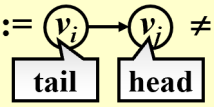
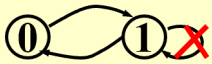
Graph

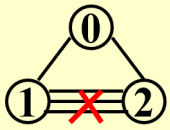
Theory

Terms

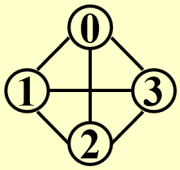
- **G(V, E) where G ::= graph, V = V(G) ::= finite nonempty set of vertices, and E = E(G) ::= finite set of edges.**
- **Undirected graph: $(v_i , v_j) = (v_j , v_i)$ ::= the same edge.**
- **Directed graph (digraph): $< v_i , v_j > ::=$** 

$\neq < v_j , v_i >$
- **Restrictions :**
  (1) Self loop is illegal.
  (2) Multigraph is not considered

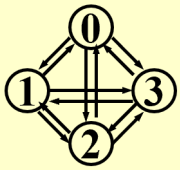




- **Complete graph: a graph that has the maximum number of edges**

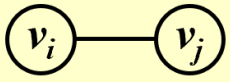


# of V = n $\Rightarrow$
# of E = $C_n^2 = \frac{n(n-1)}{2}$

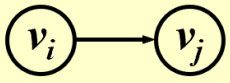


# of V = n $\Rightarrow$
# of E = $P_n^2 = n(n-1)$

- 由Edge和Vertices组成
- 有向图和无向图，注意边的表达方式
- FDS中的图，不研究self loop，不研究multigraph
- Complete graph
  - undirected: $\frac{n(n-1)}{2}$
  - directed: $n(n-1)$



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

- Edge, vertices之间关系
  - 无向图：adjacent, incident on
  - 有向图：adjacent to, adjacent from, incident on

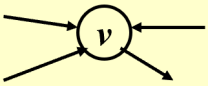
Subgraph G' $\subset$ G ::= V(G') $\subseteq$ V(G) && E(G') $\subseteq$ E(G)

- 图与图之间关系：点和边都是子集

- **Path ($\subset$ 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**

- 路径：sequence of vertices
- 路径长度：number of edges
- Simple path：中途无环路 - distinct vertices
- Cycle：起点和终点重合的simple path
- 连通图
  - 点与点联通：有path
  - 无向图联通：every pair of vertices are connected

- **(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**

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

- Connected Component of an undirected G
  - 最大连通子图
- Tree： Special Graph

  连通、无环(acyclic)
- DAG(Directed acyclic graph)

- 有向图的连通
  - strongly connected: 双向连通
    - Strongly connected component
  - weakly connected: 转换为undirected连通
- Degree
  - 有向图：in-degree, out-degree
  - Handshaking theorem

Representations

- 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.

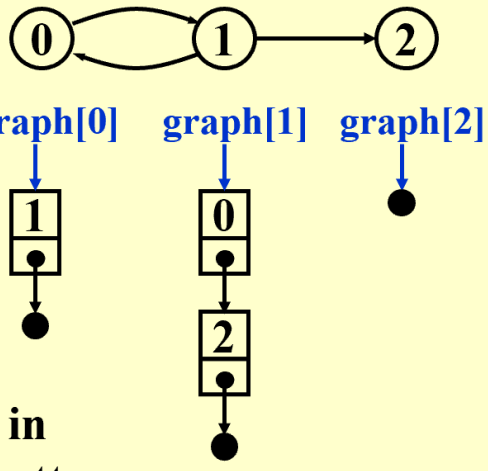
$$\text{degree}(i) = \sum_{j=0}^{n-1} \text{adj\_mat}[i][j] \quad \text{(if G is undirected)}$$

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

- Adjacency Lists

  本质上是matrix按行切割

〖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

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

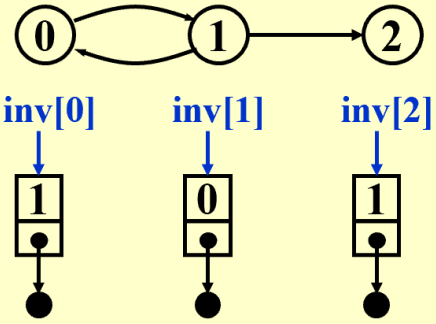
If G is **directed**, we need to find in-degree(v) as well.

求in-degree: Inverse adjacency lists / multilist

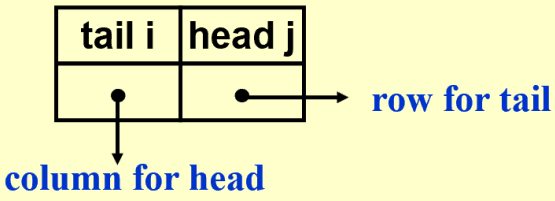
- 本质上是把matrix按列切割

Method 1 Add inverse adjacency lists.

〖Example〗

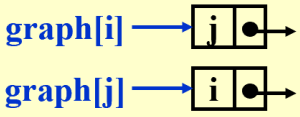


Method 2 Multilist (Ch 3.2) representation for adj_mat[i] [j]

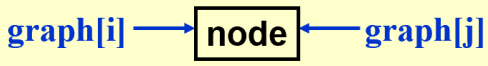


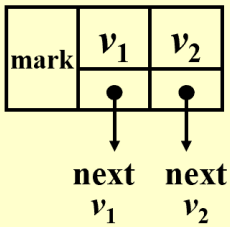
- Adjacency Multilists

In adjacency list, for each (i, j) we have two nodes:

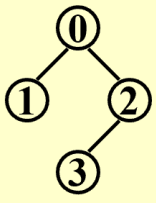


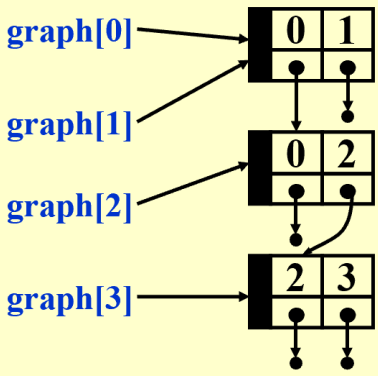
Now let's combine the two nodes into one:



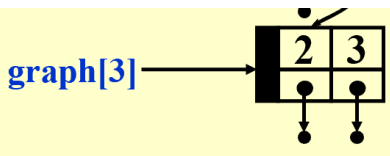


〖Example〗





Weighted Edges



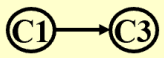
- adj_mat [i] [j] = weight
- adjacency lists \ multilists : add a **weight** field to the node.

Topological Sort

〖Example〗 Courses needed for a computer science degree at a hypothetical university

Course number	Course name	Prerequisites
C1	Programming I	None
C2	Discrete Mathematics	None
C3	Data Structure	C1, C2
C4	Calculus I	None
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Language	C3
C9	Operating Systems	C7, C8
C10	Programming Languages	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithms	C13
C15	Numerical Analysis	C6

- AOV(Activity on the Vertices)

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

  - AOV vs AOE

    AOV: 事件发生在点上，忽略事件的权重

    AOE：事件发生在边上，有权重

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

> predecessor: 有路从i到j
>
> immediate predecessor: i到j，且i与j相邻
>
> 本质上就是Partial Order

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.

> 此处规定了irreflexive的partial order，防止死循环

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

【Definition】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.

〖Example〗 One possible suggestion on course schedule for a computer science degree could be:

Course number	Course name	Prerequisites
C1	Programming I	None
C2	Discrete Mathematics	None
C4	Calculus I	None
C3	Data Structure	C1, C2
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C15	Numerical Analysis	C6
C8	Assembly Language	C3
C10	Programming Languages	C7
C9	Operating Systems	C7, C8
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C11	Compiler Design	C10
C14	Parallel Algorithms	C13

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.

> 解决问题的核心：Find in-degree with 0

Basic Implementation

```
void Topsort( Graph G )
{  int  Counter;
   Vertex  V, W;
   for ( Counter = 0; Counter < NumVertex; Counter ++ ) {
         V = FindNewVertexOfDegreeZero( );   /* O( |V| ) */
         if ( V == NotAVertex ) {
            Error ( "Graph has a cycle" );   break;  }
         TopNum[ V ] = Counter; /* or output V */
         for ( each W adjacent to V )
            Indegree[ W ] – – ;
   }
}
```

$T = O(|V|^2)$

$$T = O(|V|^2 + |E|) = O(|V|^2 + |V|(|V| - 1)) = O(|V|^2)$$

> 这个也可以用来确定图中是否有环

Improvement

☞ **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 );
   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 */
}
```

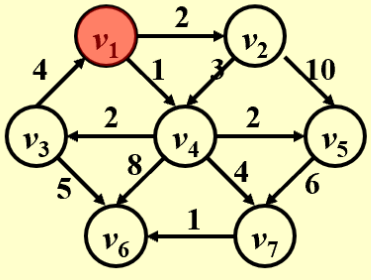
> 用一个容器暂存单次循环找到的所有0 in-degree的节点
>
> $T = O(|V| + |E|)$

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 $\sum_{e_i \subset P} c(e_i)$ (also called **weighted path length**).

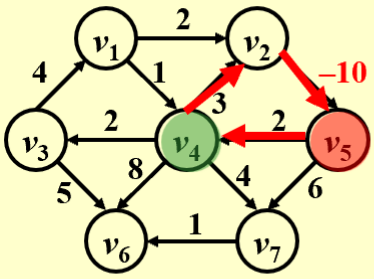
Single-Source Shortest-Path Problem

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.



> 计算点到点的最短路和 计算点到所有点的最短路 开销是一样的

因为需要探索所有可能的路径才能得出最短路的结论

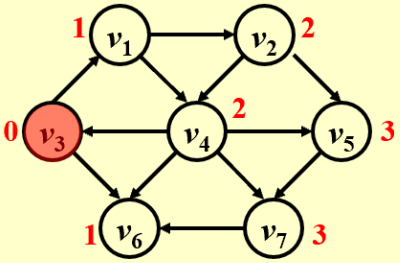


Note: If there is no negative-cost cycle, the shortest path from *s* to *s* is defined to be zero.

Unweighted Shortest Path

Equivalent to BFS

❖ **Sketch of the idea**



0: ☞ v_3

1: ☞ v_1 and v_6

2: ☞ v_2 and v_4

3: ☞ v_5 and v_7

Breadth-first search

❖ **Implementation**

Table[i].Dist ::= distance from *s* to v_i /* initialized to be ∞ except for *s* */

Table[i].Known ::= 1 if v_i is checked; or 0 if not

Table[i].Path ::= for tracking the path /* initialized to be 0 */

Distance: cost

Known: 是否被访问

Path: 上一个节点，用来tracking the path

```
void Unweighted( Table T )
{  int  CurrDist;
   Vertex  V, W;
   for ( CurrDist = 0; CurrDist < NumVertex; CurrDist ++ ) {
      for ( each vertex V )
         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 */
}
```

The worst case: 

> 时间复杂度：$O(|V|^2)$

> Known可以直接由Distance表示

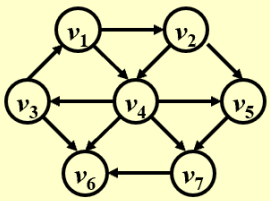
Improvement

类似于Topological Sort的优化，用Container暂存

此处的Container**只能选择Queue**，保持节点的顺序

❖ **Improvement**

```
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：$O(|V| + |E|)$

Dijkstra's Algorithm

> For weighted shortest path

➢ *Dijkstra*'s Algorithm (for weighted shortest paths)

Let S = { s and v_i's whose shortest paths have been found }

For any $u \notin S$, define distance [u] = minimal length of path { $s \rightarrow (v_i \in S) \rightarrow u$ }. If the paths are generated in non-decreasing order, then

① the shortest path must go through ONLY $v_i \in S$;

② u is chosen so that distance[u] = min{ $w \notin S$ | distance[w] } (If u is not unique, then we may select any of them) ; /* Greedy Method */

③ if distance [u_1] < distance [u_2] and we add u_1 into S, then distance [u_2] may change. If so, a shorter path from s to u_2 must go through u_1 and distance' [u_2] = distance [u_1] + length($< u_1, u_2 >$).

```
void Dijkstra( Table T )
{  /* T is initialized by Figure 9.30 on p.303 */
   Vertex  V, W;
   for ( ; ; ) {
      V = smallest unknown distance vertex;
      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( ; ; ) */
}
```

❖ **Implementation 1**

```
V = smallest unknown distance vertex;
/* simply scan the table – O( |V| ) */
```

$T = O(|V|^2 + |E|)$ — Good if the graph is dense

❖ **Implementation 2**

```
V = smallest unknown distance vertex;
/* keep distances in a priority queue and call DeleteMin – O( log|V| ) */
Decrease( T[ W ].Dist  to  T[ V ].Dist + Cvw );
   /* Method 1: DecreaseKey – O( log|V| ) */
```

$T = O(|V| \log|V| + |E| \log|V|) = O(|E| \log|V|)$ — Good if the graph is sparse

> 使用Heap + DecreaseKey：(Good for sparse graph)
>
> $|V| \log |V|$：对每一个vertex，需要进行deleteMin
>
> $|E| \log |V|$：对每一个adjacent，需要进行一次DecreaseKey

```
/* Method 2: insert W with updated Dist into the priority queue */
/* Must keep doing DeleteMin until an unknown vertex emerges */
```

$T = O(|E| \log|V|)$ but requires |E| DeleteMin with |E| space

> 直接Insert

Graph 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 );
          } /* end-if update */
   } /* end-while */
   DisposeQueue( Q ); /* free memory */
}   /* negative-cost cycle will cause indefinite loop */
```

时间复杂度：$O(|V||E|)$

Acyclic Graphs

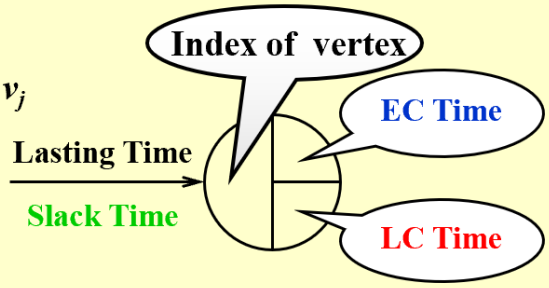
Acyclic Graphs

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.

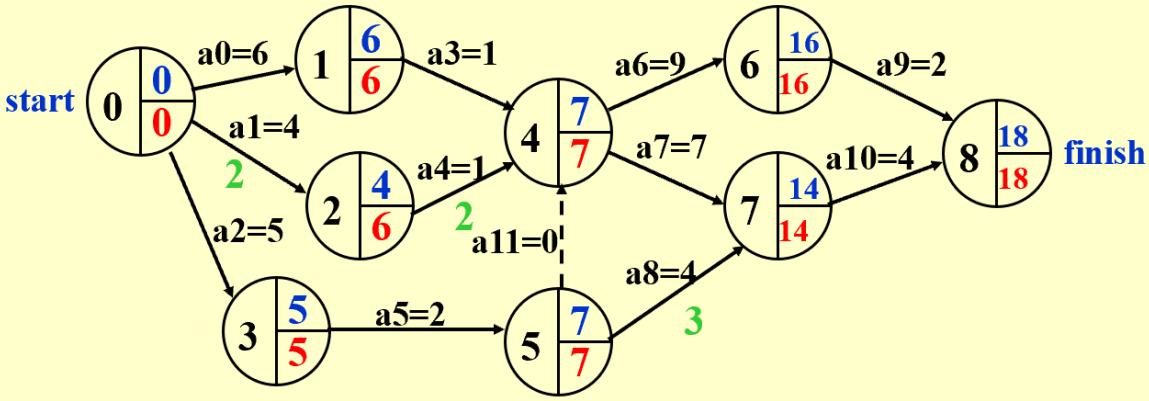
$T = O(|E| + |V|)$ and no priority queue is needed.

- 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.

2. All-Pairs Shortest Path Problem

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

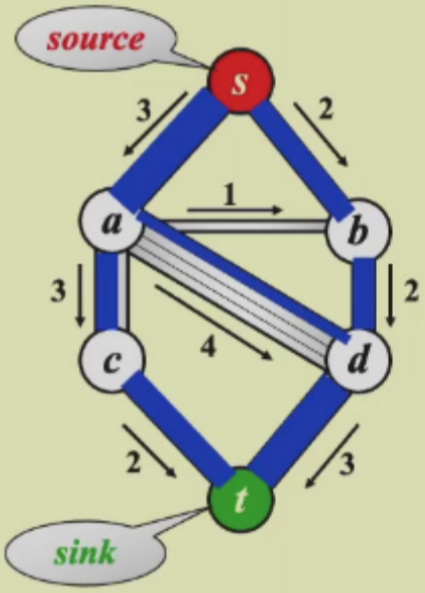
Method 1 Use single-source algorithm for |V| times.

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

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

Network Flow Problems

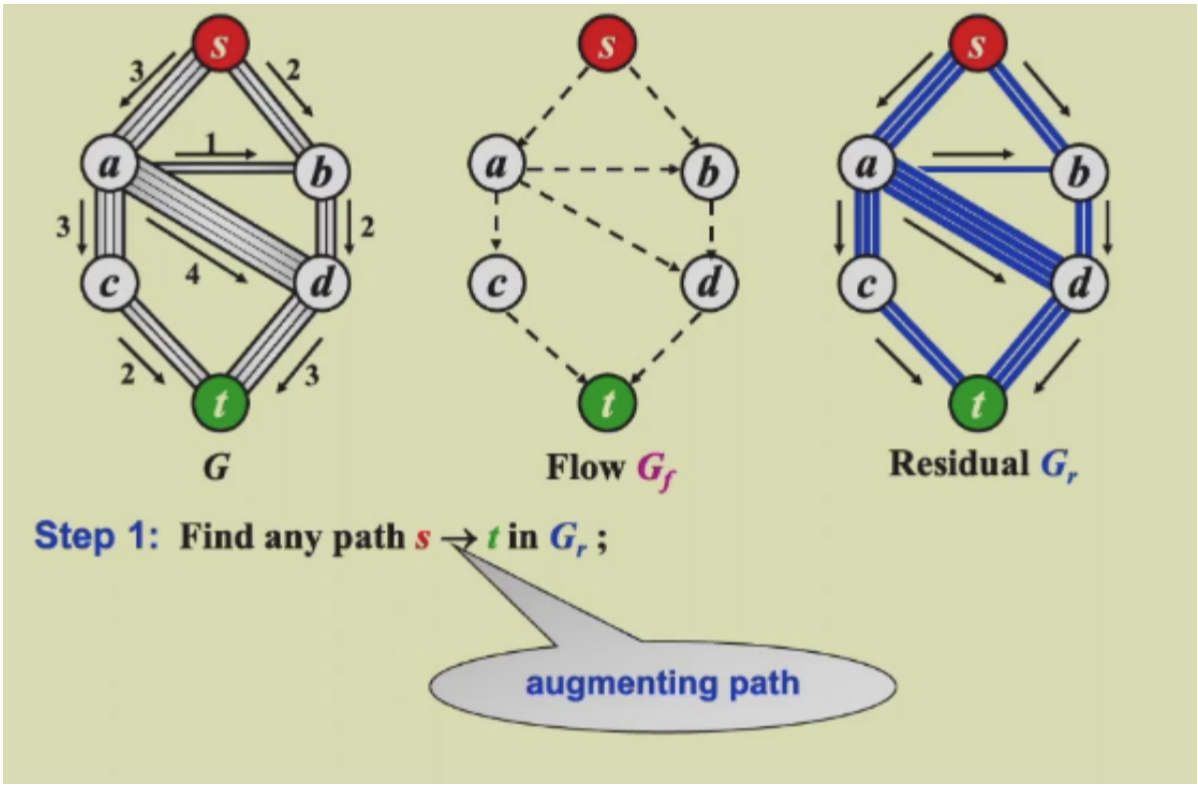
〖Example〗 Consider the following network of pipes:

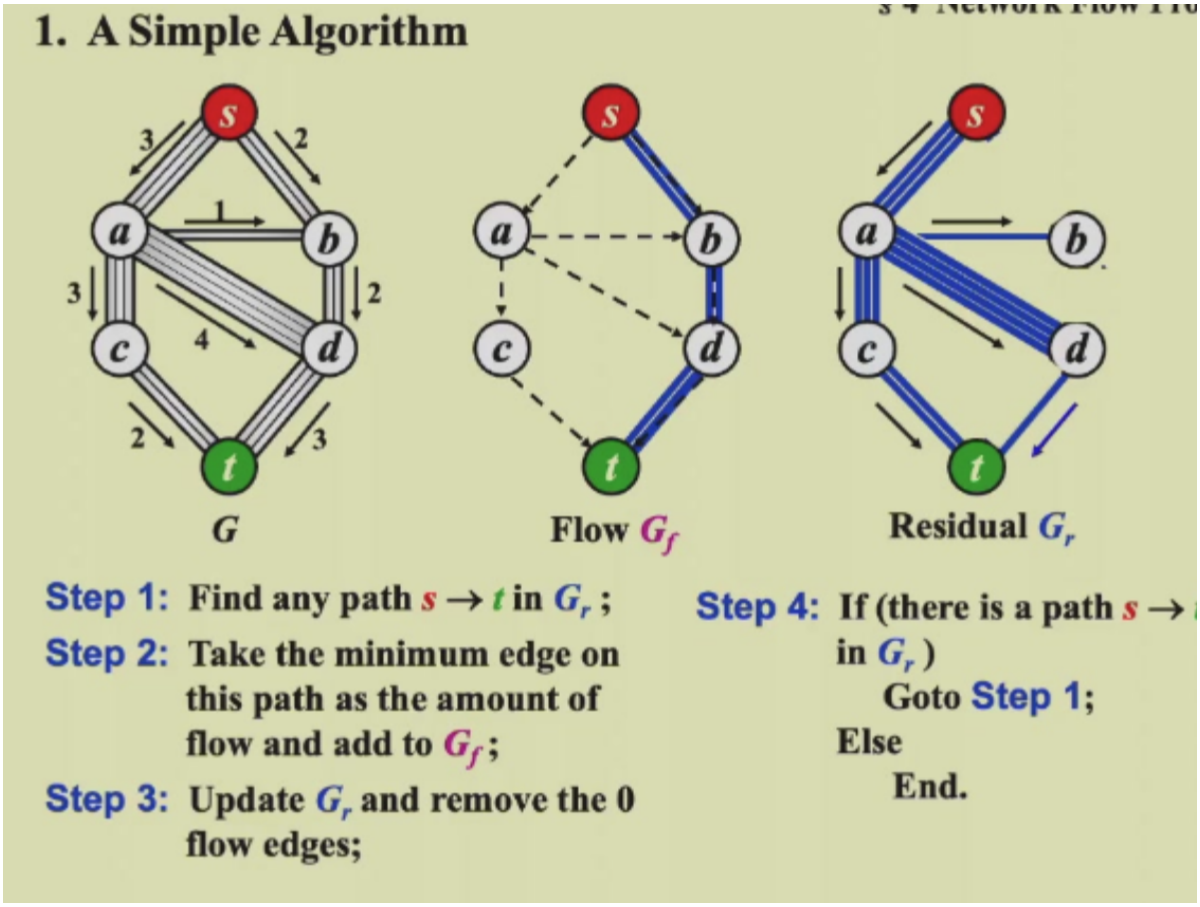


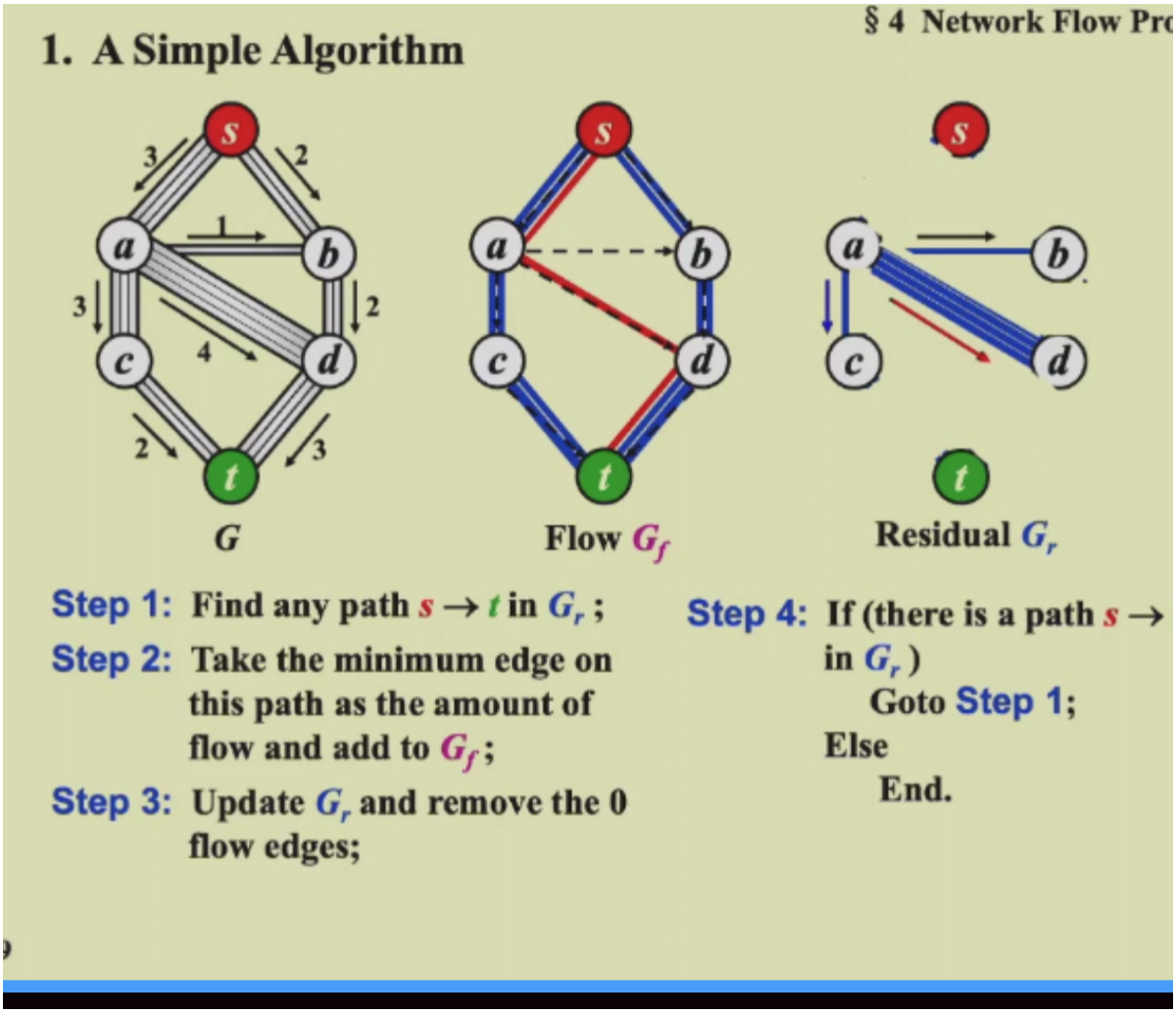
Note: Total coming in (v) ≡ Total going out (v) where $v \notin \{ s, t \}$

Determine the maximum amount of flow that can pass from s to t.

Simple Algorithm

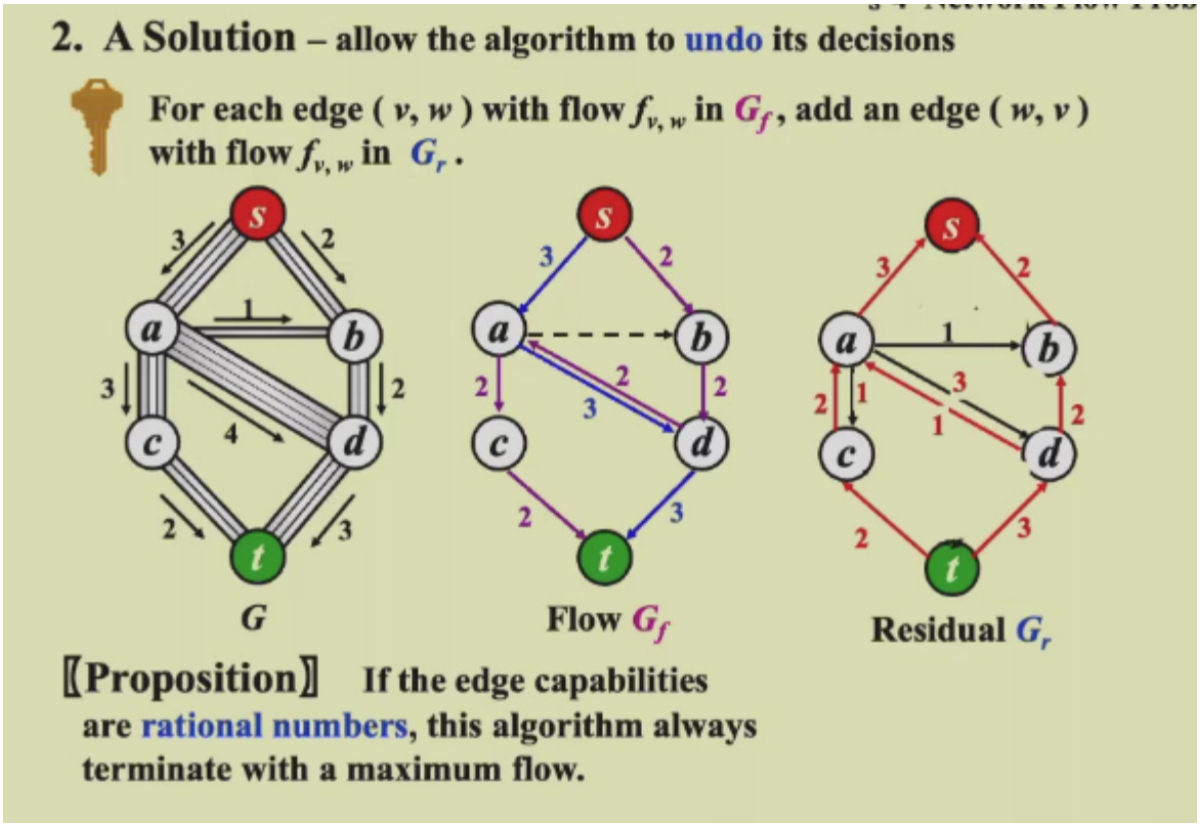






> 在greedy中，可能只会找到局部最优解，而找不到全局最优解
>
> 例如：先找sadt，找不到局部最优解

Solution



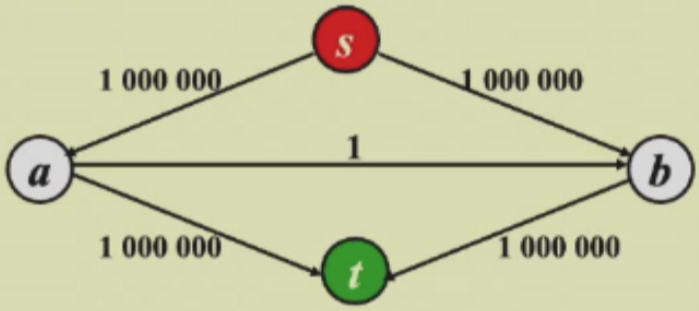
> 证明一定能找到局部最优解

Analysis

3. Analysis (If the capacities are all integers)

- 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:

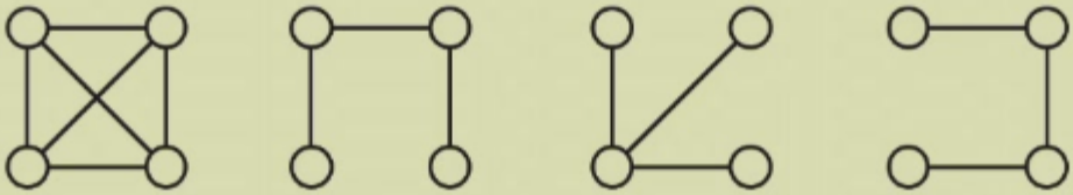
- If every $v \notin \{ s, t \}$ has either a single incoming edge of capacity 1 or a single outgoing edge of capacity 1, then time bound is reduced to $O(|E| |V|^{1/2})$.
- The *min-cost flow* problem is to find, among all maximum flows, the one flow of minimum cost provided that each edge has a cost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

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)

> 前提：图要connected

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



> Spanning Tree 不唯一

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*.

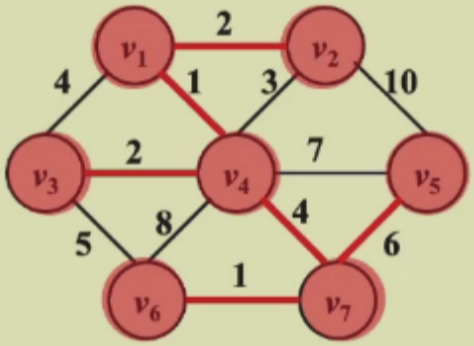
Prim's Algorithm

Greedy Method 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.

1. *Prim*'s Algorithm – grow a tree

/* very similar to Dijkstra's algorithm */



Kruskal's Algorithm

把所有的边放到堆里，疯狂deleteMin

选到成环的丢弃，不成欢的加进去，直到有|V|-1条边

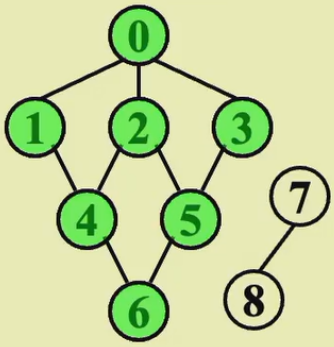
```
void Kruskal ( Graph G )                          T = O( |E| log |E| )
{  T = { } ;
   while  ( T contains less than |V| –1 edges
              && E is not empty ) {
      choose a least cost edge (v, w) from E ;   /* DeleteMin */
      delete (v, w) from E ;
      if  ( (v, w) does not create a cycle in T )
          add (v, w) to T ;     /* Union / Find */
      else
          discard (v, w) ;
   }
   if  ( T contains fewer than |V| –1 edges )
      Error ( "No spanning tree" ) ;
}
```

Application 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 */
```

1. Undirected Graphs

DFS (0)

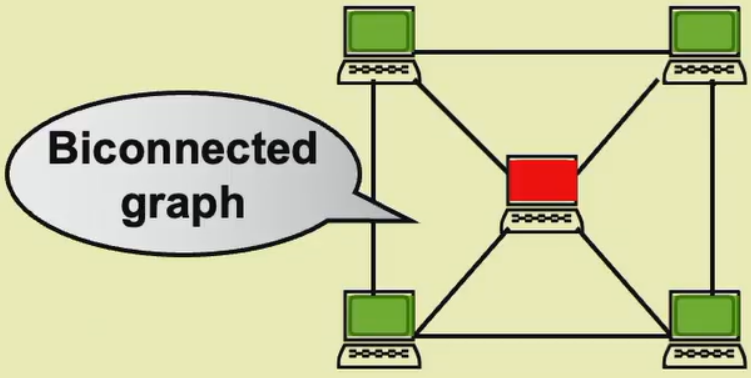



```
void ListComponents ( Graph G )
{   for ( each V in G )
        if ( !visited[ V ] ) {
            DFS( V );
            printf("\n");
        }
}
```

0 1 4 6 5 2 3
7 8

> 用于列出所有connected component

Biconnectivity



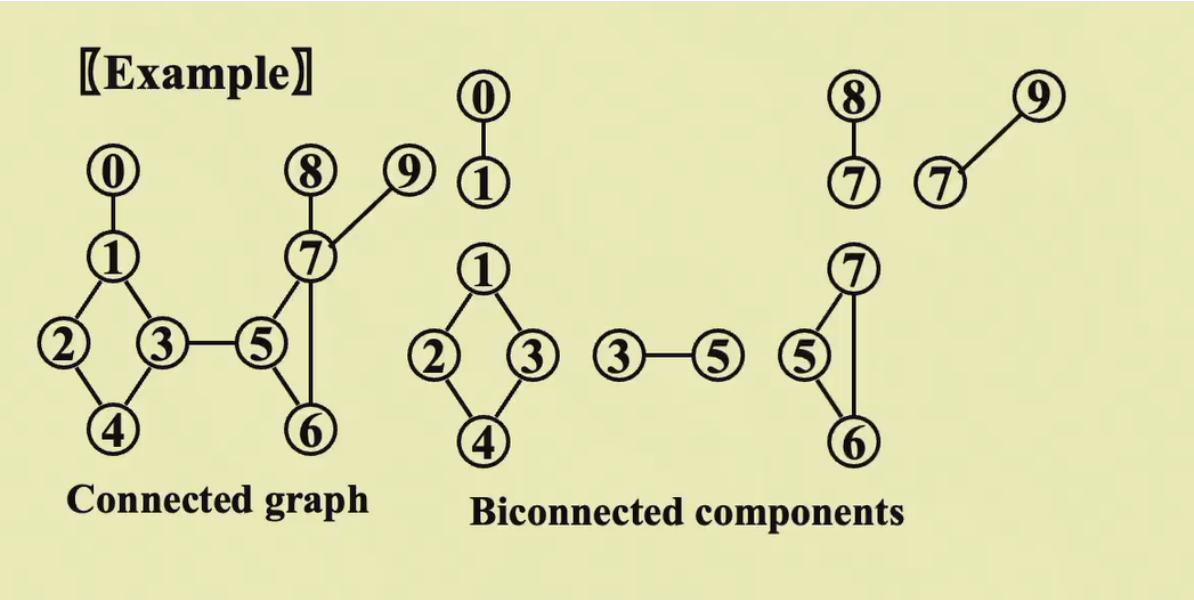
v is an articulation point if G' = DeleteVertex(G, *v*) has at least 2 connected components.

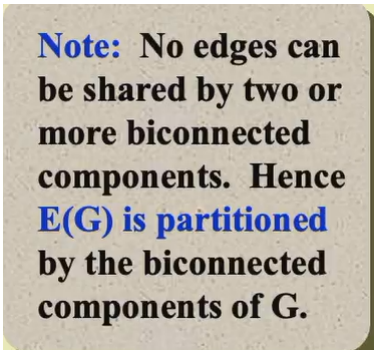
> 关节点：删除后分成多个connected component

G is a biconnected graph if G is connected and has no articulation points.

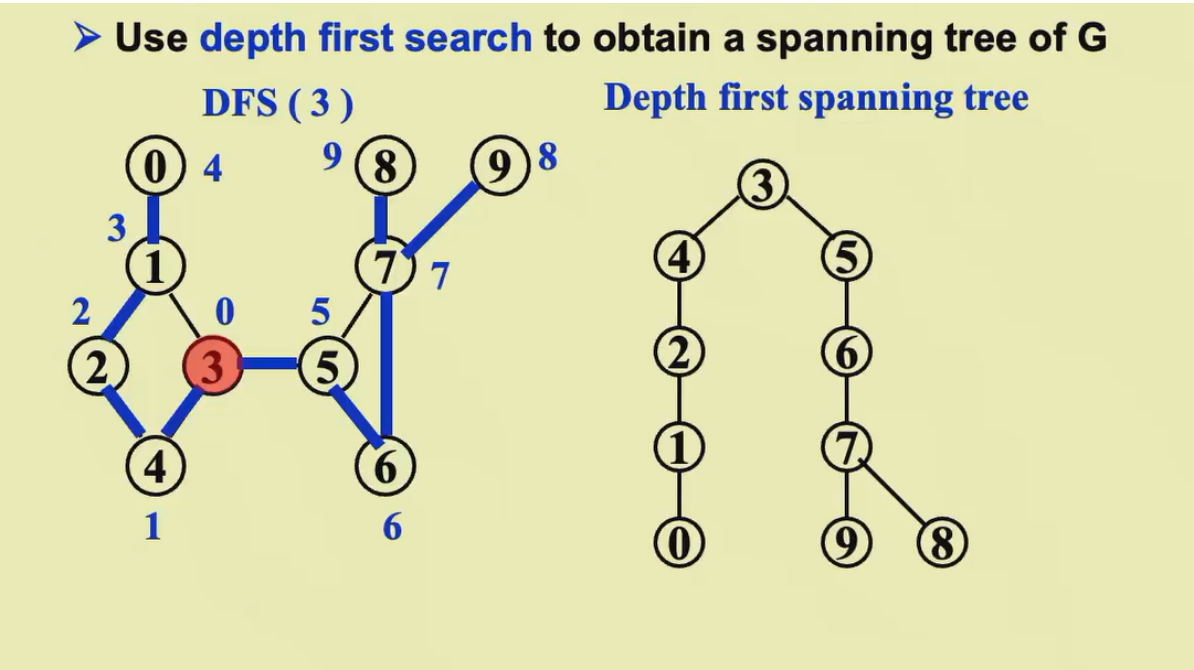
> 双向连通图：不存在关节点

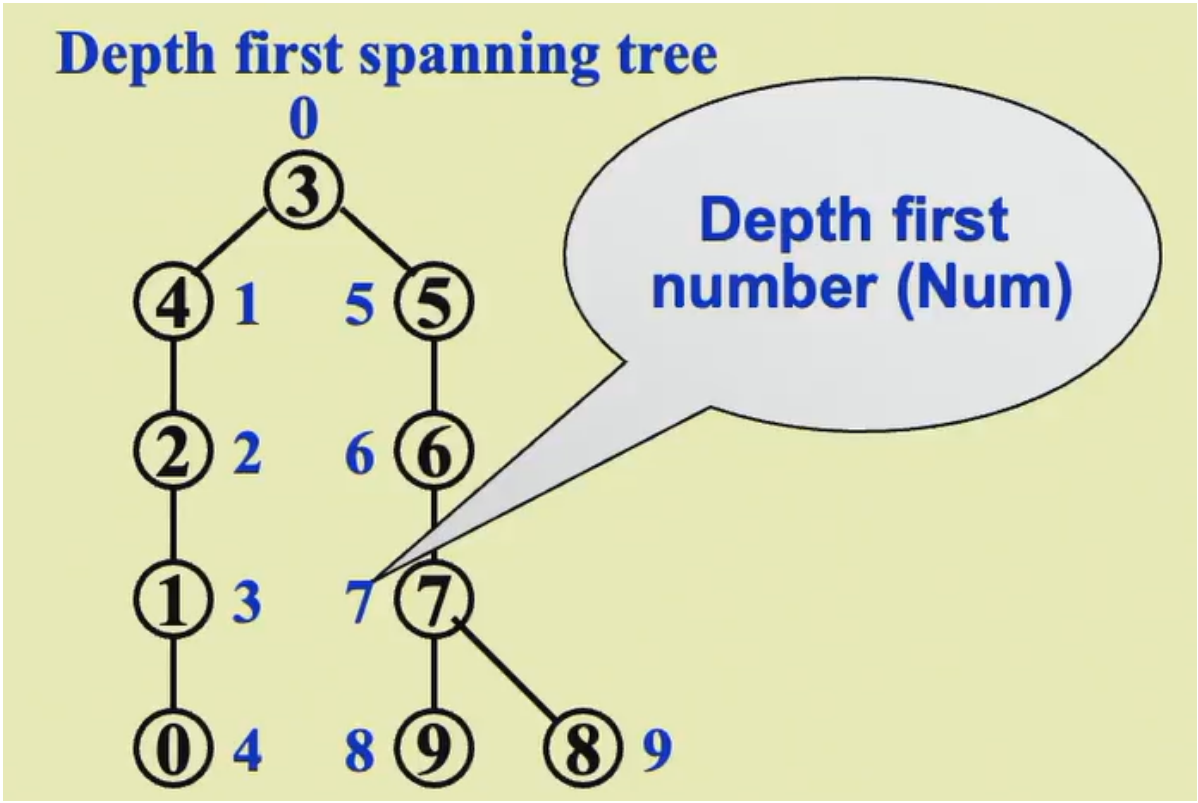
- biconnected component is a maximal biconnected subgraph



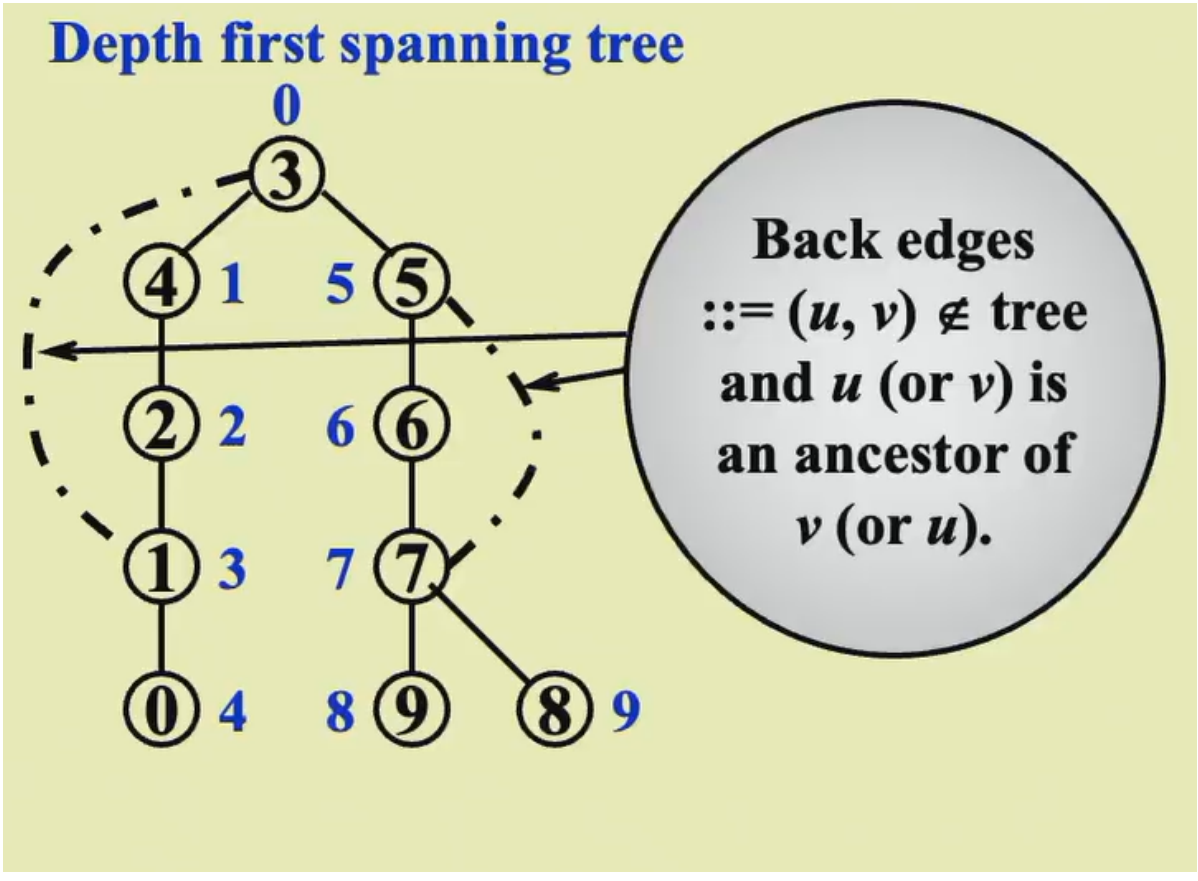


biconnected components 之间可以公用节点，但不能公用边。因此可以看做边的分割

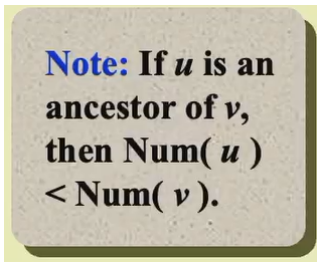




> Depth first number: 访问顺序

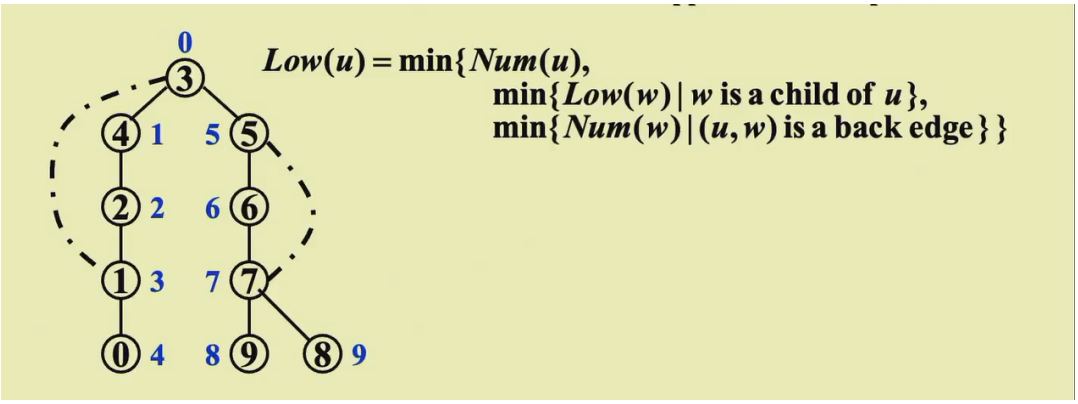


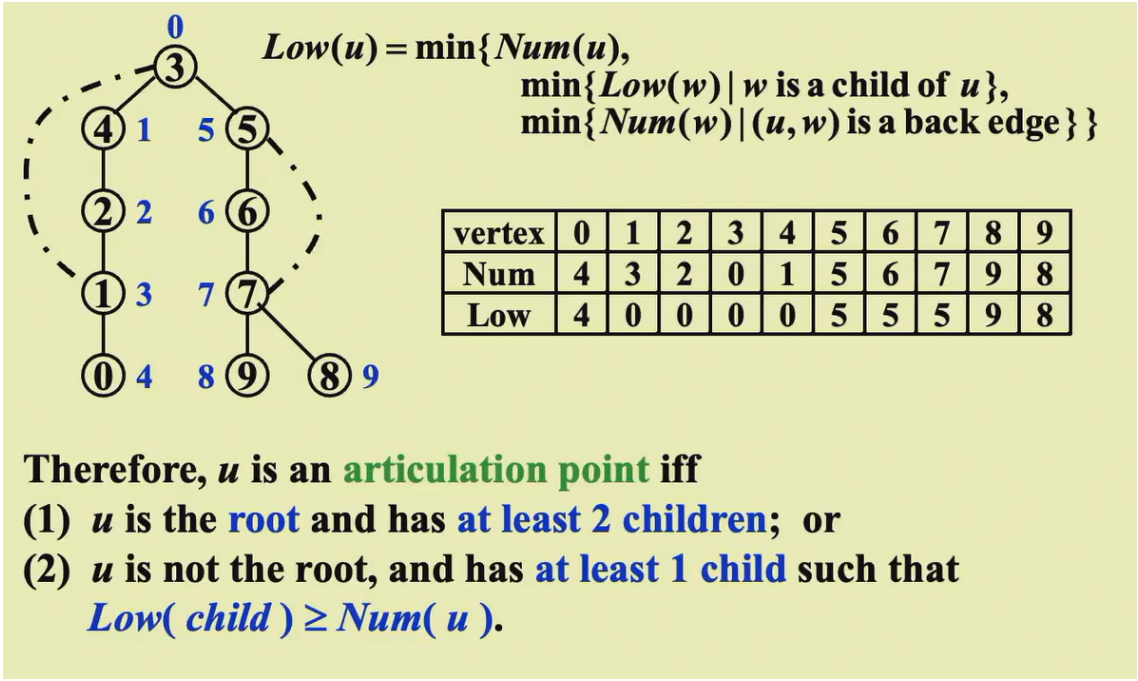
> 把这些没加上去的边加上去



> num小，level小

- 如何找关节点
  - 如果根节点有两个及以上的children，根节点就是关节点
  - 对于其他的点，至少有一个child，且it's impossible to move down at least 1 step and then jump up to u's ancestor
    - move down：不能走parent，也不能走从自己出发的backedge ❓
- 如何实现
  - low number





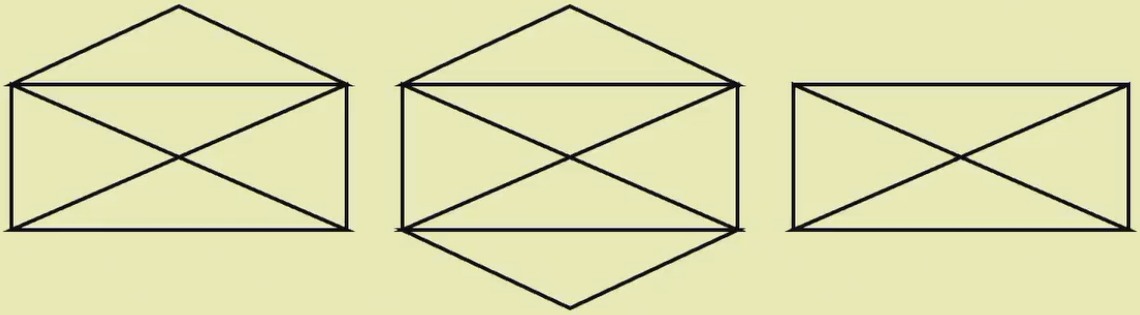
vertex	0	1	2	3	4	5	6	7	8	9
Num	4	3	2	0	1	5	6	7	9	8
Low	4	0	0	0	0	5	5	5	9	8

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)*.

> pseudocode 参考课本327 & 329

Euler Circuit

3. Euler Circuits

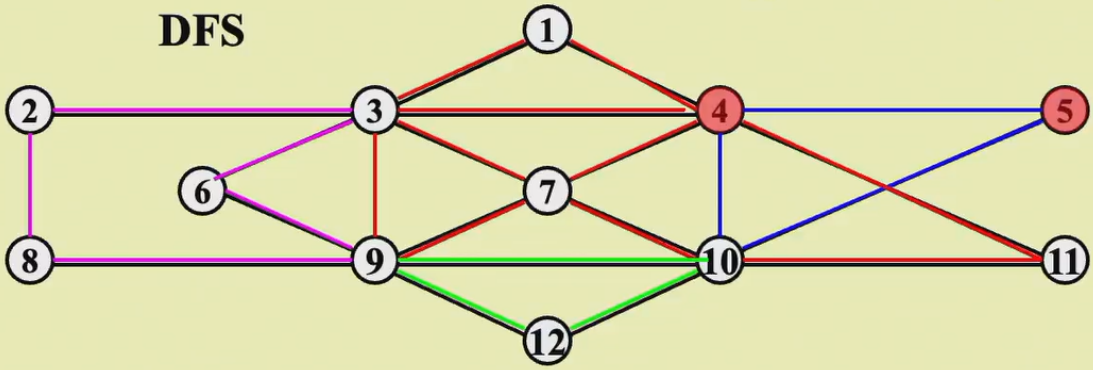


Draw each line exactly once without lifting your pen from the paper – *Euler tour*

Draw each line exactly once without lifting your pen from the paper, AND finish at the starting point – *Euler curcuit*

〖Proposition〗 An Euler circuit is possible only if the graph is connected and each vertex has an even degree.

〖Proposition〗 An Euler tour is possible if there are exactly two vertices having odd degree. One must start at one of the odd-degree vertices.



多次DFS

Note:

- The path should be maintained as a linked list.
- For each adjacency list, maintain a pointer to the last edge scanned.
- $T = O(|E| + |V|)$

Find a simple cycle in an undirected graph that visits every vertex – *Hamilton cycle*