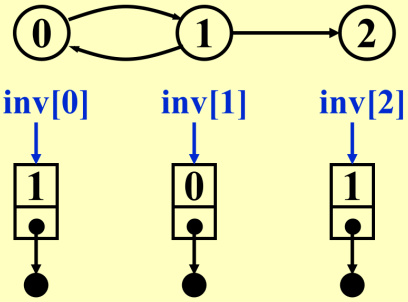
- replace each row by a linked list

Degree(i) = number of nodes in graph[i] (**if G is undirected**).
T of examine E(G) = O($n + e$)

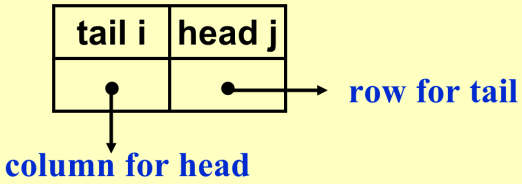
If G is **directed**, we need to find in-degree(v) as well.

Method 1 Add inverse adjacency lists.

〖Example〗

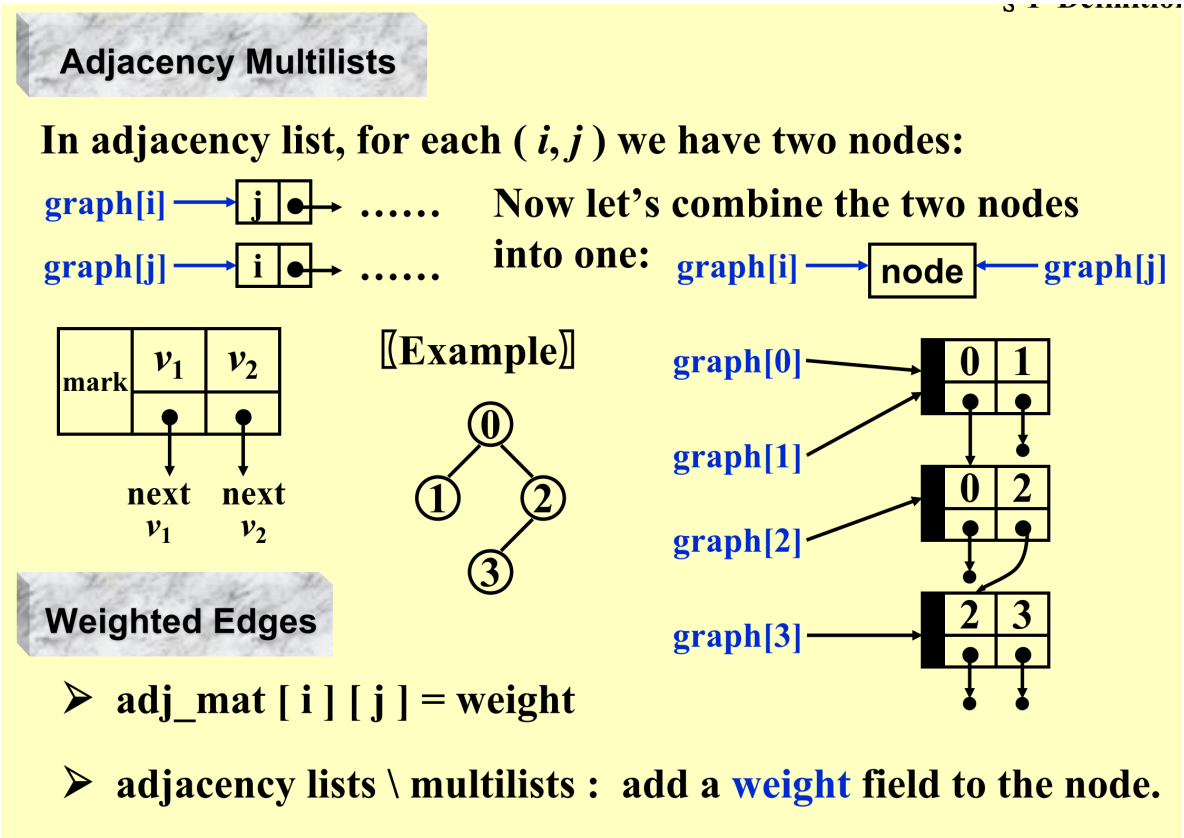


Method 2 Multilist (Ch 3.2) representation for adj_mat[i] [j]



- 其实被指向就是指向矩阵的转置

1.1.3 Adjacency Multilists



- 每个节点实际上是一条边

2 Topological Sort 拓扑排序

- 其实不是一个排序算法，是一个调度算法

AOV Network ::= digraph G in which V(G) represents activities (e.g. the courses) and E(G) represents precedence relations (e.g. C1 → C3 means that C1 is a prerequisite course of C3).

i is a **predecessor** of *j* ::= there is a path from *i* to *j*
i is an **immediate predecessor** of *j* ::= $< i, j > \in E(G)$
Then *j* is called a **successor** (**immediate successor**) of *i*

Partial order ::= a precedence relation which is both **transitive** ($i \to k, k \to j \Rightarrow i \to j$) and **irreflexive** ($i \to i$ is impossible).

Note: If the precedence relation is reflexive, then there must be an *i* such that *i* is a predecessor of *i*. That is, *i* must be done before *i* is started. Therefore if a project is **feasible**, it must be **irreflexive**.

Feasible AOV network must be a dag (directed acyclic graph).

A topological order is a linear ordering of the vertices of a graph such that, for any two vertices, i, j, if i is a predecessor of j in the network then i precedes j in the linear ordering.

Note: The topological orders may not be unique for a network. For example, there are several ways (topological orders) to meet the degree requirements in computer science.

```
1  void Topsort( Graph G )
2  {   int  Counter;
3      Vertex  V, W;
4      for ( Counter = 0; Counter < NumVertex; Counter ++ ) {
5        V = FindNewVertexOfDegreeZero( ); // 每次要扫描中间优先级空的
6        if ( V == NotAVertex ) {
7          Error ( “Graph has a cycle” );
8          break;
9        }
10       TopNum[ V ] = Counter; /* or output V */
11       for ( each W adjacent to V )
12         // 和V这个节点有关系的其要求都会-1，一直减到0，那么下次循环就会被塞到
13         Indegree[ W ] – – ;
14     }
15 }
```

$$T = O(|V|^2) \tag{6}$$

Improvement: Keep all the unassigned vertices of degree 0 in a special box (queue or stack).

```
void Topsort( Graph G )
{   Queue  Q;
    int  Counter = 0;
    Vertex  V, W;
    Q = CreateQueue( NumVertex );  MakeEmpty( Q );
    // 入度为0的，说明不需要准备的，直接纳入queue
    for ( each vertex V )
      if ( Indegree[ V ] == 0 )   Enqueue( V, Q );

    while ( !IsEmpty( Q ) ) {
      V = Dequeue( Q );
      TopNum[ V ] = ++ Counter; /* assign next */
      for ( each W adjacent to V )
        if ( – – Indegree[ W ] == 0 )  Enqueue( W, Q );
    }  /* end-while */

    if ( Counter != NumVertex )
      Error( “Graph has a cycle” );
    DisposeQueue( Q ); /* free memory */
}
```

$$T = O(|V|) \tag{7}$$

- 这个算法会要求，上面那个较差的算法是不要求的
- 就是每次不去找已经为0的了，就是队列当前节点对其所有关系点做工作，做完工作顺便判断

3 Shortest Path Algorithm

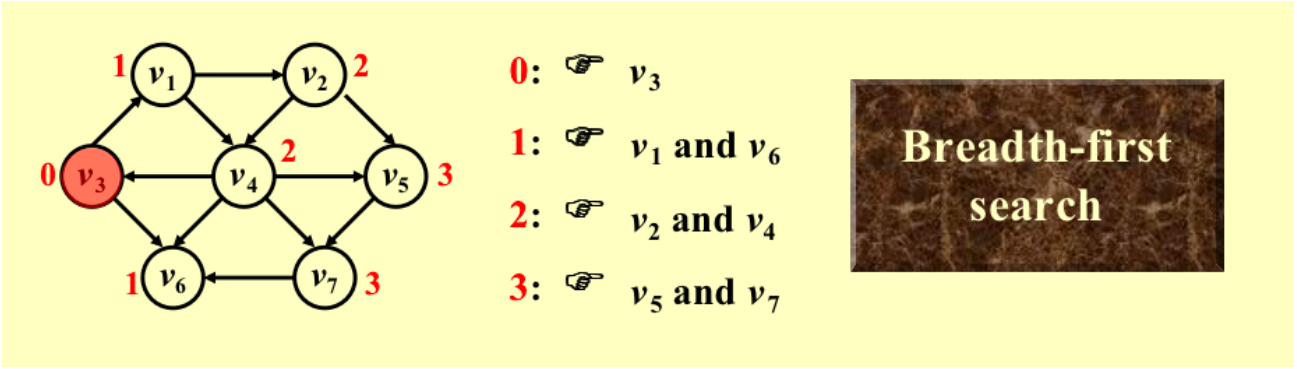
- Given a digraph G = (V, E), and a cost function c(e) for$e \in E(G)$. The length of a path P from source to destination is $\sum_{e_i \subset P} c\left(e_i\right)$ (also called weighted path length).
- 不带权的最短路就是权都等于1

3.1 Single-Source Shortest-Path Problem

> Given as input a weighted graph, G = (V, E), and a distinguished vertex, s, find the shortest weighted path from s to every other vertex in G.

3.1.1 Unweighted Shortest Paths BFS

3.1.1.1 Sketch of the idea



3.1.1.2 Implementation

```
Table[ i ].Dist ::= distance from s to vi  /* initialized to be infinite except for s */
Table[ i ].Known ::= 1 if vi is checked; or 0 if not
Table[ i ].Path ::= for tracking the path   /* initialized to be 0 */
```

3.1.1.2.1 basis

```
void Unweighted( Table T )
{   int  CurrDist;
    Vertex  V, W;
    for ( CurrDist = 0; CurrDist < NumVertex; CurrDist ++ ) {
        for ( each vertex V )
          // 这里很冗余，我做了一次后，全部再遍历一遍很花时间
          // 可以直接用queue来处理
          if ( !T[ V ].Known && T[ V ].Dist == CurrDist ) {
            T[ V ].Known = true;
            for ( each W adjacent to V )
              if ( T[ W ].Dist == Infinity ) {
                    T[ W ].Dist = CurrDist + 1;
                    T[ W ].Path = V;
              } /* end-if Dist == Infinity */
          } /* end-if !Known && Dist == CurrDist */
```

```
    }  /* end-for CurrDist */
}
```

- Worst case: 一条线的时候，时间复杂度到$T(N) = O(N^2)$

3.1.1.2.2 Improvement （要求）

```
void Unweighted( Table T )
{   /* T is initialized with the source vertex S given */
    Queue  Q;
    Vertex  V, W;
    Q = CreateQueue ( NumVertex );  MakeEmpty( Q );
    Enqueue( S, Q ); /* Enqueue the source vertex */
    while ( !IsEmpty( Q ) ) {
        V = Dequeue( Q );
        T[ V ].Known = true; /* not really necessary */
        for ( each W adjacent to V )
          if ( T[ W ].Dist == Infinity ) {
            T[ W ].Dist = T[ V ].Dist + 1;
            T[ W ].Path = V;
            Enqueue( W, Q );
          } /* end-if Dist == Infinity */
    } /* end-while */
    DisposeQueue( Q ); /* free memory */
}
```

时间复杂度：本质上就是每个点每条边都会算一次，那么复杂度就是根据节点个数和边数来定

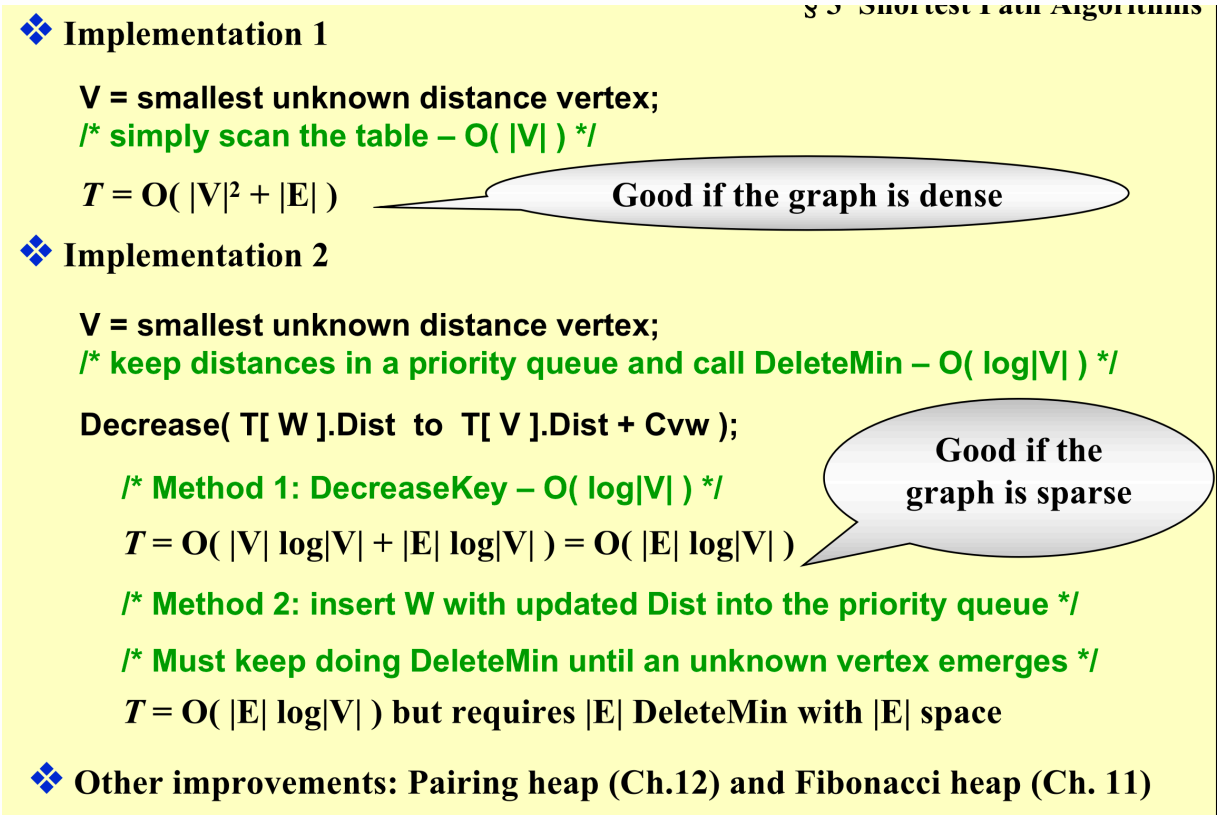
$$T(N) = O(|V| + |E|) \tag{8}$$

3.1.2 Dijkstra's Algorithm (for weighted shortest paths)

- Let S = { s and vi's whose shortest paths have been found } 定义一个known set，确定已知的最短
- For any $u \notin S$, define distance [u] = minimal length of path $s \to (vi \in S) \to u$. If the paths are generated in non-decreasing order, then
  - the shortest path must go through ONLY $v_i \in S$;
  - u is chosen so that $distance[u] = minw \notin S|distance[w]$ (If u is not unique, then we may select any of them) ; /* Greedy贪婪 Method */
  - if distance [u1] < distance [u2] and we add u1 into S, then distance [u2] may change. If so,

a shorter path from s to u2 must go through u1 and distance' [u2] = distance [u1] + length(< u1, u2>).

```
void Dijkstra( Table T )
{   /* T is initialized by Figure 9.30 on p.303 */
    Vertex  V, W;
    for ( ; ; ) {
        V = smallest unknown distance vertex;
      // current smallest is known as the global smallest
      // 可以用 Heap 实现
        if ( V == NotAVertex )
          break;
        T[ V ].Known = true;
        for ( each W adjacent to V )
          if ( !T[ W ].Known )
            if ( T[ V ].Dist + Cvw < T[ W ].Dist ) {
                Decrease( T[ W ].Dist  to   T[ V ].Dist + Cvw );
                T[ W ].Path = V;
            } /* end-if update W */
    } /* end-for( ; ; ) */
/* not work for edge with negative cost!!! */
```



3.1.3 Graphs with Negative Edge Costs

```
void  WeightedNegative( Table T )
{   /* T is initialized by Figure 9.30 on p.303 */
    Queue  Q;
    Vertex  V, W;
    Q = CreateQueue (NumVertex );  MakeEmpty( Q );
    Enqueue( S, Q ); /* Enqueue the source vertex */
    while ( !IsEmpty( Q ) ) {
        V = Dequeue( Q );
        for ( each W adjacent to V )
          if ( T[ V ].Dist + Cvw < T[ W ].Dist ) {
            T[ W ].Dist = T[ V ].Dist + Cvw;
            T[ W ].Path = V;
            if ( W is not already in Q )
              Enqueue( W, Q ); // 可以在程序内设一个Inqueue[],enqueue的时候设为1，dequeue的时候设为0
          } /* end-if update */ // 进入if中的都是需要update的，把其放入queue中
    } /* end-while */
    DisposeQueue( Q ); /* free memory */
}
/* negative-cost cycle will cause indefinite loop */
// 为解决这种问题可以限制dequeue的次数
```

3.1.4 Acyclic Graphs

- If the graph is acyclic, vertices may be selected in topological order since when a vertex is selected, its distance can no longer be lowered without any incoming edges from unknown nodes.

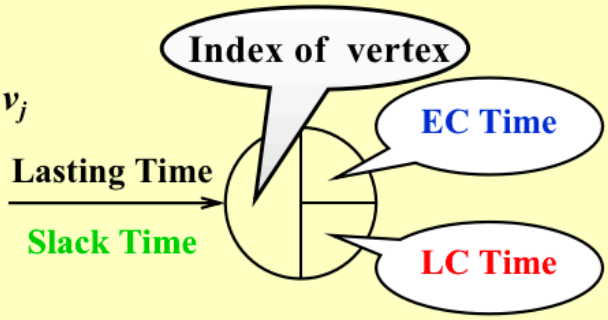
T = O(|E| + |V|) and no priority queue is needed.

❖ Application: AOE (Activity On Edge) Networks

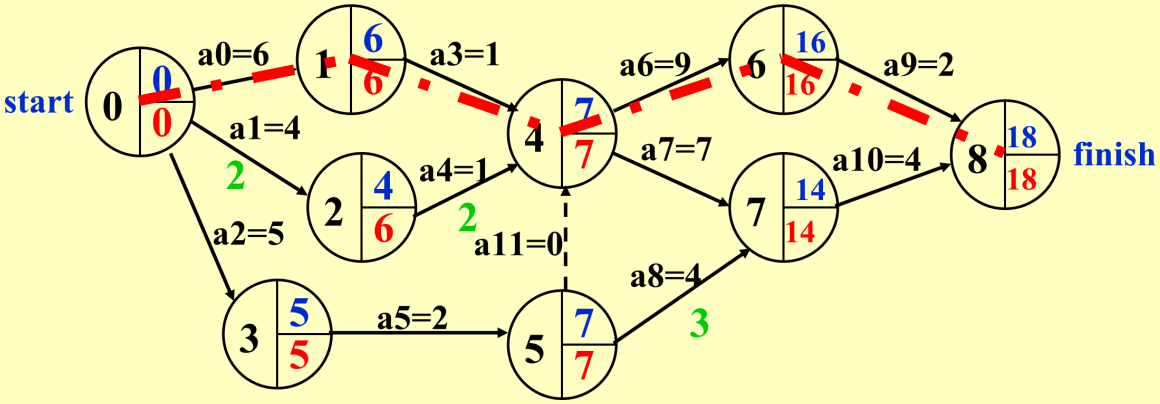
—— scheduling a project

EC[j] \ LC[j] ::= the earliest \ latest completion time for node v_j

CPM (Critical Path Method)



〖Example〗 AOE network of a hypothetical project



➢ Calculation of EC: Start from v0, for any a_i = <v, w>, we have

$$EC[w] = \max_{(v,w)\in E}\{EC[v] + C_{v,w}\}$$

➢ Calculation of LC: Start from the last vertex v8, for any a_i = <v, w>, we have $LC[v] = \min_{(v,w)\in E}\{LC[w] - C_{v,w}\}$

➢ Slack Time of <v,w> = $LC[w] - EC[v] - C_{v,w}$

➢ Critical Path ::= path consisting entirely of zero-slack edges.

- EC：最迟完工时间
- LC：最早完工时间
- Slack Time：可摸鱼时间
- A、从开始顶点 v1 出发，令 ve(1)=0，按拓扑有序序列求其余各顶点的可能最早发生时间。

- Ve(k)=max{ve(j)+dut(<j,k>)} , j∈T 。其中T是以顶点vk为尾的所有弧的头顶点的集合（2 ≤ k ≤ n）。如果得到的拓朴有序序列中顶点的个数小于网中顶点个数n，则说明网中有环，不能求出关键路径，算法结束。
- B、从完成顶点vn出发，令vl(n) = ve(n)，按逆拓扑有序求其余各顶点的允许的最晚发生时间：vl(j)=min{vl(k)-dut(<j,k>)} ,k∈S 。其中 S 是以顶点vj是头的所有弧的尾顶点集合（1 ≤ j ≤ n-1）。
- C、==求每一项活动ai（1 ≤ i ≤ m）的最早开始时间e(i)=ve(j)，最晚开始时间l(i)=vl(k)-dut(<j,k>)==。

  若某条弧满足 e(i)=l(i) ，则它是关键活动。

3.1.4.1 All-Pairs Shortest Path Problem

For all pairs of v_i and v_j ($i \neq j$), find the shortest path between.

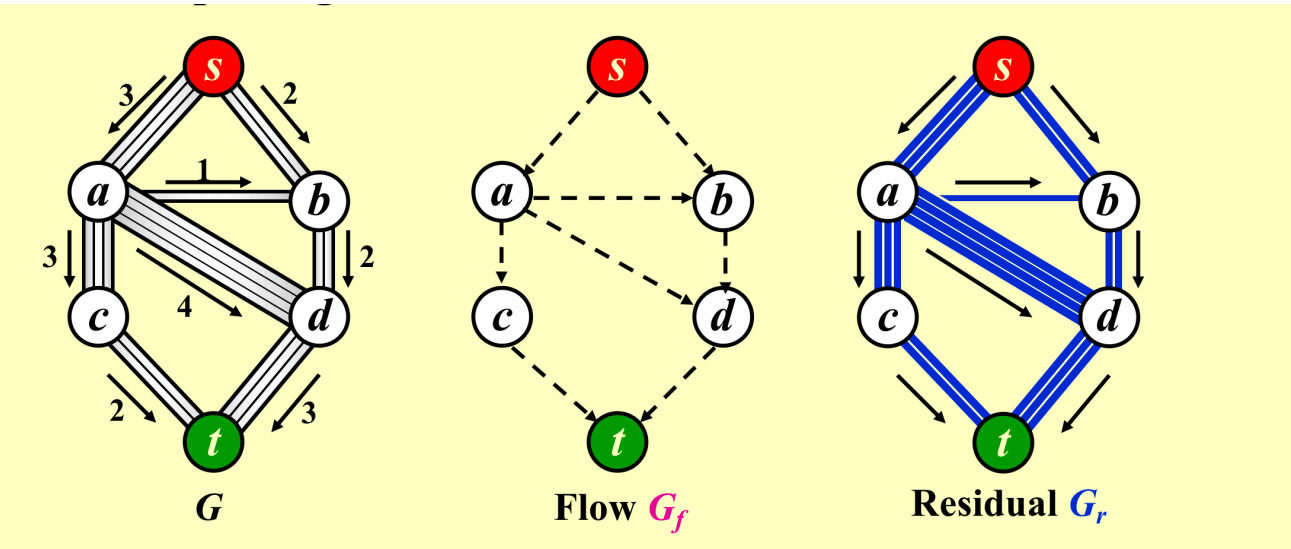
Method 1 **Use single-source algorithm for |V| times.**

$T = O(|V|^3)$ **– works fast on sparse graph.**

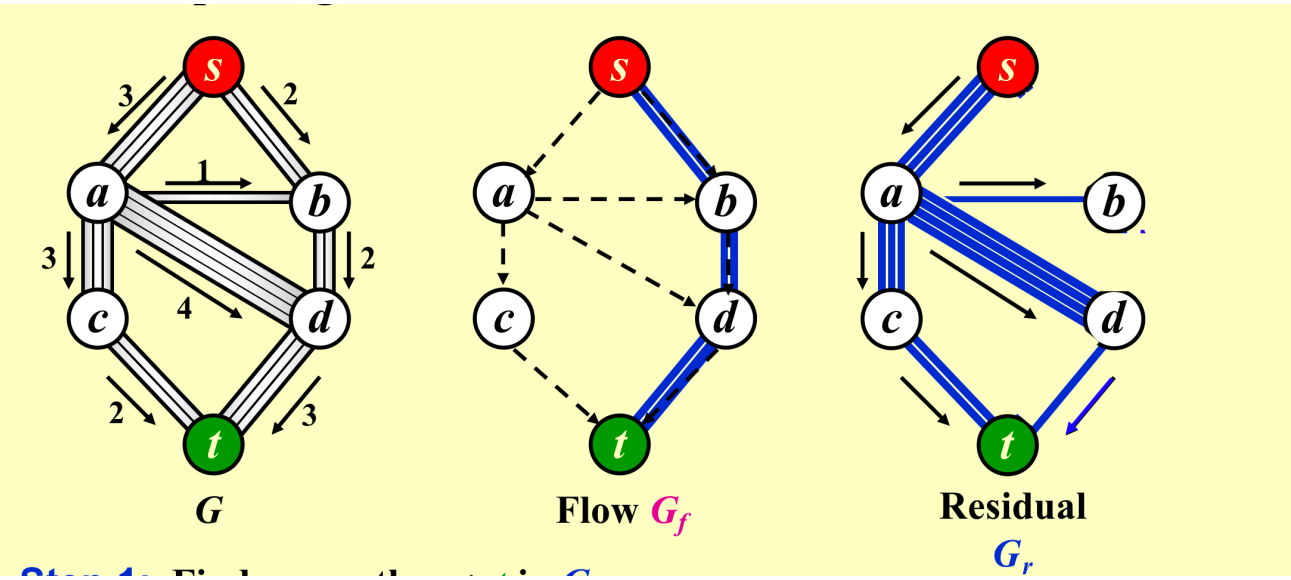
Method 2 **$O(|V|^3)$ algorithm given in Ch.10, works faster on dense graphs.**

4 Network Flow Problems

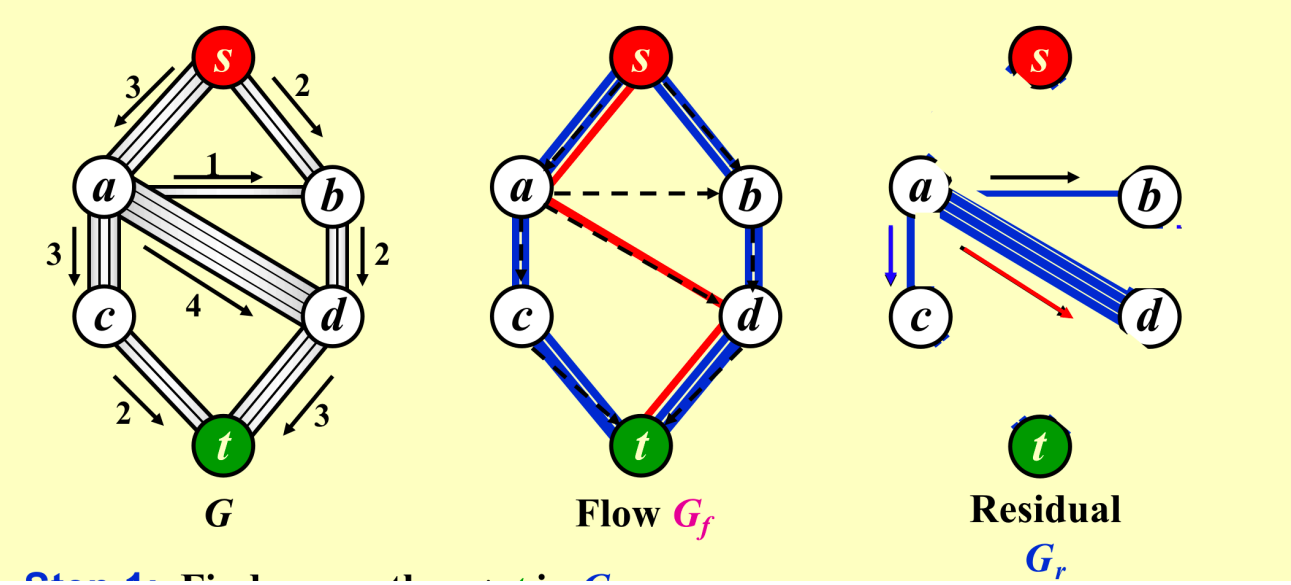
4.1 A Simple Algorithm



- Step 1: Find any path $s \in t$ in Gr ;



- Step 2: Take the minimum edge on this path as the amount of flow and add to Gf ;
- Step 3: Update Gr and remove the 0 flow edges; 加一条路径，就会在残差图里减掉一个

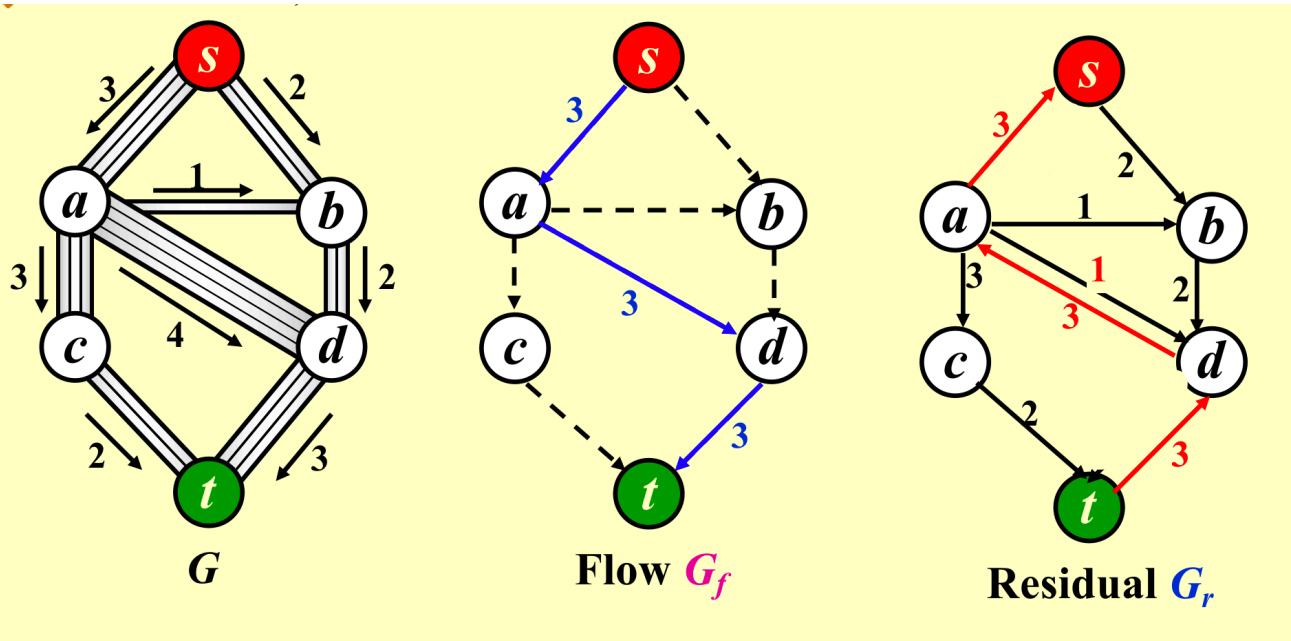


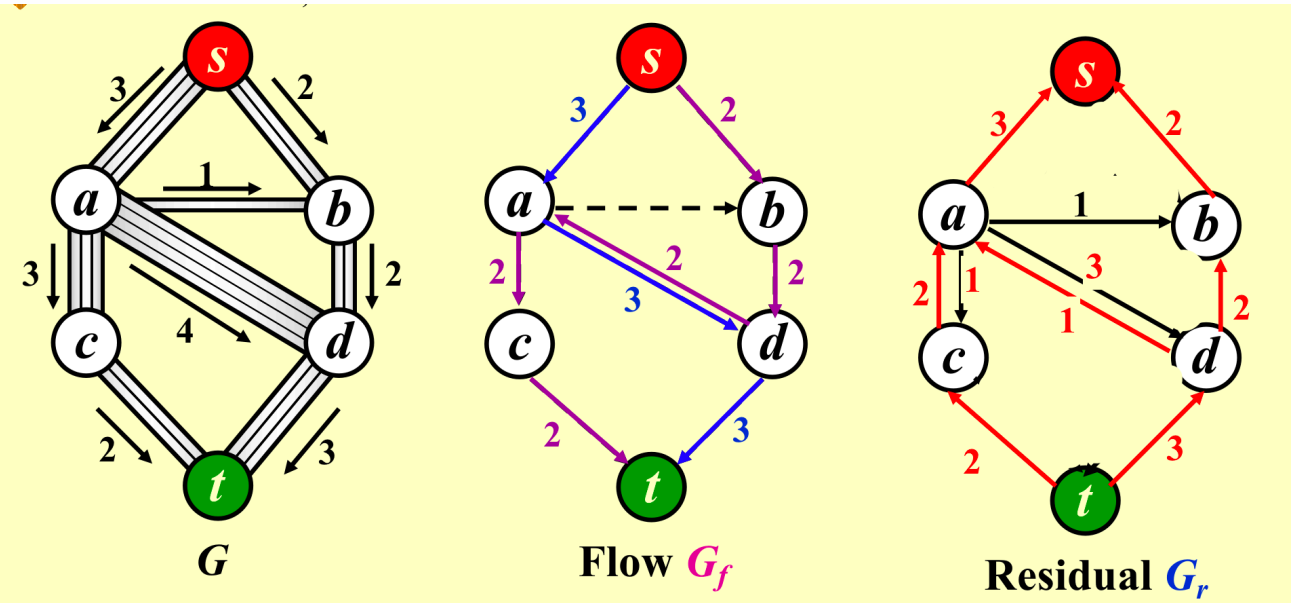
- Step 4:

```
if (there is a path s \in t in Gr )
  Goto Step 1;
else
  End.
```

- Problem：如果一开始的路不好，就会把后面的路都给影响到

4.2 A Solution – allow the algorithm to undo its decisions



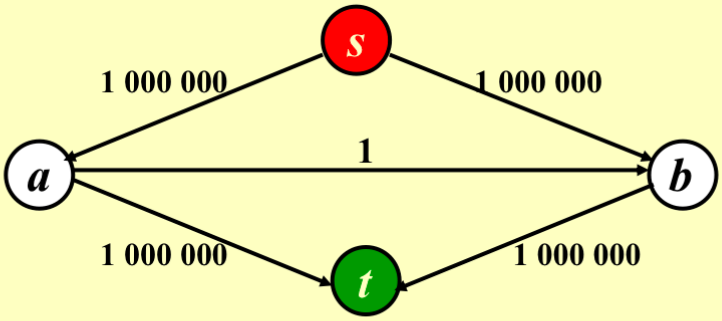


- For each edge (v, w) with flow fv, w in Gf , add an edge (w, v) with flow fv, w in Gr .
- 就是说设两个图，一个是flow的图，一个是残差图。之前的残差图对于选好的路径才用删除的做法，但是现在的残差图才用增加反向流来判断残差，是可修改的。实现了自取消的能力。
- 还是很巧妙的一种算法

4.3 Analysis

☞ An augmenting path can be found by an unweighted shortest path algorithm.

$T = O(\ f \cdot |E|\ )$ where f is the maximum flow.



☞ Always choose the augmenting path that allows the largest increase in flow. /* modify Dijkstra's algorithm */

$$T = T_{augmentation} * T_{find\ a\ path}$$
$$= O(|E| \log cap_{max}) * O(|E| \log |V|)$$
$$= O(|E|^2 \log |V|) \text{ if } cap_{max} \text{ is a small integer.}$$

☞ Always choose the augmenting path that has the least number of edges.

$$T = T_{augmentation} * T_{find\ a\ path}$$
$$= O(|E|) * O(|E| \cdot |V|) \quad \text{/* unweighted shortest path algorithm */}$$
$$= O(|E|^2 |V|)$$

Note:

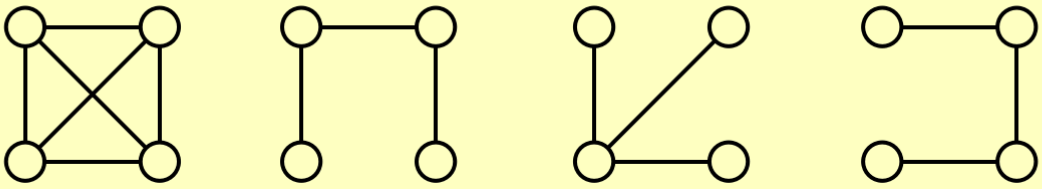
- If every $v \notin \{ s, t \}$ has either a single incoming edge of capacity 1 or a single outgoing edge of capacity 1, then time bound is reduced to $O(|E|\ |V|^{1/2})$.
- The *min-cost flow* problem is to find, among all maximum flows, the one flow of minimum cost provided that each edge has a cost per unit of flow.

5 Minimum Spanning Tree

§ 5 Minimum Spanning Tree

【Definition】 A *spanning tree* of a graph G is a tree which consists of V(G) and a subset of E(G)

〖Example〗 A complete graph and three of its spanning trees



Note:

- **The minimum spanning tree is a *tree* since it is acyclic -- the number of edges is |V| – 1.**
- **It is *minimum* for the total cost of edges is minimized.**
- **It is *spanning* because it covers every vertex.**
- **A minimum spanning tree exists iff G is *connected*.**
- **Adding a non-tree edge to a spanning tree, we obtain a *cycle*.**

- 首先要是个生成树，到了每个顶点；
- 生成树的边是顶点个数减1
- 边的总和的最小的认为是最小生成树

5.1 Greedy Method - Prim's Algorithm

Make the best decision for each stage, under the following constrains :

(1) we must use only edges within the graph;
(2) we must use exactly |V| - 1 edges;
(3) we may not use edges that would produce a cycle.

- 看ppt的图，和Dijkstra算法很像
- 复杂度和Dijkstra一样？？？

5.2 Kruskal's Algorithm – maintain a forest

```
void Kruskal ( Graph G )
{   T = { } ;
    while  ( T contains less than |V| - 1 edges && E is not empty ) {
        choose a least cost edge (v, w) from E ; /* DeleteMin 可以用
Heap */
        delete (v, w) from E ;
        if  ( (v, w) does not create a cycle in T ) // 可以验证两个端点不
是一个集合里的，用并查集
          add (v, w) to T ; /* Union / Find */
        else
          discard (v, w) ;
    }
    if  ( T contains fewer than |V| - 1 edges )
        Error ( "No spanning tree" ) ;
}
```

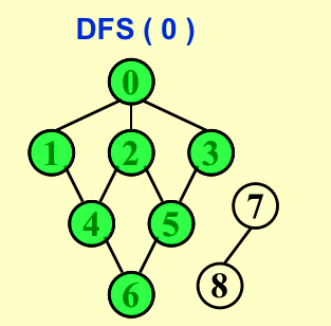
$$T = O(|E| \log |E|) \tag{9}$$

6 Applications of DFS

```
void DFS ( Vertex V )  /* this is only a template */
{   visited[ V ] = true;  /* mark this vertex to avoid cycles */
    for ( each W adjacent to V )
        if ( !visited[ W ] )
          DFS( W );
} /* T = O( |E| + |V| ) as long as adjacency lists are used */
```

- 就是先找到最深的节点。最多是每条边和每个节点都访问一次

6.1 Undirected Graph

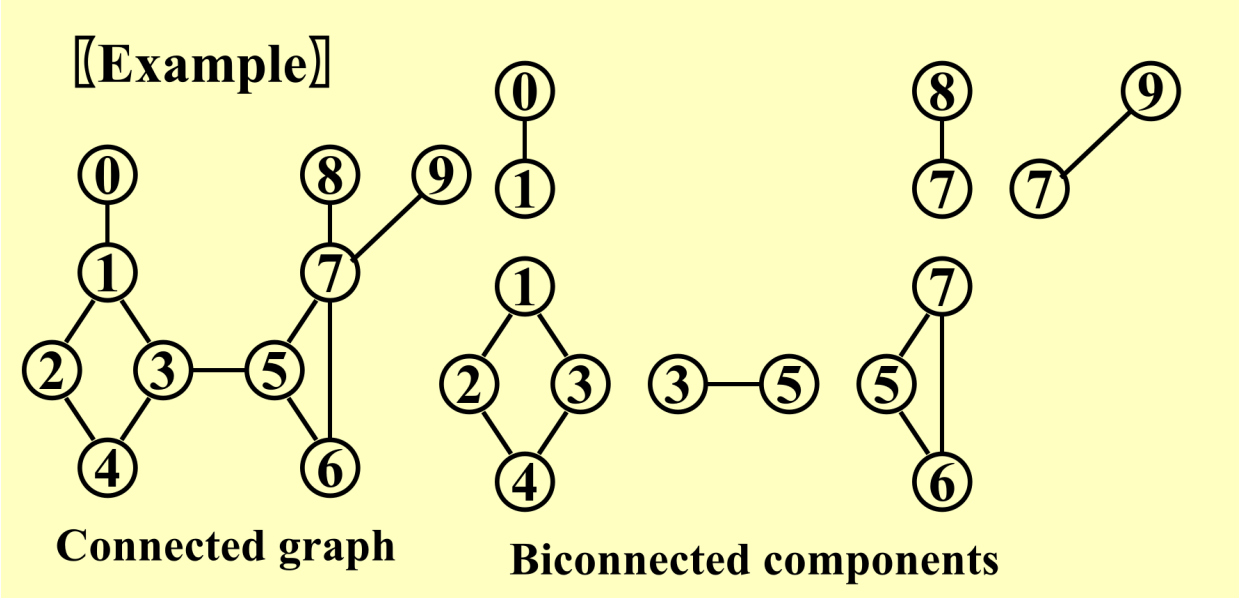



```
void ListComponents ( Graph G )
{   for ( each V in G )
        if ( !visited[ V ] ) {
          DFS( V ); // 一个DFS相当于把联通了的所有节点都访问一次
          printf("\n");
        }
}
// 0 1 4 6 5 2 3
// 7 8
```

- 其实有点像中序遍历
- 有几个回车符，就有几个联通元素

6.2 Biconnectivity

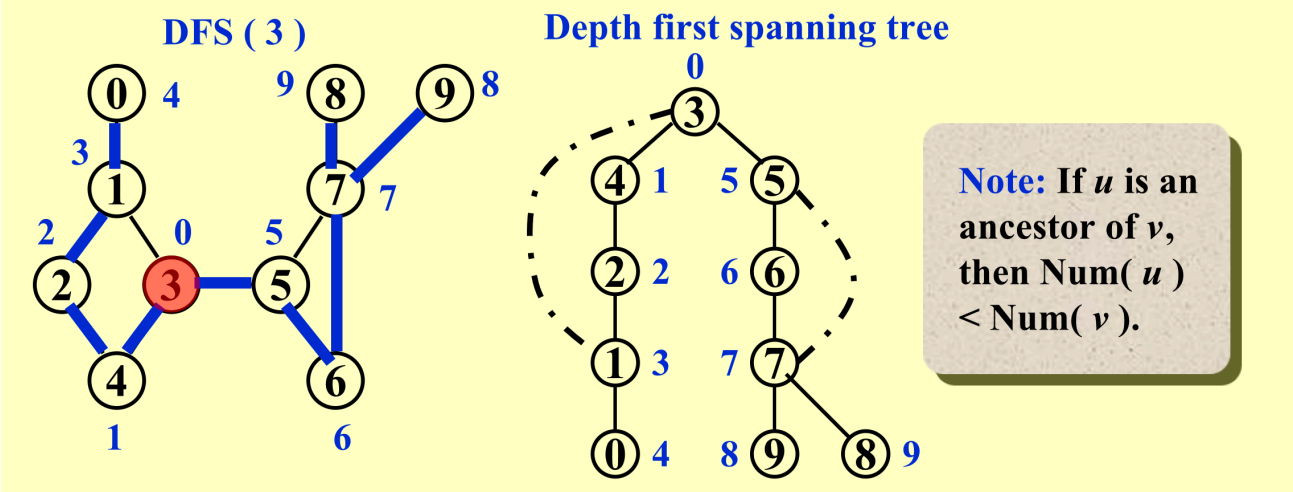
- v is an articulation point（关节点） if G′ = DeleteVertex(G, v) has at least 2 connected components. 关节点（就是关节）去除后，会变成至少两个联通分量。
- G is a biconnected graph if G is connected and has no articulation points.
- A biconnected component is a maximal biconnected subgraph.



- Note: No edges can be shared by two or more biconnected components. Hence E(G) is partitioned by the biconnected components of G.

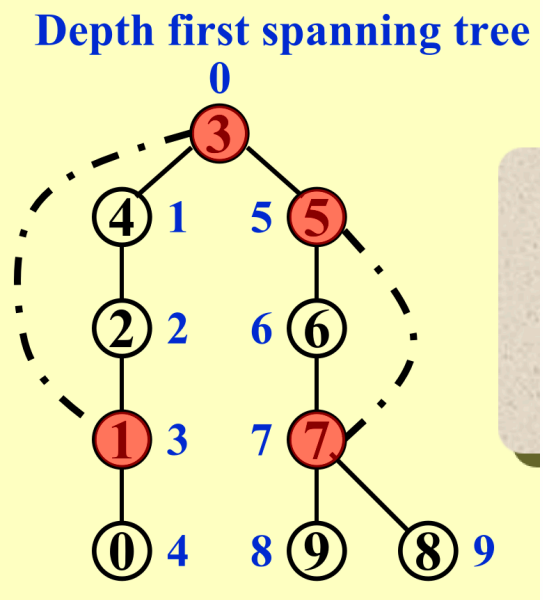
6.2.1 How to find articulation point

6.2.1.1 Use depth first search to obtain a spanning tree of G

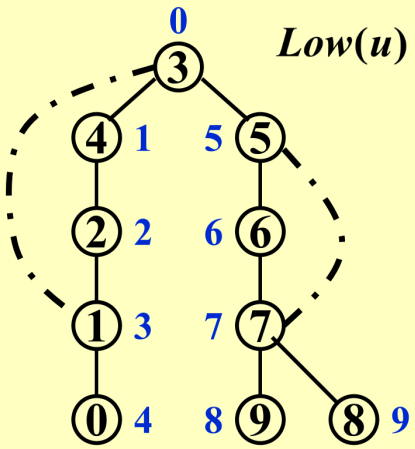


- 先用DFS找出spanning tree，根据DFS顺序标号
- 把缺失的边补齐，记作backedge

6.2.1.2 Find the articulation points in G



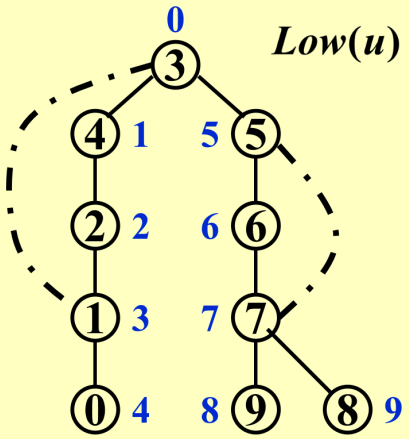
- The root is an articulation point if it has at least 2 children
- Any other vertex u is an articulation point iff u has at least 1 child, and it is impossible to move down at least 1 step and then jump up to u's ancestor. 判断是不是在环路上，不是环路关键节点。



$$Low(u) = \min\{Num(u), \min\{Low(w) \mid w \text{ is a child of } u\}, \min\{Num(w) \mid (u,w) \text{ is a back edge}\}\}$$

vertex	0	1	2	3	4	5	6	7	8	9
Num	4	3	2	0	1	5	6	7	9	8
Low	4	0	0	0	0	5	5	5	9	8

- 从根节点开始
- 1、7节点有back edge
- 2、4、6就是用Low(w) child
- 0、9、8用的是Num(u)



$$Low(u) = \min\{Num(u), \min\{Low(w) \mid w \text{ is a child of } u\}, \min\{Num(w) \mid (u,w) \text{ is a back edge}\}\}$$

vertex	0	1	2	3	4	5	6	7	8	9
Num	4	3	2	0	1	5	6	7	9	8
Low	4	0	0	0	0	5	5	5	9	8

Therefore, *u* is an articulation point iff
(1) *u* is the root and has at least 2 children; or
(2) *u* is not the root, and has at least 1 child such that *Low(child) ≥ Num(u)*.

Please read the pseudocodes on p.327 and p.329 for more details.

6.3 Euler Circuits

- Draw each line exactly once without lifting your pen from the paper – Euler tour（一笔画）
  - An Euler tour is possible if there are exactly two vertices having odd degree. One must start at one of the odd-degree vertices.
- Draw each line exactly once without lifting your pen from the paper, AND finish at the starting point – Euler curcuit（欧拉环，一笔画+回到起点）
  - An Euler circuit is possible only if the graph is connected and each vertex has an even degree.
- 具体做法
- Note:
  - The path should be maintained as a linked list.
  - For each adjacency list, maintain a pointer to the last edge scanned.

$$T = O(|E| + |V|) \tag{10}$$