Graph Traversal

Contents

- **Flavors of Graphs**
- **Data Structures for Graph**
- **Breadth-First Search and Applications**
  - **Connected Components**
  - **Two-Coloring Graphs**
- **Depth-First Search and Applications**
  - Finding Cycles
  - Articulation Vertices
  - Topological Sorting
  - Strongly Connected Components

FLAVORS OF GRAPHS

Graphs

Definition. A ***directed graph (digraph)*** $G = (V, E)$ is an ordered pair consisting of

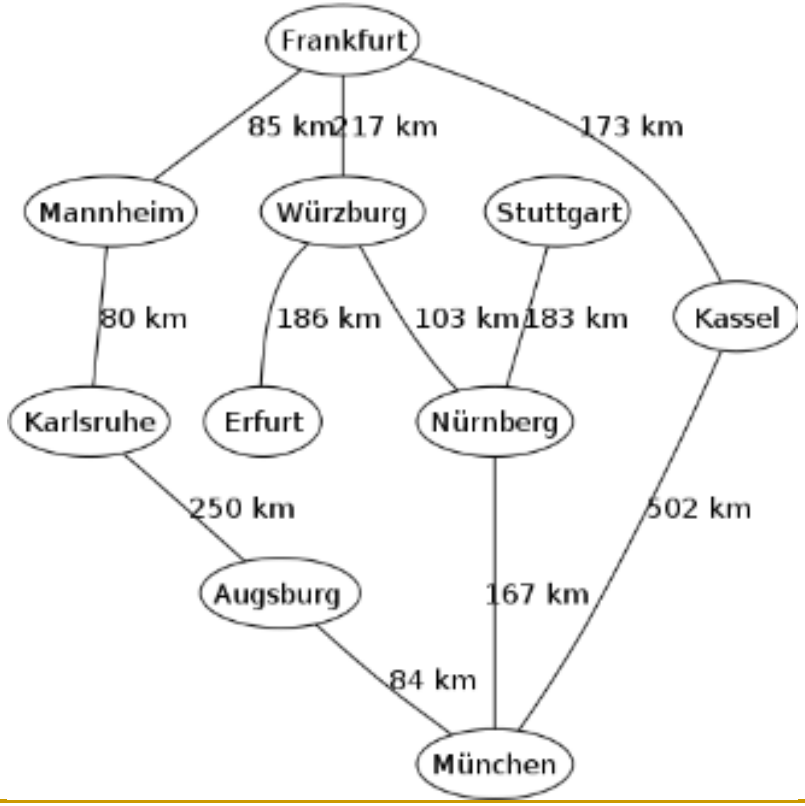
- a set V of ***vertices*** (singular: ***vertex***),
- a set $E \subseteq V \times V$ of ***edges***.

In an ***undirected graph*** $G = (V, E)$, the edge set E consists of *unordered* pairs of vertices.

In either case, we have $|E| = O(V^2)$. Moreover, if G is connected, then $|E| \geq |V| - 1$, which implies that $\lg |E| = \Theta(\lg V)$.

Real Life Examples

- An example map of Germany with some connections between cities.

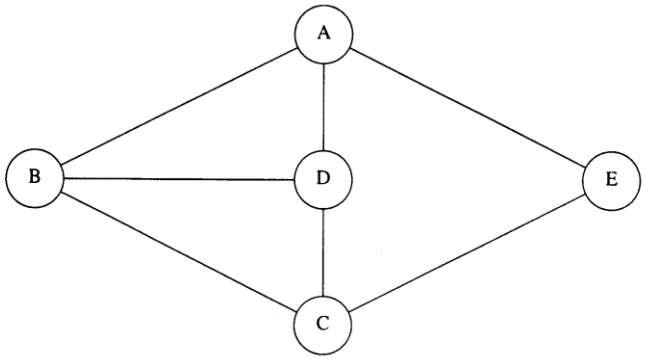


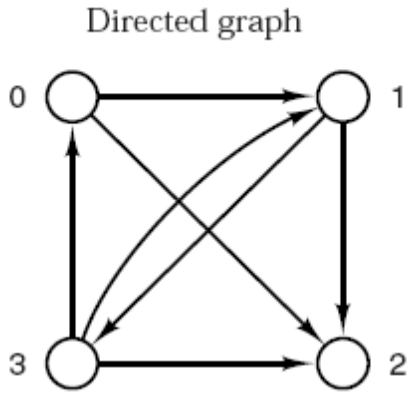
Flavor of Graphs

- The first step in any graph problem is determining which flavor of graph you are dealing with.
- The flavor of graph has a big impact on which algorithms are appropriate and efficient.

Directed vs. Undirected Graphs

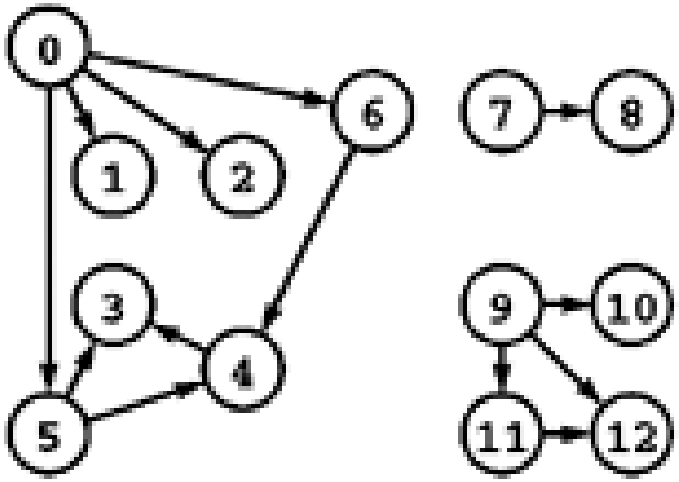
- A graph G = (V; E) is *undirected* if edge (x,y) $\in$ E implies that (y,x) is also in E, else it is *directed* and called *Digraph*.
- Road networks *between* cities are typically undirected.
- Street networks *within* cities are almost always directed because of one-way streets.





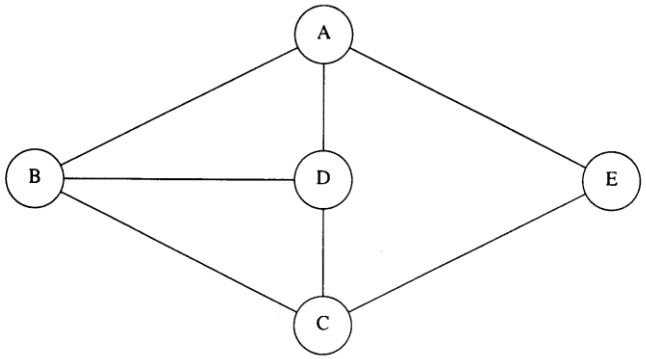
有向图

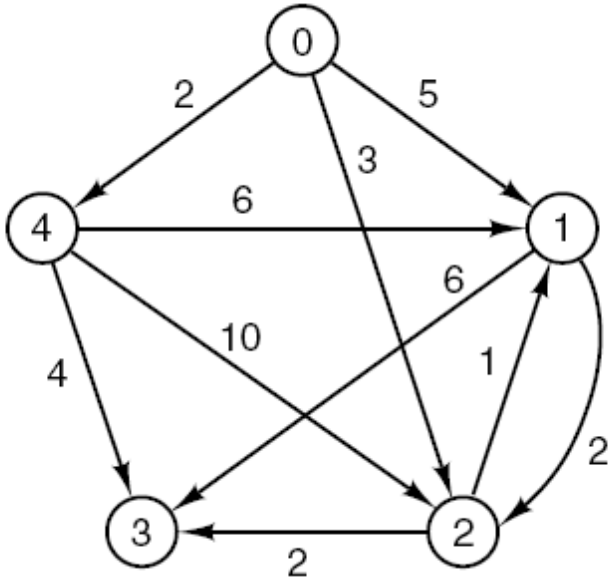
- 边都是单向(unidirectional)的, 因此边(u,v)是有序数对. 有时用弧(arc)专指有向边
- 在有向边(u, v)中, u和v分别叫
  - 源(source)和目的(destination)
  - 尾(tail)和头(head), 不过和数据结构有冲突
- 有向无环图(directed acyclic graph, DAG)不是树, 它的基图(underlying undirected graph)也不一定是树



Weighted vs. Unweighted Graphs

- In *weighted* graphs, each edge (or vertex) of G is assigned a numerical value, called *weight* or *cost*.
- The edges of a road network graph might be weighted with their length, drive-time or speed limit.



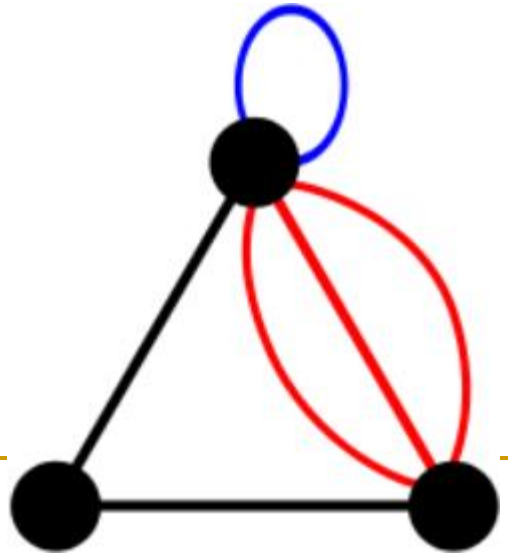


带权图

- 可以给边加权(weight), 成为带权图, 或加权图(weighted graph). 权通常代表费用、距离等, 可以是正数, 也可以是负数
- 也可以给点加权, 或者边上加多种权
- 带权有向图一般也称为网络(network)
- 带权图的问题多为组合优化问题, 在运筹学中有广泛应用

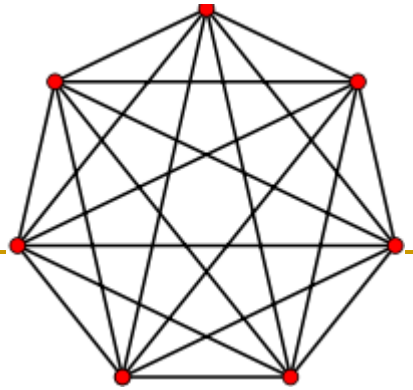
Simple vs. Non-simple Graphs

- Certain types of edges complicate the task of working with graphs. A *self-loop* is an edge (x, x) involving only one vertex.
- An edge (x, y) is a *multi-edge* if it occurs more than once in the graph.
- Any graph which avoids these structures is called *simple*.



Sparse vs. Dense Graphs

- Graphs are *sparse* when only a small fraction of the possible number of vertex pairs actually have edges defined between them.
- Graphs are usually *sparse* due to application-specific constraints. Road networks must be sparse because of road junctions.
- Typically *dense* graphs have a quadratic number of edges while *sparse* graphs are linear in size.

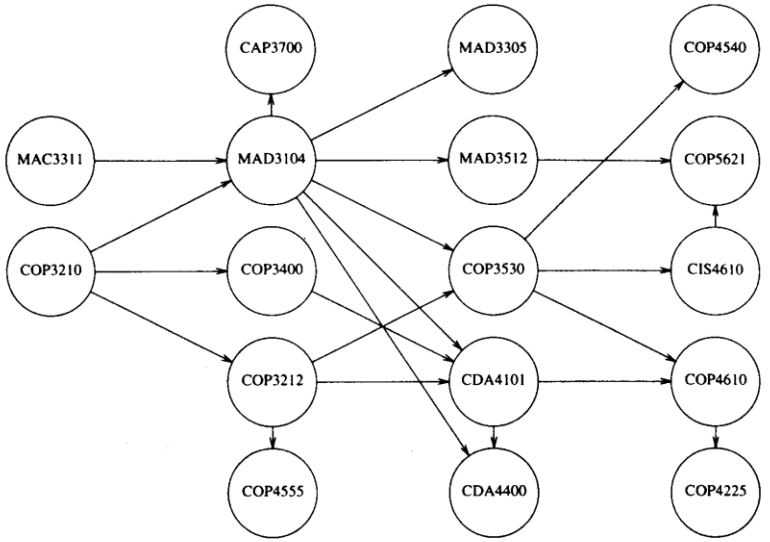


稀疏图和稠密图

- 边和V(V-1)/2相比非常少的称为稀疏图(sparse graph), 它的补图为稠密图(dense graph)
- 时间复杂度为ElogE的算法和V^2的算法对于稀疏图来说前者好, 稠密图来说后者好
- 一般来说, 即使对于稀疏图, V和E相比都很小, 可以用E来代替V+E, 因此O(V(V+E))可以简写为O(VE)

Cyclic vs. Acyclic Graphs

- An *acyclic* graph does not contain any cycles. *Trees* are connected acyclic *undirected* graphs.
- Directed acyclic graphs are called *DAGs*. They arise naturally in scheduling problems, where a directed edge (x, y) indicates that x must occur before y.

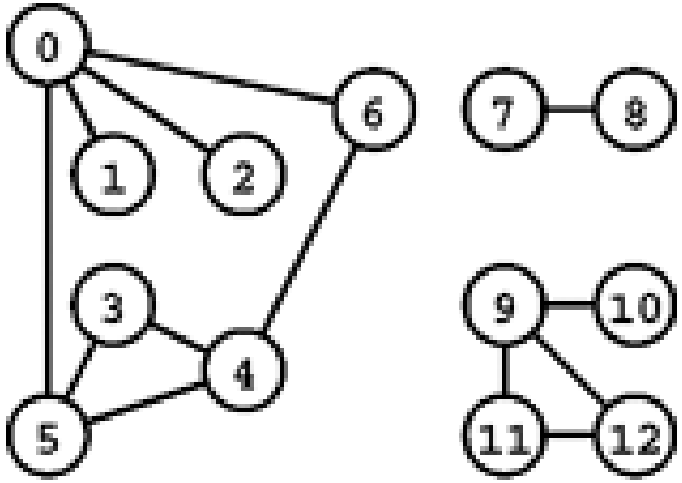


Implicit vs. Explicit Graphs

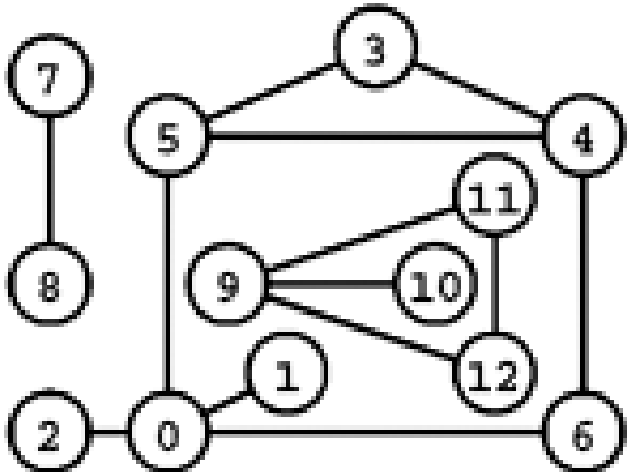
- Many graphs are not *explicitly* constructed and then traversed, but built as we use them.
- A good example arises in backtrack search.

Isomorphism Graphs

- *isomorphism testing*: determining whether the topological structure of two graphs are in fact identical if we ignore any labels.
- 右图是同一个图的不同表示。所有边如下表



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



The Friendship Graph

- Consider a graph where the vertices are people, and there is an edge between two people if and only if they are friends.
- This graph is well-defined on any set of people: SYSU, Guang Zhou, or the world.
- What questions might we ask about the friendship graph?

If I am your friend, does that mean you are my friend?

- A graph is *undirected* if (x, y) implies (y, x). Otherwise the graph is directed.
- The “heard-of” graph is directed since countless famous people have never heard of me!
- The “be-married-with” graph is presumably undirected, since it requires a partner.

Am I my own friend?

- An edge of the form (x; x) is said to be a *loop*.
- If x is y's friend several times over, that could be modeled using *multiedges*, multiple edges between the same pair of vertices.
- A graph is said to be *simple* if it contains no loops and multiple edges.

Am I linked by some chain of friends to the President?

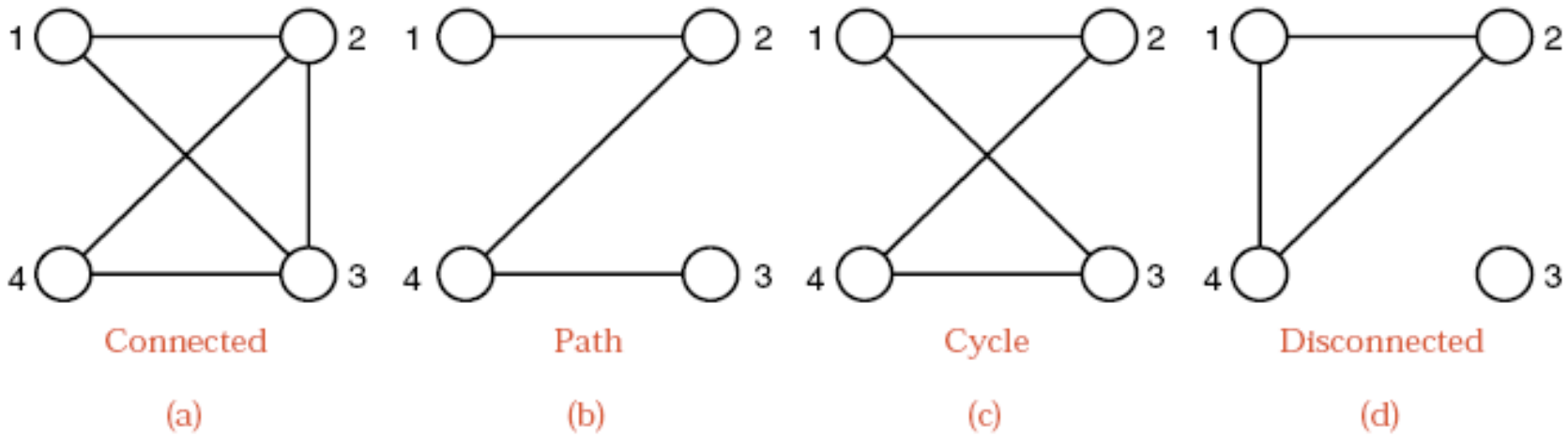
- A *path* is a sequence of edges connecting two vertices.
- Since *the President of Sun Yat-sen University* is my friend's director, there is a path between me and him.

路径和圈

- 一条路径(path)是一个结点序列, 路上的相邻结点在图上是邻接的.
- 如果结点和边都不重复出现, 则称为简单路径(simple path). 如果除了起点和终点相同外没有重复顶点和边, 称为圈(cycle).
- 不相交路(disjoint path)表示没有除了起点和终点没有公共点的路. 更严格地
  - 任意点都不相同的叫严格不相交路(vertex-disjoint path)
  - 同理定义边不相交(edge-disjoint path)路
- 注意: 汉语中圈和环经常混用(包括一些固定术语). 由于一般不讨论自环(self-loop), 所以以后假设二者等价而不会引起混淆

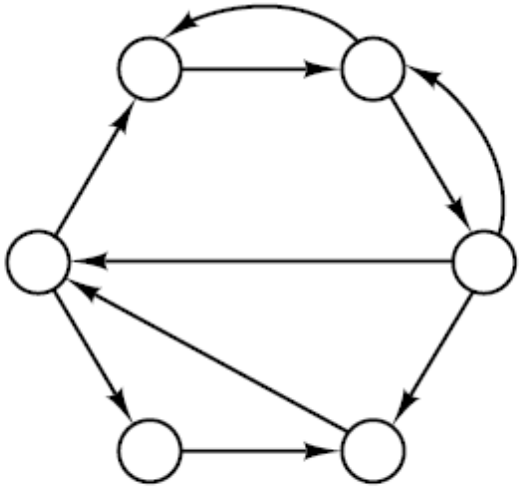
How close is my link to the President?

- we are often interested in the *shortest path* between two nodes.
- In graph (a), path 1-2-4 is shorter than 1-2-3-4.

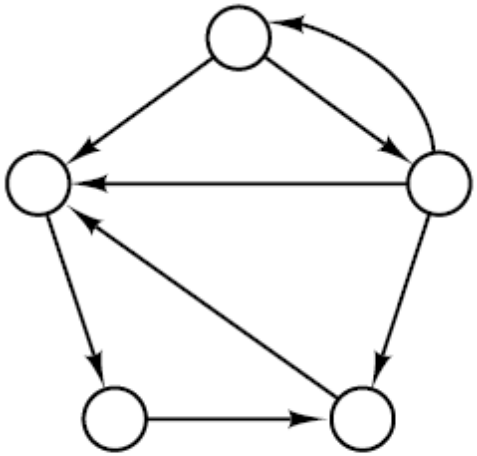


Is there a path of friends between any two people?

- A graph is *connected* if there is a path between any two vertices.
- A directed graph is *strongly connected* if there is a directed path between any two vertices.



Strongly connected



Weakly connected

连通性

- 如果任意两点都有路径, 则称图是连通(connected)的, 否则称图是非连通的.
- 非连通图有多个连通分量(connected component, cc), 每个连通分量是一个极大连通子图(maximal connected subgraph), 因为任意加一个结点以后将成为非连通图

Who has the most friends?

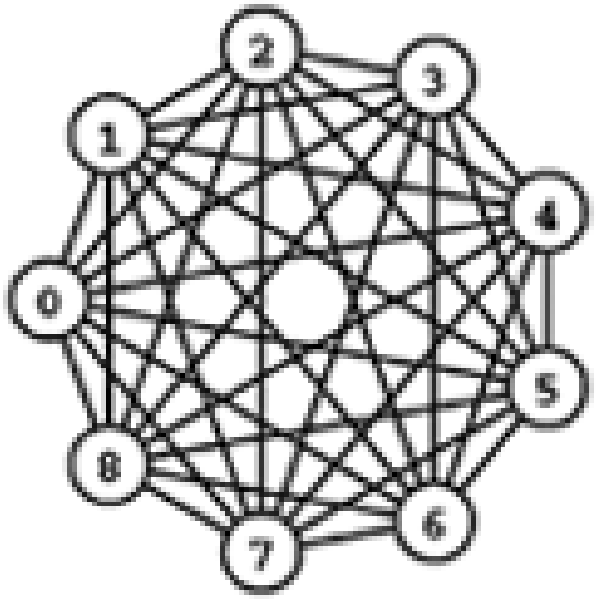
- The *degree* of a vertex is the number of edges adjacent to it.

生成树

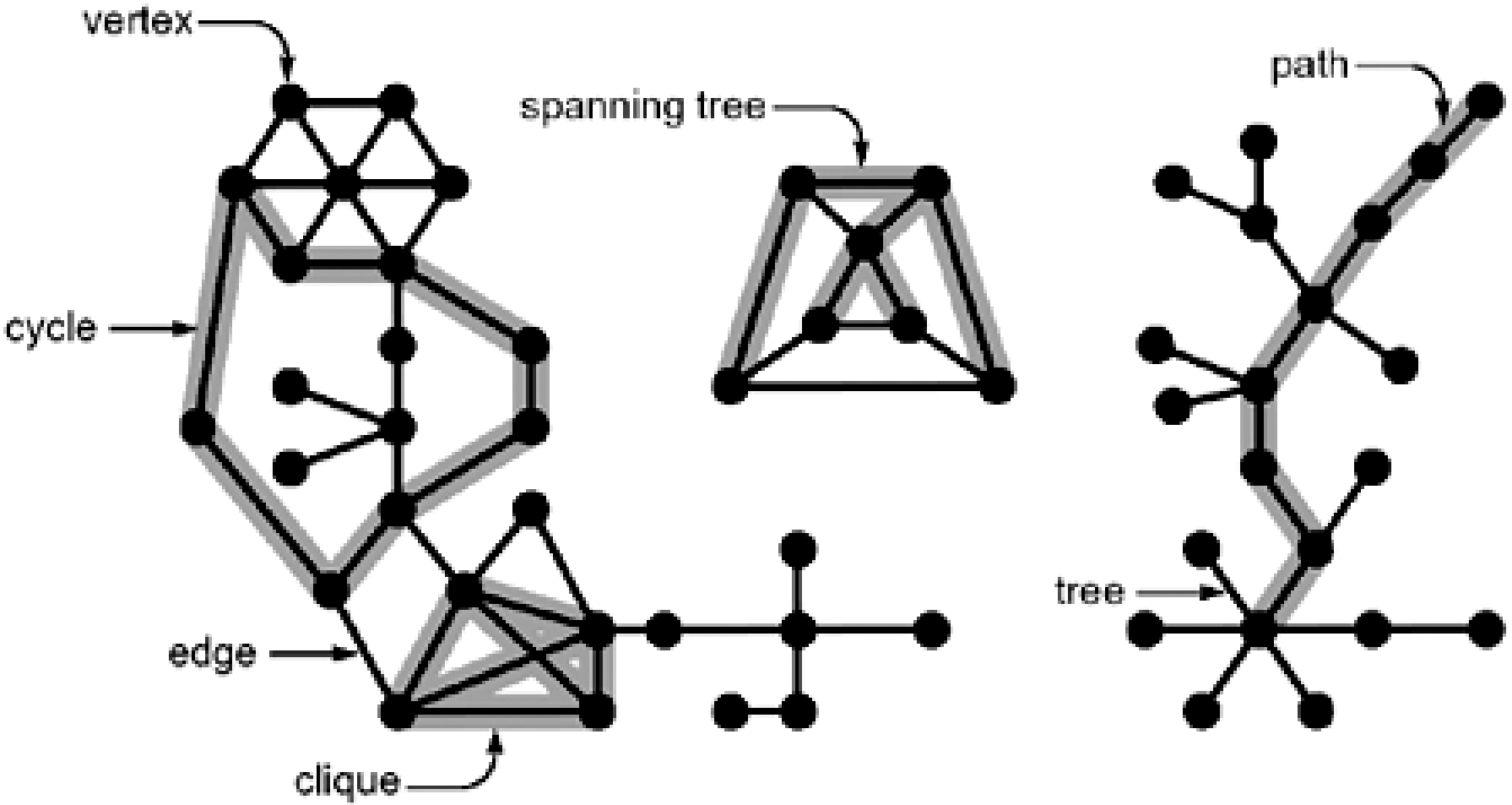
- 连通无圈图称为树(tree)
- 树的集合称为森林(forest)
- 生成树: 包含某图G所有点的树
- 生成森林: 包含某图G所有点的森林
- 一个图G是树当且仅当以下任意一个条件成立
  - G有V-1条边, 无圈
  - G有V-1条边, 连通
  - 任意两点只有唯一的简单路径
  - G连通, 但任意删除一条边后不连通
- 还有其他条件,可类似定义

完全图和补图

- 如果V个点的图有V(V-1)/2图, 称为完全图
- 对于(u,v), 若邻接则改为非邻接, 若非邻接则改为邻接, 得到的图为原图的补图
- 原图和补图(complement graph)的并(union)为完全图
- 完全子图称为团(clique)

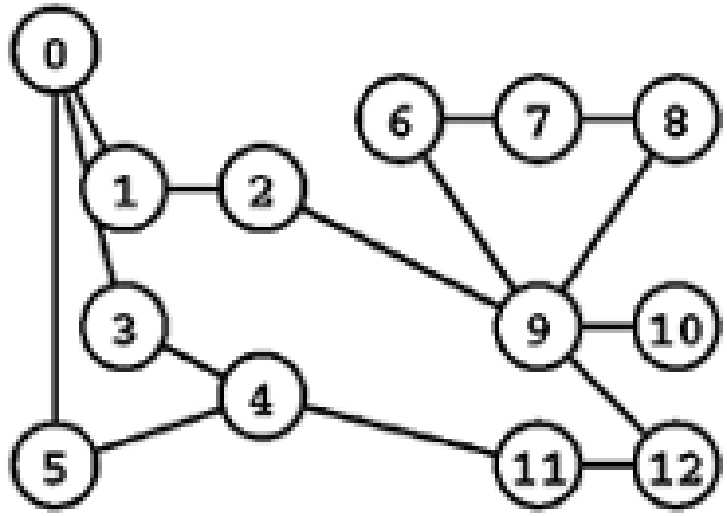


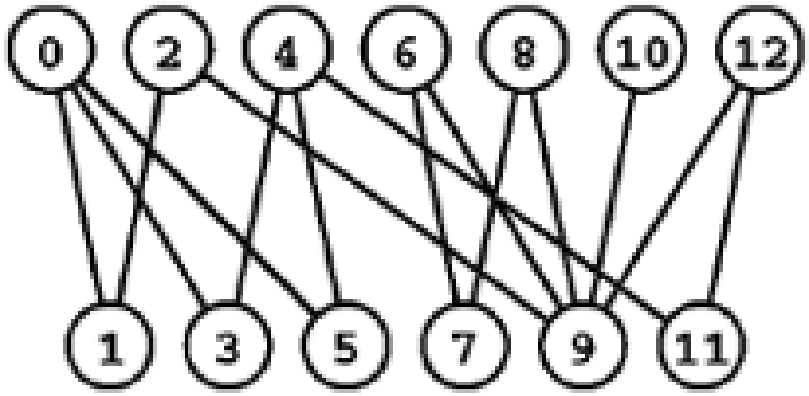
术语示意



二分图

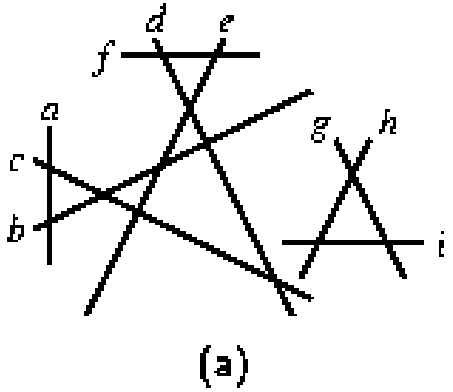
- 可以把结点分成两部分, 每部分之间没有边. 这样的图只有奇圈(odd-cycle), 即包含奇数条边的圈
- 许多困难问题在二分图上有有效算法



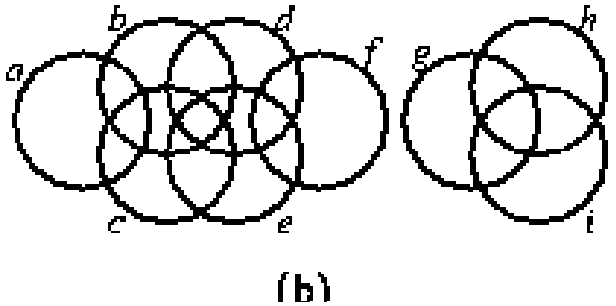


相交图、区间图

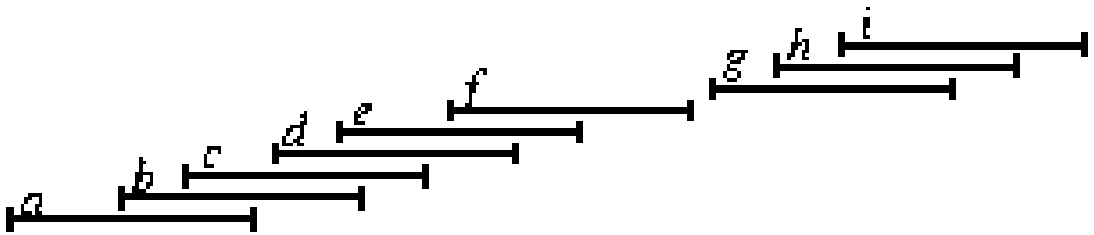
- 交图: 把物体看作顶点, 相交关系看为边
- 特殊情况: 区间图(interval graph). 很多困难问题在区间图上有有效算法



(a)



(b)



(c)

DATA STRUCTURES FOR GRAPH

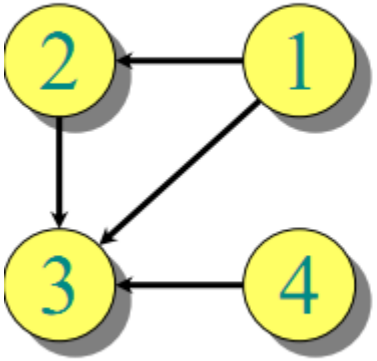
Data Structures for Graphs

- There are two main data structures used to represent graphs.
  - Adjacency Matrix (邻接矩阵)
  - Adjacency Lists (邻接表)
  - Forward Star (向前星)
- We assume the graph G = (V, E) contains n = | V | vertices and m = | E | edges.

Adjacency Matrix Representation

The ***adjacency matrix*** of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is the matrix $A[1 \ldots n, 1 \ldots n]$ given by

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in \text{E}, \\ 0 & \text{if } (i, j) \notin \text{E}. \end{cases}$$



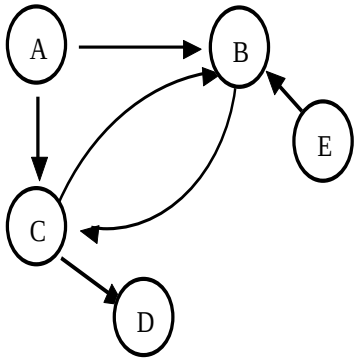
A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

$\Theta(V^2)$ storage

$\Rightarrow$ ***dense*** representation.

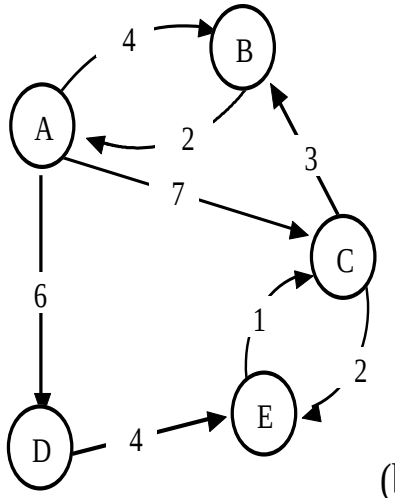
Adjacency Matrix Representation

- Adjacency matrix representation for weighted graph



	A	B	C	D	E
A	-1	1	1	-1	-1
B	-1	-1	1	-1	-1
C	-1	1	-1	1	-1
D	-1	-1	-1	-1	-1
E	-1	1	-1	-1	-1

(a)



	A	B	C	D	E
A	-1	4	7	6	-1
B	2	-1	-1	-1	-1
C	-1	3	-1	-1	2
D	-1	-1	-1	-1	4
E	-1	-1	1	-1	-1

(b)

Adjacency Matrix Representation

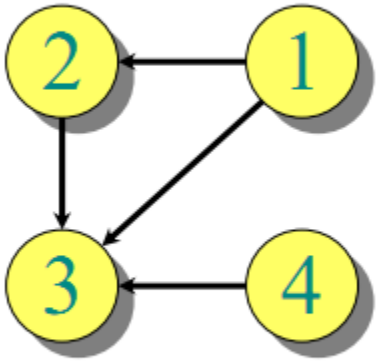
```
int n; //num of vertex
int m; //num of edges
int u, v; // vertex of edge (u,v)
cin >> n >> m;
int g[n+1][n+1]; //store the graph
memset(g,0,sizeof(g));
for(int j=1; j<=m; j++)
{
     cin >> u >> v;
     g[u][v] = 1;
     if (directed==false) g[v][u]=1;
}
```

相关问题

- 如何处理重边和自环?
- 如何用位存储节省空间?
- 如果用上三角法储存无向图?
- 如何用边测试函数而非完整的邻接矩阵储存隐式图?
- 如何修改A的元素类型以储存边的附加信息?

Adjacency Lists Representation

An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



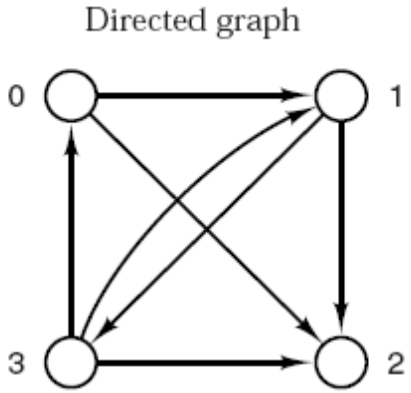
$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

For undirected graphs, $|Adj[v]| = degree(v)$.
For digraphs, $|Adj[v]| = out\text{-}degree(v)$.

Handshaking Lemma: $\sum_{v \in V} = 2|\mathrm{E}|$ for undirected graphs $\Rightarrow$ adjacency lists use $\Theta(V + E)$ storage — a ***sparse*** representation (for either type of graph).

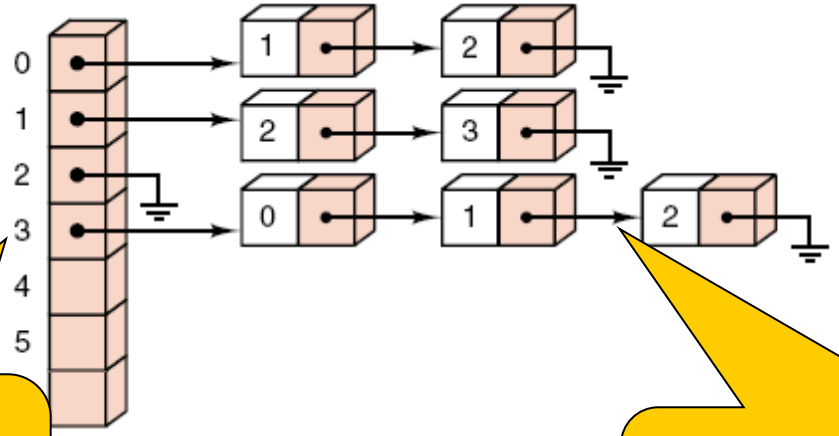
Adjacency Lists Representation

- Directed graph



Adjacency sets

vertex	Set
0	{ 1, 2 }
1	{ 2, 3 }
2	Ø
3	{ 0, 1, 2 }

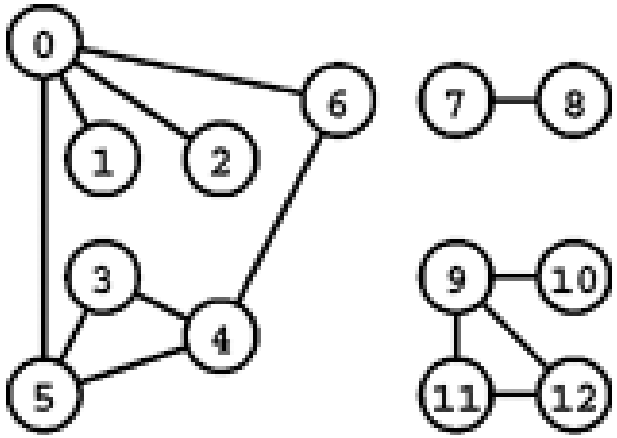


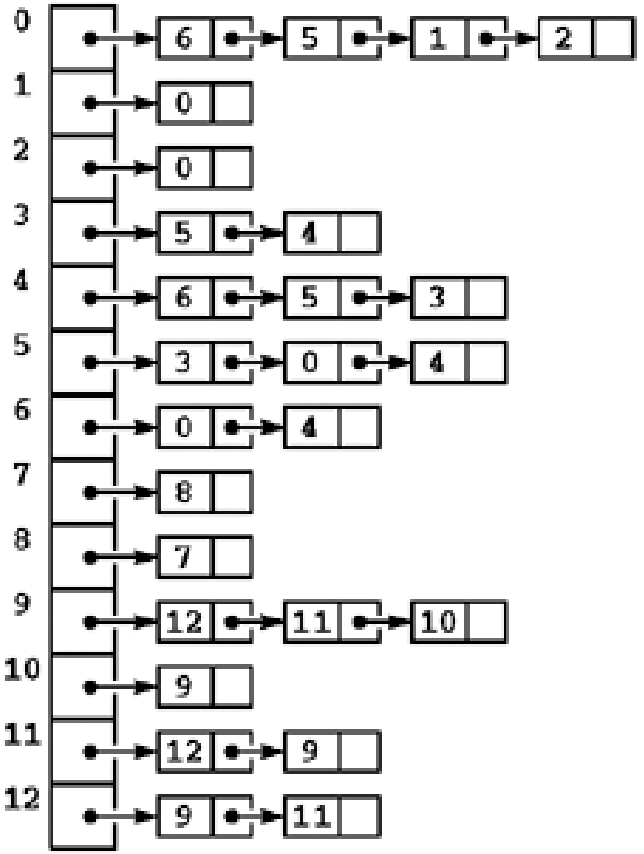
The set of vertices is a contiguous list

The set of adjacent vertices is a linked list

Adjacency Lists Representation

- Undirected graph





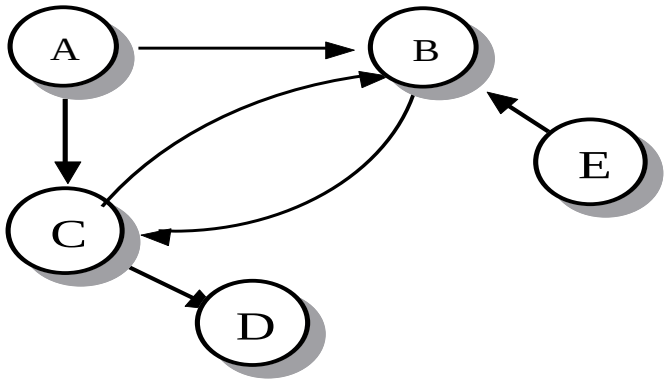
Adjacency Lists Representation

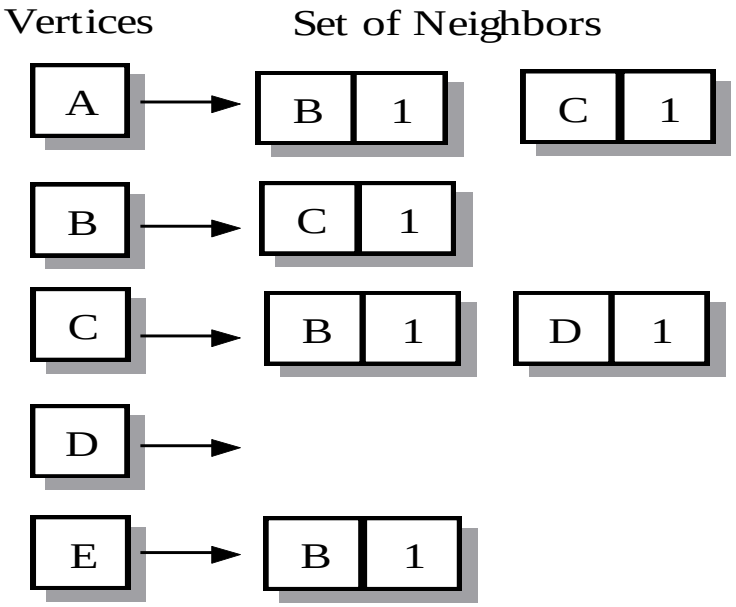
```
int n; //num of vertex
int m; //num of edges
int u, v; // vertex of edge (u,v)
cin >> n >> m;
vector<int> g[n+1]; //store the graph
for(int j=1; j<=m; j++)
{
      cin >> u >> v;
      g[u].push_back(v);
      if (directed==false) g[v].push_back(u);
}
```

相关问题

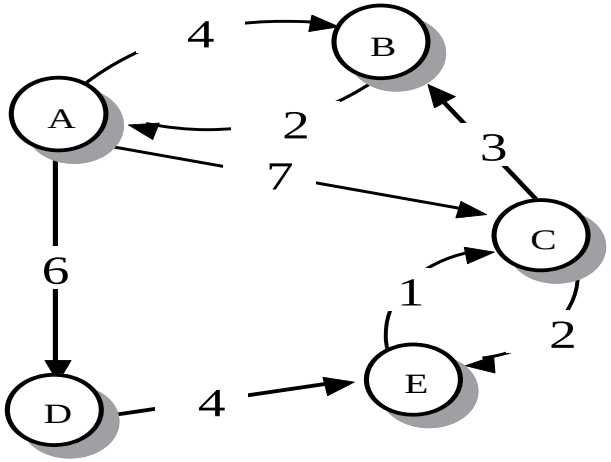
- 邻居排序方式可能影响结果
- 查找/删除边不是常数时间
- 邻接表的空间O(V+E), 对于稀疏图优于邻接矩阵
- 可以用编号代替指针, 加快速度并节省空间
- For weighted graph, store neighboring vertices and their edge costs
- If necessary, the two copies of each edge can be linked by a pointer to facilitate deletions.

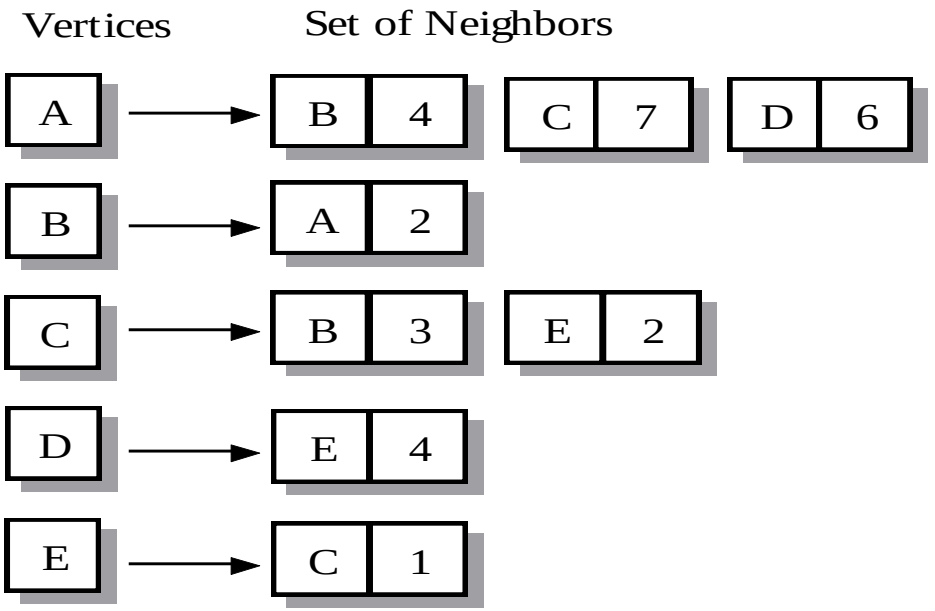
(a)


A
B
C
D
E


Vertices
Set of Neighbors
A
B 1
C 1
B
C 1
C
B 1
D 1
D
E
B 1

(b)


A
B
C
D
E
4
2
3
7
6
1
2
4


Vertices
Set of Neighbors
A
B 4
C 7
D 6
B
A 2
C
B 3
E 2
D
E 4
E
C 1

Tradeoffs Between Adjacency Lists and Adjacency Matrices

Comaprison	Winner
Faster to test if (x,y) exists?	matrices
Faster to find vertex degree?	lists
Faster to find vertex degree?	lists
Less memory on small graphs?	lists $O(m+n)$ vs. $O(n^2)$
Less memory on big graphs?	matrices (small win)
Edge insertion or deletion?	matrices $O(1)$ vs $O(d)$
Faster to traverse the graph?	lists $O(m+n)$ vs. $O(n^2)$
Better for most problems?	lists

Question – Adjacency Matrix or List?

Would you use the adjacency list structure or the adjacency matrix structure in each of the following cases? Justify your choice.

a) The graph has 10,000 vertices and 20,000 edges.
It is important to use as little space as possible.

Answer: Use adjacency list as there are on average 2 edges on each vertex.
We will waste a lot of memory space if adjacency matrix is used.

b) The graph has 10,000 vertices and 20,000,000 edges.
It is important to use as little space as possible.

Answer: Use adjacency matrix as there are on average 2000 edges on each vertex.
It would be much faster to find the neighbors of each vertex.

c) You need to answer the query areAdjacent as fast as possible.
No matter how much space you use.

Answer: Use adjacency Matrix since space is not a problem.
More importantly, the query areAdjacent can be answered in O(1) time.

前向星表示

- 把所有边(u, v)按u的主关键字, v为次关键字排序, 并记录每个结点u的邻居列表的开始位置start[u](则start[u+1]是结束位置)
- 紧凑存储, 不需要使用指针, 但边的插入和删除操作可能引起大幅度变化.
- 一般用于静态图. 可以方便的遍历一个点的所有邻居并通过可以储存”当前弧”提高某些图算法的效率

Think About

- Present correct and efficient algorithms to convert between the following graph data structures, for an undirected graph G with n vertices and m edges. You must give the time complexity of each algorithm.
  - 1. Convert from an adjacency matrix to adjacency lists.
  - 2. Convert from an adjacency matrix to adjacency lists.

GRAPH TRAVERSAL BASIC

Traversing a Graph

- One of the most fundamental graph problems is to traverse every edge and vertex in a graph.
- *efficiency:* visit each edge at most twice.
- *correctness:* do the traversal in a systematic way so that we don't miss anything.

Marking Vertices

- 必须在访问一个结点后再给它加上一个标记，使得它不会被重复访问。BFS和DFS的区别在于访问结点的顺序。
- 任意结点总处于以下状态中的一种:
  - *undiscovered* – 所有结点的初始状态.
  - *discovered* – 结点第一次被访问，但它的出边未处理完毕。
  - *processed* – 结点的所有出边都已经处理完毕。

Marking Vertices

- Obviously, a vertex cannot be *processed* before we discover it, so over the course of the traversal the state of each vertex progresses from *undiscovered* to *discovered* to *processed*.

To Do List

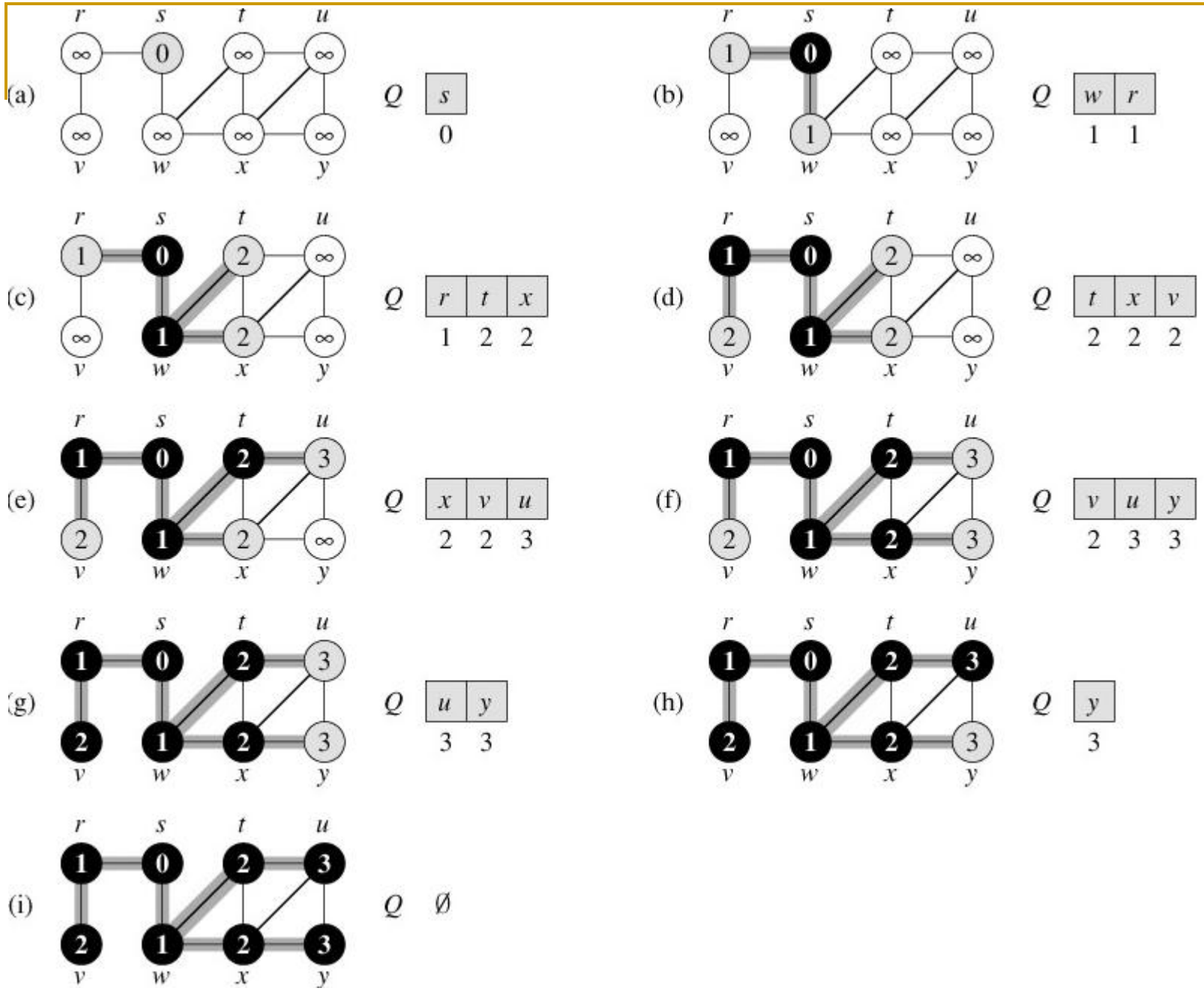
- We must also maintain a structure containing all the vertices we have discovered but not yet *processed*.
- Initially, only a single start vertex is considered to be *discovered*.
- To completely explore a vertex, we look at each edge going out of it. For each edge which goes to an *undiscovered* vertex, we mark it *discovered* and add it to the list of work to do.
- Note that regardless of what order we fetch the next vertex to explore, each edge is considered exactly twice, when each of its endpoints are explored.

BREADTH-FIRST SEARCH

BFS基本算法

- 由Moore和Lee独立提出
- 给定图G和一个源点s, 宽度优先遍历按照从近到远的顺序考虑各条边. 算法求出从s到各点的距离
- 宽度优先的过程对结点着色.
  - 白色(undiscovered): 没有考虑过的点
  - 黑色(processed): 已经完全考虑过的点
  - 灰色(discovered): 发现过, 但没有处理过, 是遍历*边界*
- 依次处理每个灰色结点u, 对于邻接边(u, v), 把v着成灰色并加入树中, 在树中u是v的父亲(parent)或称前驱(predecessor). 距离d[v] = d[u] + 1
- 整棵BFS树的根为s

```
BFS(G, s)
 1  for each vertex u ∈ V [G] - {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          p[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  p[s] ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10   while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12          for each v ∈ Adj[u]
13              do if color[v] = WHITE
14                    then color[v] ← GRAY
15                         d[v] ← d[u] + 1
16                         p[v] ← u
17                         ENQUEUE(Q, v)
18          color[u] ← BLACK
```


(a)
r s t u
∞ 0 ∞ ∞
∞ ∞ ∞ ∞
v w x y
Q s
0
(b)
r s t u
1 0 ∞ ∞
∞ 1 ∞ ∞
v w x y
Q w r
1 1
(c)
r s t u
1 0 2 ∞
∞ 1 2 ∞
v w x y
Q r t x
1 2 2
(d)
r s t u
1 0 2 ∞
2 1 2 ∞
v w x y
Q t x v
2 2 2
(e)
r s t u
1 0 2 3
2 1 2 ∞
v w x y
Q x v u
2 2 3
(f)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q v u y
2 3 3
(g)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q u y
3 3
(h)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q y
3
(i)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q Ø

```
void bfs(int s)
{
  queue<int> c;
  color[s] = 1;  dist[s] = 0; parent[s] = -1;
  c.push(s);
  while(!c.empty()){
    u = c.front(); c.pop();
    /* do sth when vertex discovered */
    printf("%d ",u);
    for(int i=0;i<graph[u].size();i++) {
      v = graph[u][i];
      if (color[v]==0) {
        color[v]=1; parent[v] = u;
        dist[v] = dist[u] + 1;
        c.push(v);
        //do sth to the edge (u,v)
      }
    }
    color[u] = 2;
  }
}
```

```
void bfs()
{
  memset(color,0,sizeof(color));
  memset(dist,-1,sizeof(dist));
  memset(parent,-1,sizeof(parent));
  for(int i=1; i<=n; i++)
  {
    if (color[i]==0) //vertex i is undiscovered
      bfs(i);
  }
}

int main()
{
  read(false);
  printf("vertex discoverd: "); bfs();  printf("\n");
  printf("path from 1 to 4: ");
  find_path(1,4,parent); printf("\n");
}
```

Finding Paths

- The parent array set within bfs() is very useful for finding interesting paths through a graph.
- The vertex which discovered vertex *i* is defined as parent[*i*].
- The parent relation defines a tree of discovery with the initial search node as the root of the tree.

Recursion and Path Finding

```
find_path(int start, int end, int parents[])
{
   if ((start == end) && (end == -1))
       printf("%d ",start);
   else {
       find_path(start,parents[end],parents);
       printf("%d ",end);
   }
}
```

vertex	1	2	3	4	5	6
parent	-1	1	2	5	1	1

Shortest Paths in Unweighted Graph and BFS

- In BFS, vertices are discovered in order of increasing distance from the root, so this tree has a very important property.
- The unique tree path from the root to any node $x \in V$ uses the smallest number of edges (or equivalently, intermediate nodes) possible on any root-to-x path in the graph.

用BFS求最短路

- 定理: 对于无权图(每条边的长度为1), BFS算法计算出的d[i]是从s到i的最短路
- 满足d[i]=1的点一定是正确的(因为长度至少为1), 并且其他点都满足d[i]>1. 容易证明对于任意距离值x, d[i]=x的点一定是正确的, 而且白色点(没有计算出距离的点)的距离一定至少为x+1
- 更进一步, 根据每个点的parent值, 可以计算出它到s的一条最短路

Connected Components

- The *connected components* of an undirected graph are the separate “pieces” of the graph such that there is no connection between the pieces.
- Anything we discover during a BFS must be part of the same connected component. We then repeat the search from any undiscovered vertex (if one exists) to define the next component, until all vertices have been found:

```
void connected_components()
{
  int cc = 0; //cc: num of connected components
  memset(color,0,sizeof(color));
  memset(dist,-1,sizeof(dist));
  memset(parent,-1,sizeof(parent));

  for(int i=1; i<=n; i++)
  {
    if (color[i]==0) //vertex i is undiscovered
    {
      ++cc;
      printf("Component %d:", cc);
      bfs(i);
      printf("\n");
    }
  }
}
```

Two-Coloring Graphs

- The *vertex coloring* problem seeks to assign a label (or color) to each vertex of a graph such that no edge links any two vertices of the same color.
- A graph is *bipartite* if it can be colored without conflicts while using only two colors. Bipartite graphs are important because they arise naturally in many applications.

```
bool bicoloring(int s)
{
  queue<int> c;
  int paint[MAXN]; /*0未染色，1红色，-1黑色*/
  memset(color,0,sizeof(color));
  memset(paint,0,sizeof(paint));
  color[s] = 1;  c.push(s);
  while(!c.empty())
  {
    u = c.front(); c.pop();
    for(int i=0;i<graph[u].size();i++) {
      v = graph[u][i];
      if (color[v]==1) {
        if (paint[v] == -paint[u]) continue;
        else return false;
      }
      color[v] = 1;
      paint[v] = -paint[u];
      c.push(v);
    }
  }
  return true;
}
```

BFS相关思考题

- 给出判断图是否为二分图的线性时间算法
- 一棵树T的直径定义为结点两两间距离的最大值. 给出求树直径的线性时间算法
- 对无向图G, 给出一个路径, 经过每条边恰好两次(一个方向一次). 如何利用这条路径来走迷宫?

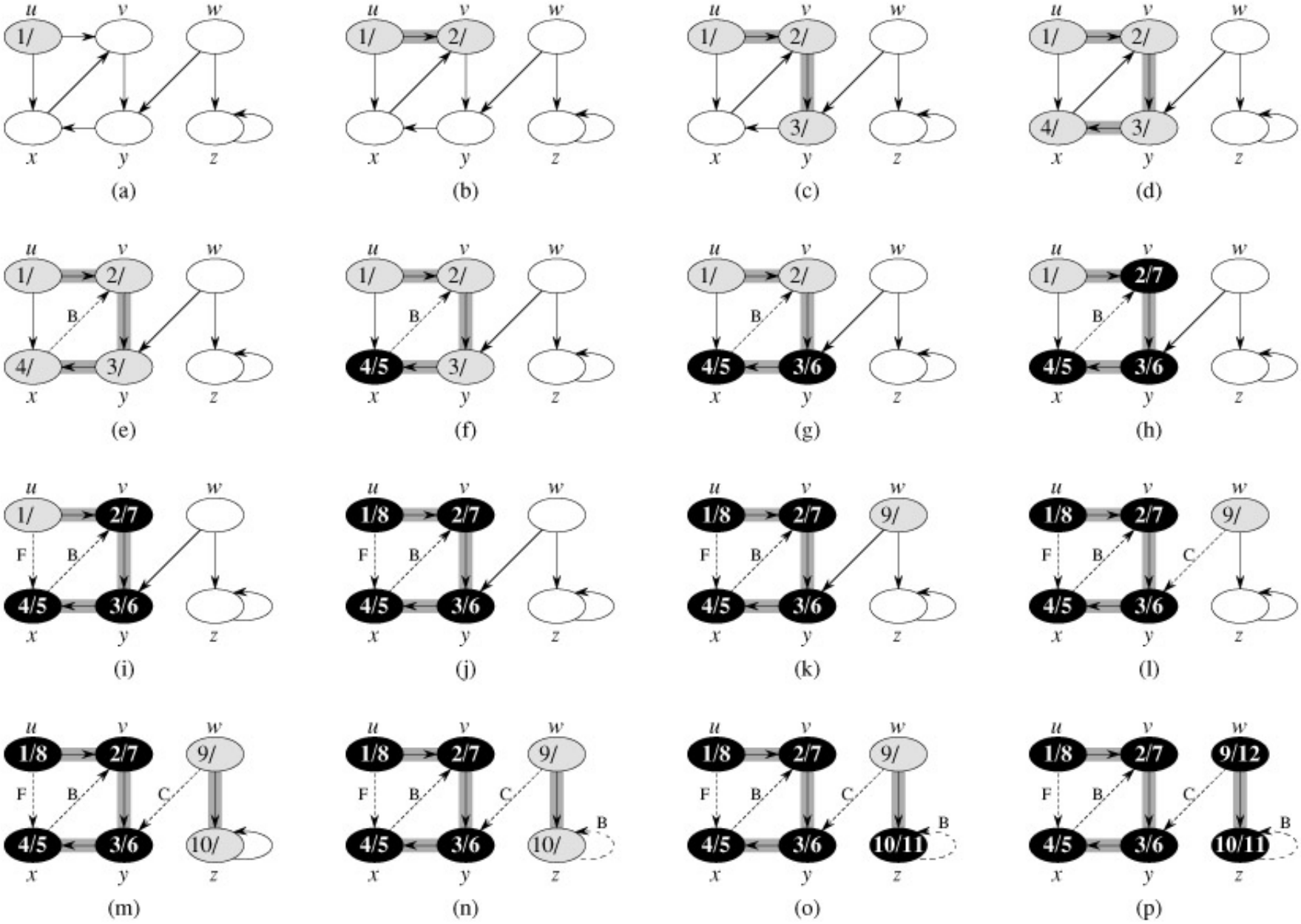
DEPTH-FIRST SEARCH

基本算法

- 新发现的结点先扩展。（可以把BFS程序中的队列直接改成栈）
- 得到的可能不是一棵树而是森林, 即深度优先森林(Depth-first forest)
- 特别之处: 引入时间戳(timestamp)
  - 发现时间pre[v]: 变灰的时间
  - 结束时间post[v]: 变黑的时间
  - 1<=pre[v] < post[v] <= 2|V|
- 初始化: time为0, 所有点为白色, dfs森林为空
- 对每个白色点u执行一次DFS-VISIT(u)

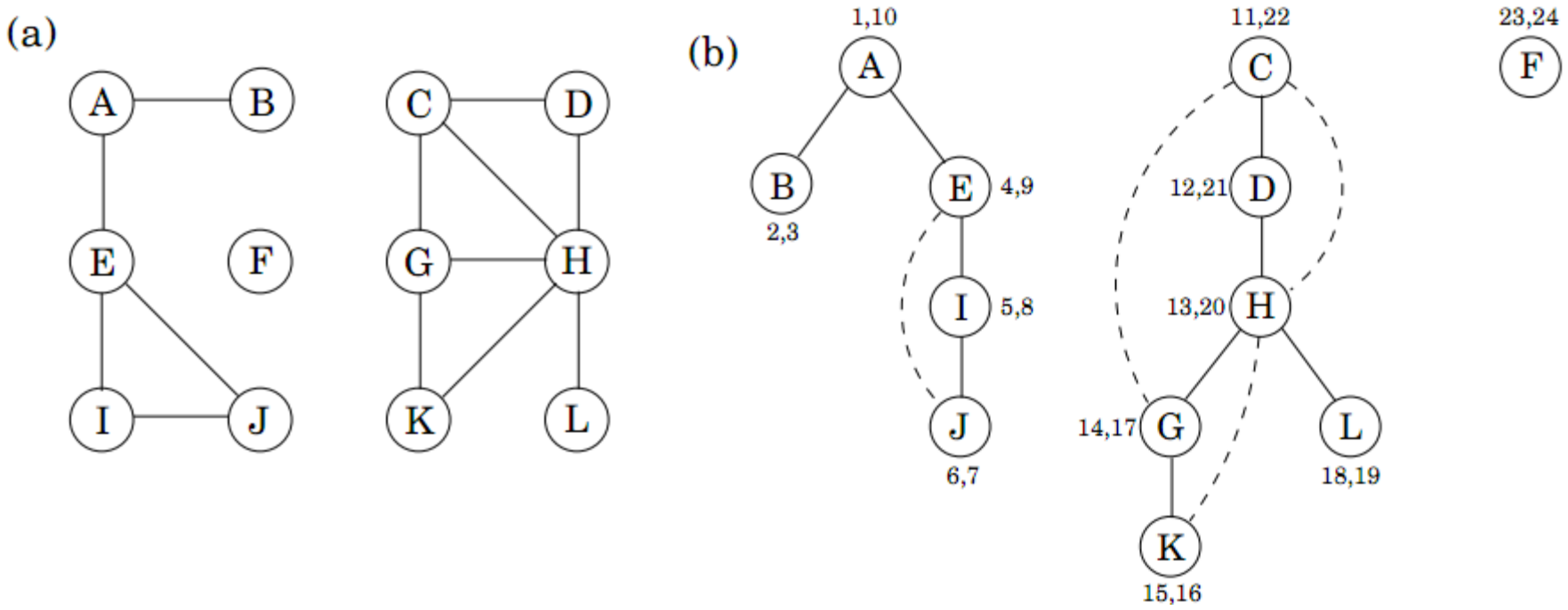
```
DFS(G)
1  for each vertex u ∈ V [G]
2       do color[u] ← WHITE
3          parent[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V [G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)

DFS-VISIT(u)
1  color[u] ← GRAY     //White vertex u has just been discovered.
2  pre[u] = ++time
4  for each v ∈ Adj[u]  //Explore edge(u, v).
5       do if color[v] = WHITE
6             then parent[v] ← u
7                     DFS-VISIT(v)
8  color[u] = BLACK     // Blacken u, it is finished.
9  post [u] = ++time
```


u v w
1/
x y z
(a)
1/ 2/
(b)
1/ 2/ 3/
(c)
1/ 2/ 4/ 3/
(d)
B
(e)
4/5
(f)
3/6
(g)
2/7
(h)
F
(i)
1/8
(j)
9/
(k)
C
(l)
10/
(m)
(n)
10/11
(o)
9/12
(p)

Depth-First Search

(a) A 12-node graph. (b) DFS search forest.



DFS树的性质

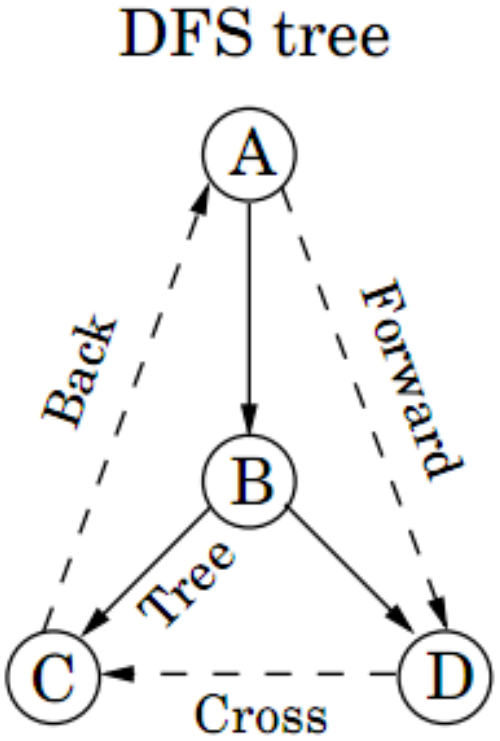
- 括号结构性质
- 对于任意结点对(u, v), 考虑区间[pre[u],post[u]]和[pre[v], post[v]], 以下三个性质恰有一个成立:
  - 完全分离
  - u的区间完全包含在v的区间内, 则在dfs树上u是v的后代
  - v的区间完全包含在u的区间内, 则在dfs树上v是u的后代
- 三个条件非常直观

DFS树的性质

- 定理1(嵌套区间定理): 在DFS森林中v是u的后代当且仅当pre[u]<pre[v]<post[v]<post[u], 即区间包含关系. 由区间性质立即得到.
- 定理2(白色路径定理): 在DFS森林中v是u的后代当且仅当在pre[u]时刻(u刚刚被发现), v可以由u出发*只经过白色结点*到达. 证明: 由嵌套区间定理可以证明

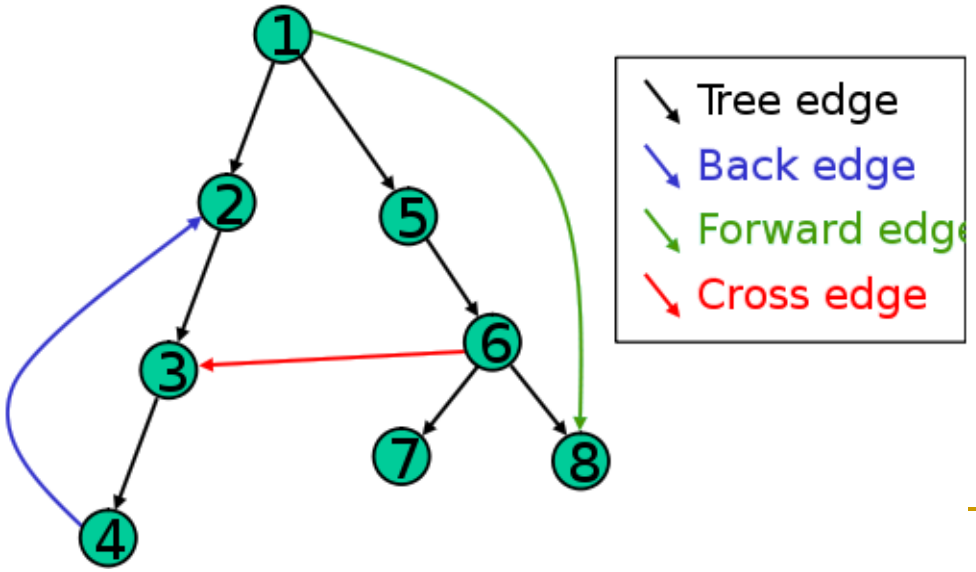
Edge Classification for DFS

- *Tree edges* are actually part of the DFS forest.
- *Back edges* lead to an ancestor in the DFS tree.
- *Forward edges* lead from a node to a nonchild descendant in the DFS tree.
- *Cross edges* lead to neither descendant nor ancestor; they therefore lead to a node that has already been completely explored (that is, already processed).



边分类规则

- 一条边(u, v)可以按如下规则分类
  - 树边(Tree Edges, T): v通过边(u, v)发现,
  - 后向边(Back Edges, B): u是v的后代
  - 前向边(Forward Edges, F): v是u的后代
  - 交叉边(Cross Edges, C): 其他边，可以连接同一个DFS树中没有后代关系的两个结点, 也可以连接不同DFS树中的结点
- 判断后代关系
  可以借助定理1



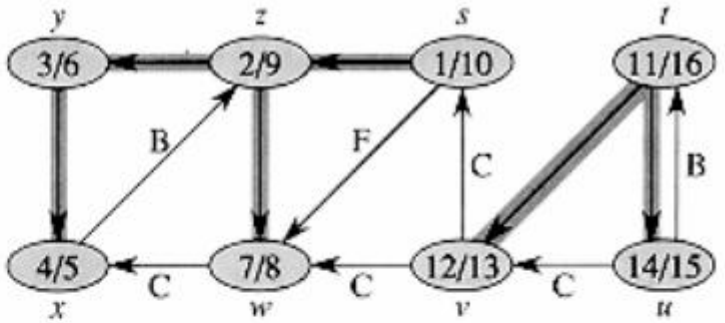
边分类算法

- 当(u, v)第一次被遍历, 考虑v的颜色
  - 白色, (u,v)为Tree边
  - 灰色, (u,v)为Back边 (*只有它的祖先*是灰色)
  - 黑色: (u,v)为Forward边或Cross边. 此时需要进一步判断
    - pre[u]<pre[v]: Forward边 (v是u的后代, 因此为F边)
    - pre[u]>pre[v]: Cross边 (v早就被发现了, 为另一DFS树中)
- 时间复杂度: O(n+m)
- 定理: 无向图只有Tree边和Back边

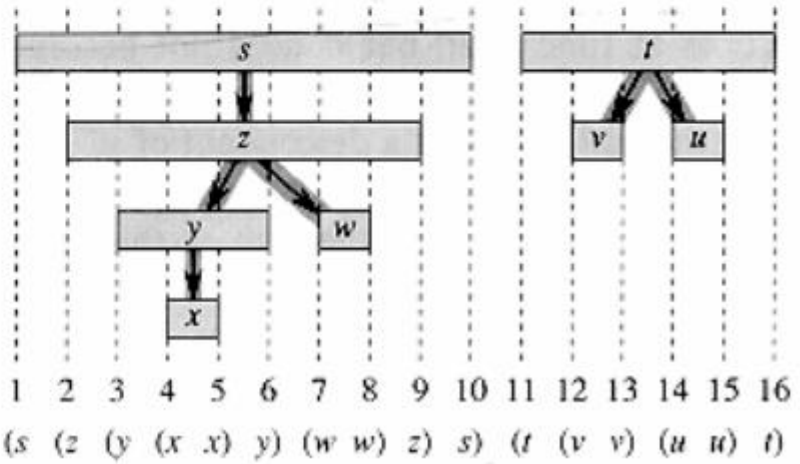
DFS数的性质演示

- 图a: DFS森林
- 图b: 括号性质
- 图c: 重画以后的DFS森林, 边的分类更形象

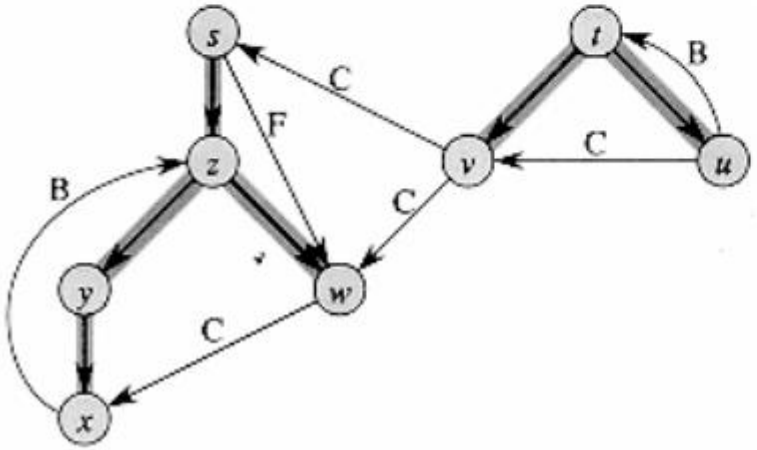
(a)



(b)



(c)



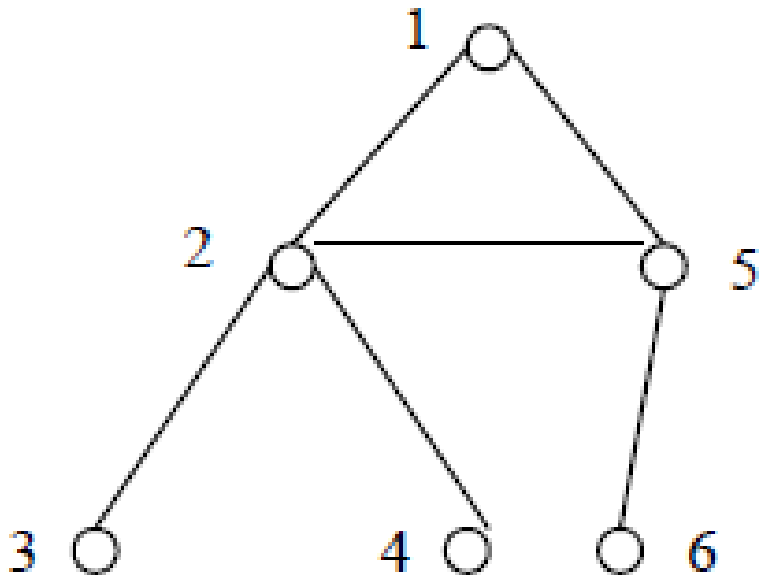
实现细节

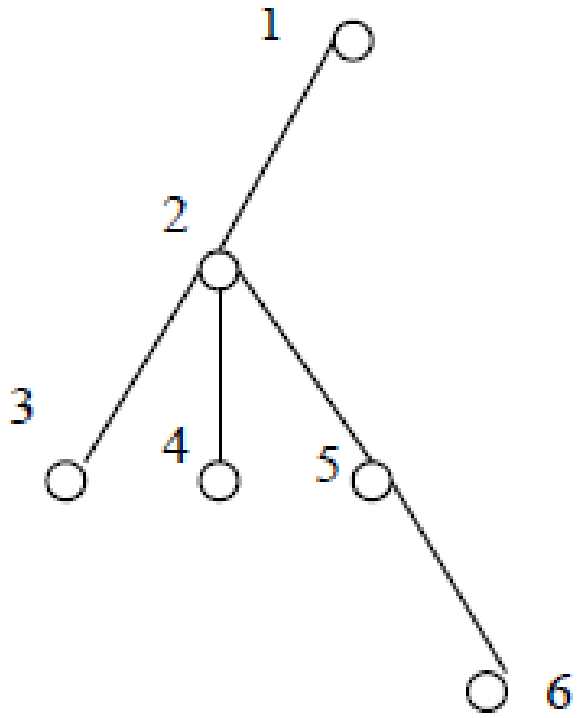
- 颜色值以及时间戳可以省略, 用pre[u]和post[u]代表***点u的先序/后序编号***, 则检查(u,v)可以写为
  - if (pre[v] == 0) dfs(v); //树边, 递归遍历
  - else if (post[v] == 0) show(“Back”); //后向边
  - else if (pre[v] > pre[u]) show(“Forward”); // 前向边
  - else show(“Cross”); // 交叉边
- pre和post的初值均为0

```
/* 有向图的深度优先搜索标记
   INIT: edge[][]邻接矩阵; pre[], post[], tag全置0;
   CALL: dfstag(i, n);   pre/post:开始/结束时间      */
int edge[V][V], pre[V], post[V], tag = 0;
void dfstag(int cur, int n)
{  // vertex: 0 ~ n-1
  pre[cur] = ++tag;
  for (int i=0; i<n; ++i) if (edge[cur][i]) {
    if (0 == pre[i]) {
      printf("Tree Edge!\n");
      dfstag(i, n); }
    else {
      if (0 == post[i]) printf("Back Edge!\n");
      else if(pre[i]>pre[cur]) printf("Down Edge!\n");
      else printf("Cross Edge!\n");
    }
  }
  post[cur] = ++tag;
}
```

DFS in Undirected Graph: No Cross Edges in DFS

- When expanding 2, we would discover 5, so the tree would look like:





DFS Application: Finding Cycles

- *Back edges* are the key to finding a cycle in an undirected graph.
- Any *back edge* going from x to an ancestor y creates a cycle with the path in the tree from y to x.

无根树转有根树

- 输入一个n个结点的无根树的各条边，并指定其中一个结点作为结点，要求把该树转化为有根树，输出各个结点的父亲编号。n<=10^6。

无根树转有根树

- 存储结构

  n太大，邻接矩阵存储不合适。借用vector,空间O(n)

```
vector<int> g[maxn];
void read_tree()
{
    int u,v;    scanf(“%d”,&n);
    for(int i=0;i<n-1;i++) {
        scanf(“%d”, &u, &v);
        g[u].push_back(v); g[v].push_back(u);
    }
}
```

无根树转有根树

- 转化过程

```
void dfs(int u, int fa) //递归转化u为根的子树,u的父亲为fa
{
  int d = g[u].size();  //结点u的相邻点个数
  for(int i=0;i<d;i++)
  {
    int v = g[u][i];    //结点u的第i个相邻点
    //把v的父亲设为u，然后递归转化以v为根的子树
    if (v!=fa) dfs(v,p[v]=u);
}
```

主程序中设置p[root] = -1 （表示根节点的父亲不存在），然后调用dfs(root,-1)即可。

练习

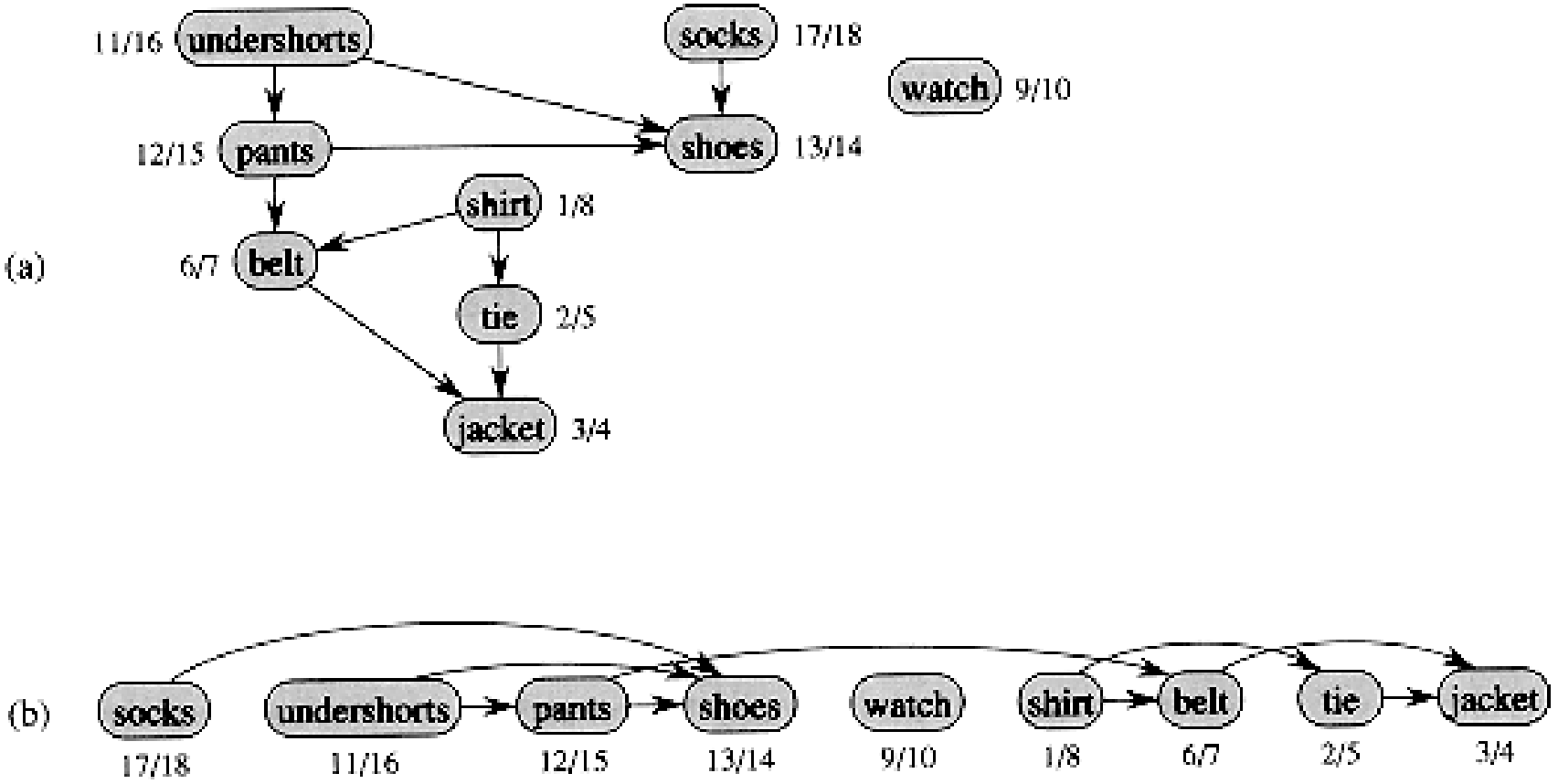
- 对于三种颜色WHITE, GRAY和BLACK, 作一个3*3表格, 判断一种颜色到另一种颜色是否可能有边, 如有可能, 边的类型如何
- 对于边(u,v), 证明它是
  - T或F边当且仅当pre[u]<pre[v]<post[v]<post[u]
  - B边当且仅当pre[v]<pre[u]<post[u]<post[v]
  - C边当且仅当pre[v]<post[v]<pre[u]<post[u]
  - 如何区分T边和F边?
- 修改DFS算法, 使得对于无向图, 可以求出每个点i所处的连通分量编号cc[i]

Topological Sorting

- In graph theory, a **topological sort** or **topological ordering** of a directed acyclic graph (DAG) is a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges. Every DAG has one or more topological sorts.
- More formally, define the reachability relation *R* over the nodes of the DAG such that *xRy* if and only if there is a directed path from *x* to *y*. Then, *R* is a partial order, and a topological sort is a linear extension of this partial order, that is, a total order compatible with the partial order.

Topological Order(拓扑顺序)

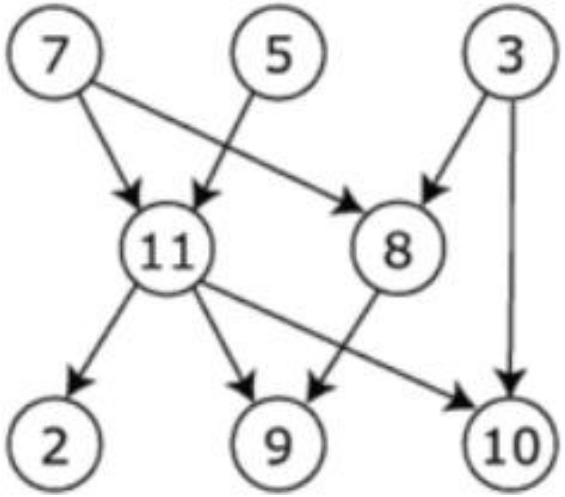
- Dressing Order



Applications of Topological Sorting

- The canonical application of topological sorting (topological order) is in scheduling a sequence of jobs or tasks; topological sorting algorithms were first studied in the early 1960s in the context of the PERT technique for scheduling in project management (Jarnagin 1960).
- In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, and resolving symbol dependencies in linkers.

Example of Topological Sorting



- The graph shown to the left has many valid topological sorts, including:
  - 7, 5, 3, 11, 8, 2, 9, 10 (visual left-to-right, top-to-bottom)
  - 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
  - 3, 7, 8, 5, 11, 10, 2, 9
  - 5, 7, 3, 8, 11, 10, 9, 2 (least number of edges first)
  - 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
  - 7, 5, 11, 2, 3, 8, 9, 10

Topological Sorting Algorithm One

- Kahn (1962), (O(|V|+|E|)).
- First, find a the complete set of “source vertices" which have no incoming edges; at least one such node must exist if graph is acyclic. Deleting all the outgoing edges of these source vertices will create new source vertices, which can then sit comfortably to the immediate right of the first set. We repeat until all vertices are accounted for.

Topological Sorting Algorithm One

L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while S is non-empty **do**
    remove a node n from S
    insert n into L
    for each node m with an edge *e* from n to m **do**
        remove edge e from the graph
        if m has no other incoming edges **then**
            insert m into S
    if graph has edges **then**
        output error message (graph has at least one cycle)
    else output message (proposed topologically sorted order: L)

Topological Sorting Algorithm One

- If the graph was a DAG, a solution is contained in the list L (the solution is not unique). Otherwise, the graph has at least one cycle and therefore a topological sorting is impossible.
- Note that, reflecting the non-uniqueness of the resulting sort, the structure S can be simply a set or a queue or a stack. Depending on the order that nodes n are removed from set S, a different solution is created.

Topological Sort Algorithm Two

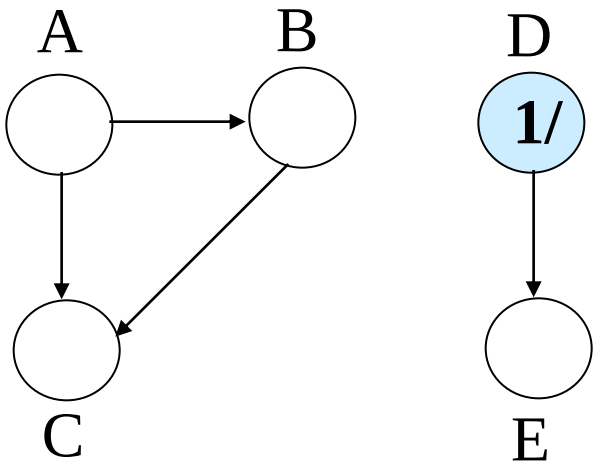
- Performed on a DAG.
- Linear ordering of the vertices of G such that if $(u, v) \in E$, then u appears somewhere before v.

Topological-Sort (G)

1. call DFS(G) to compute finishing times $f[v]$ for all $v \in V$
2. as each vertex is finished, insert it onto the front of a linked list
3. **return** the linked list of vertices

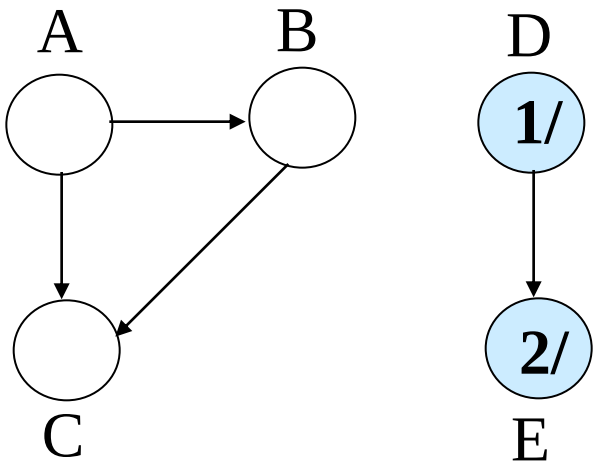
Time: $\Theta(V + E)$.

Topological Sort Example



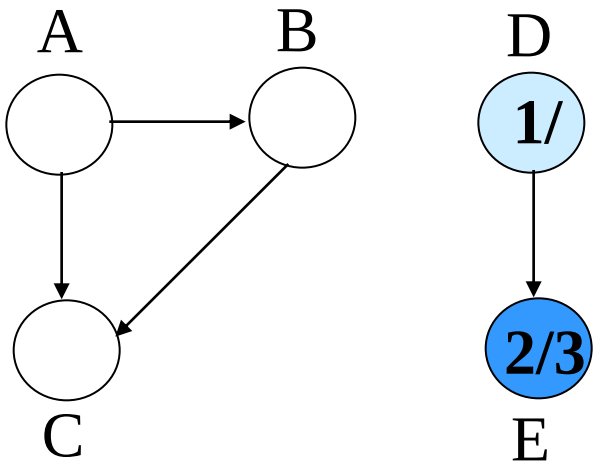
Linked List:

Topological Sort Example



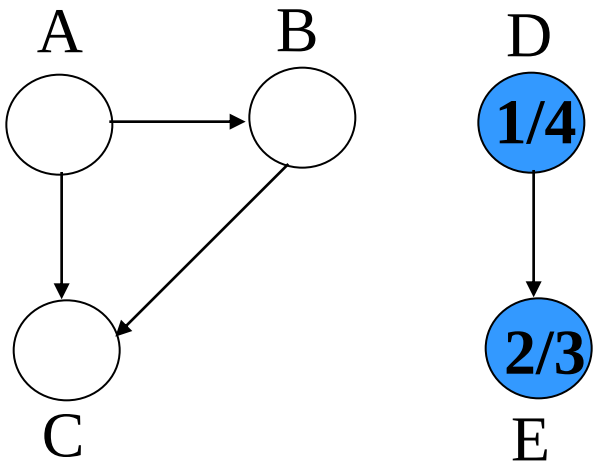
Linked List:

Topological Sort Example

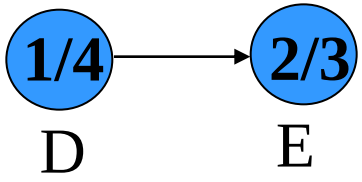


Linked List:

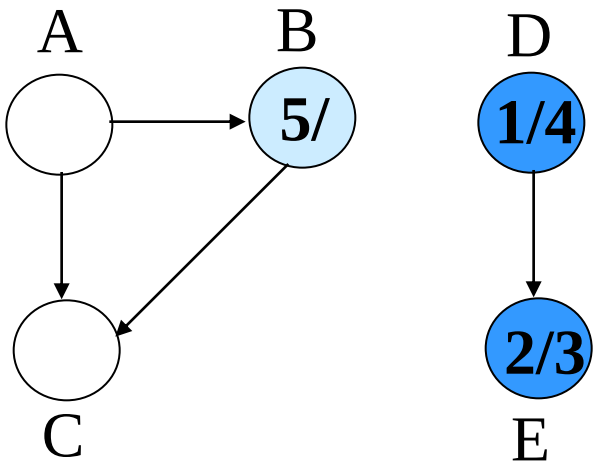
Topological Sort Example



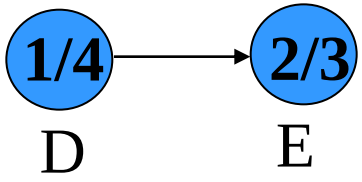
Linked List:



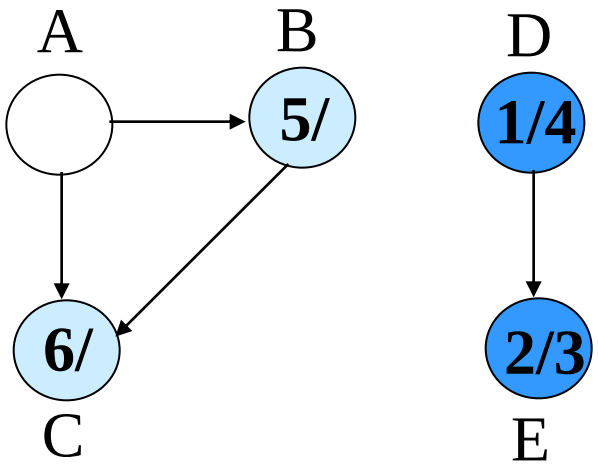
Topological Sort Example



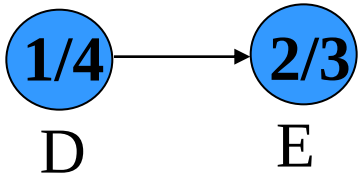
Linked List:



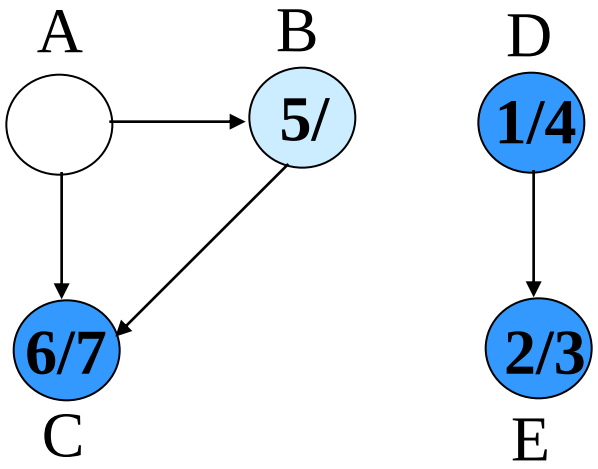
Topological Sort Example



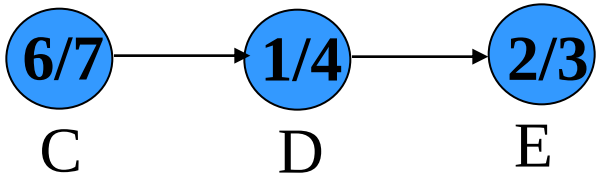
Linked List:



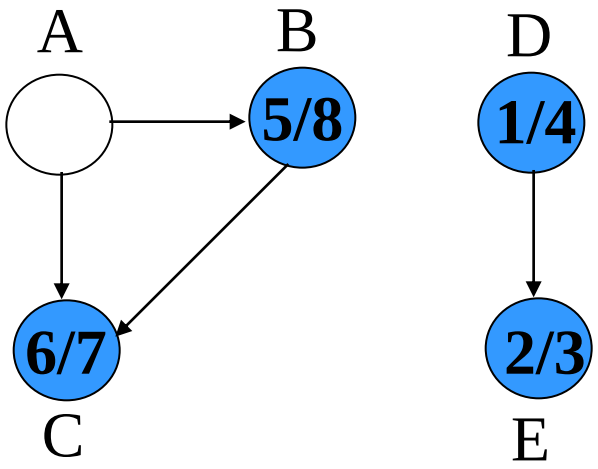
Topological Sort Example



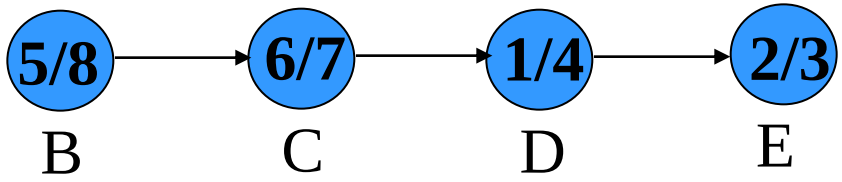
Linked List:



Topological Sort Example



Linked List:



Application Toposort

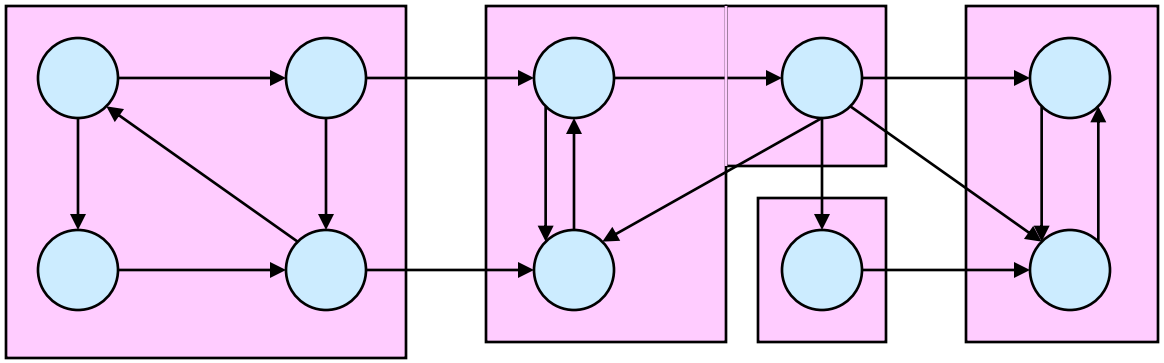
- 假设有n个变量，还有m个二元组(u,v)，表示变量u < v。那么，所有变量从小到大排列起来应该是什么样子的呢？
- 例如有4个变量a, b, c, d，若已知a < b, c < b, d < c, 则这4个变量的排序可能是a<d<c<b。可能有多种排序结果(如d<a<c<b)，你只需找出其中一个即可。

Application Toposort

```
int color[MAXN], topo[MAXN], t;
bool dfs(int u)
{
   color[u] = 1; //u is discovered
   for(int v=0; v<n; v++) if (G[u][v])
   {
       if (color[v]==1) return false;//backedge exist
       else if (!color[v] && !dfs(v)) return false;
   }
   color[u] = 2; topo[--t] = u;
   return true;
}
bool toposort()
{
   t = n; memset(color,0,sizeof(color));
   for(int u=0; u<n; u++) if (!c[u] && !dfs(u))
       return false;
   return true;
}
```

Strongly Connected Components

- G is strongly connected if every pair (u, v) of vertices in G is reachable from one another.
- A **strongly connected component** (***SCC***) of G is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$ exist.

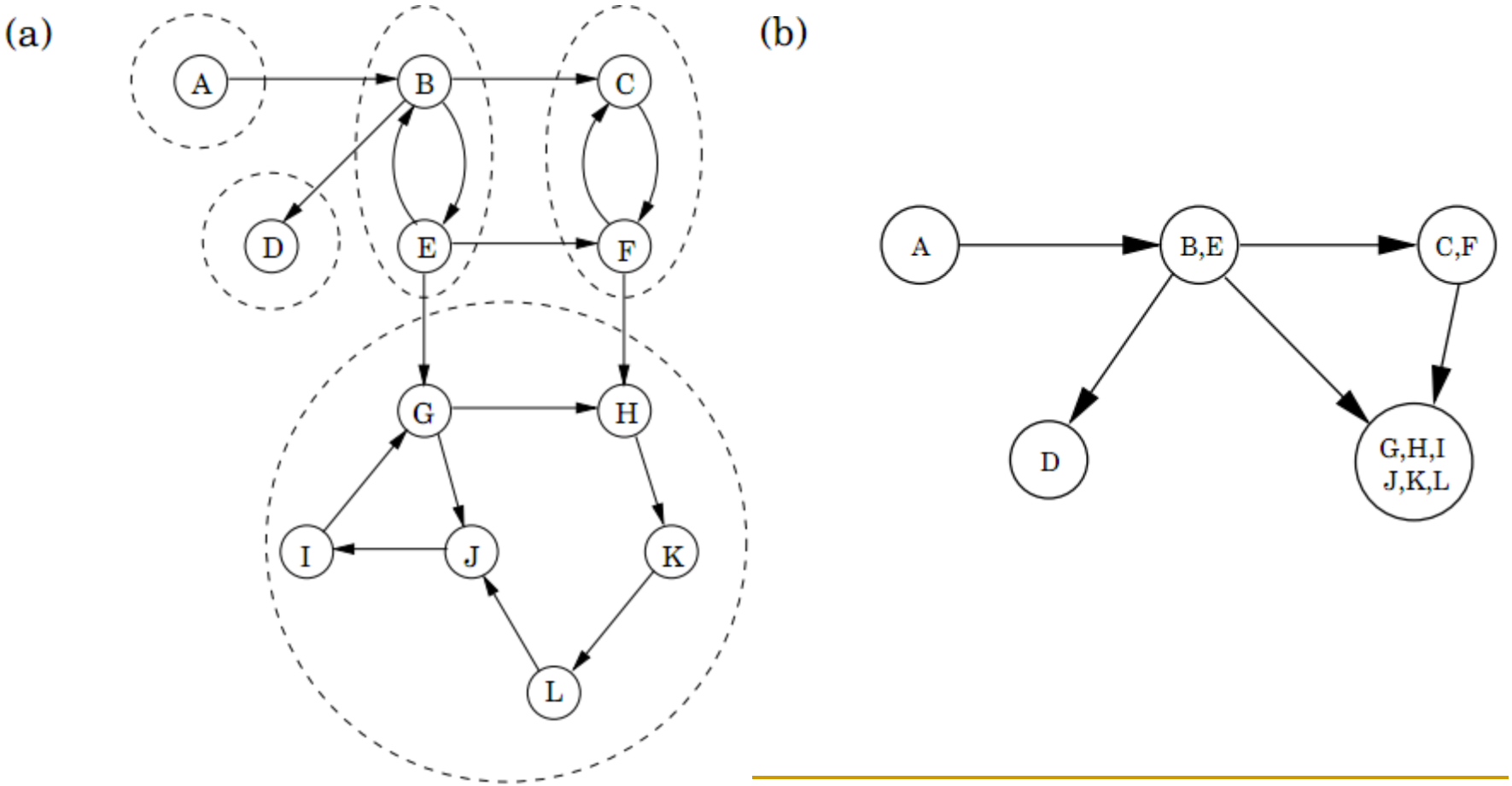


Strongly Connected Components

- There is an elegant, linear time algorithm to find the strongly connected components of a directed graph using DFS which is similar to the algorithm for biconnected components.
- The algorithm is based on the observation that it is easy to find a directed cycle using a depth-first search, since any back edge plus the down path in the DFS tree gives such a cycle. All vertices in the cycle must be in the same strongly connected component. Thus, we can shrink the vertices on this cycle down to a single vertex representing the component, and then repeat. This process terminates when no directed cycle remains, and each vertex represents a different SCC.

Strongly Connected Components

(a) A directed graph and its strongly connected components. (b) The meta-graph.



Strongly Connected Graph

- How to know whether a directed graph is *strongly connected*?
- The simplest linear-time algorithm performs a search from some vertex v to demonstrate that the entire graph is reachable from v. We then construct a graph G' where we reverse all the edges of G. A traversal of G' from v suffices to decide whether all vertices of G can reach v.

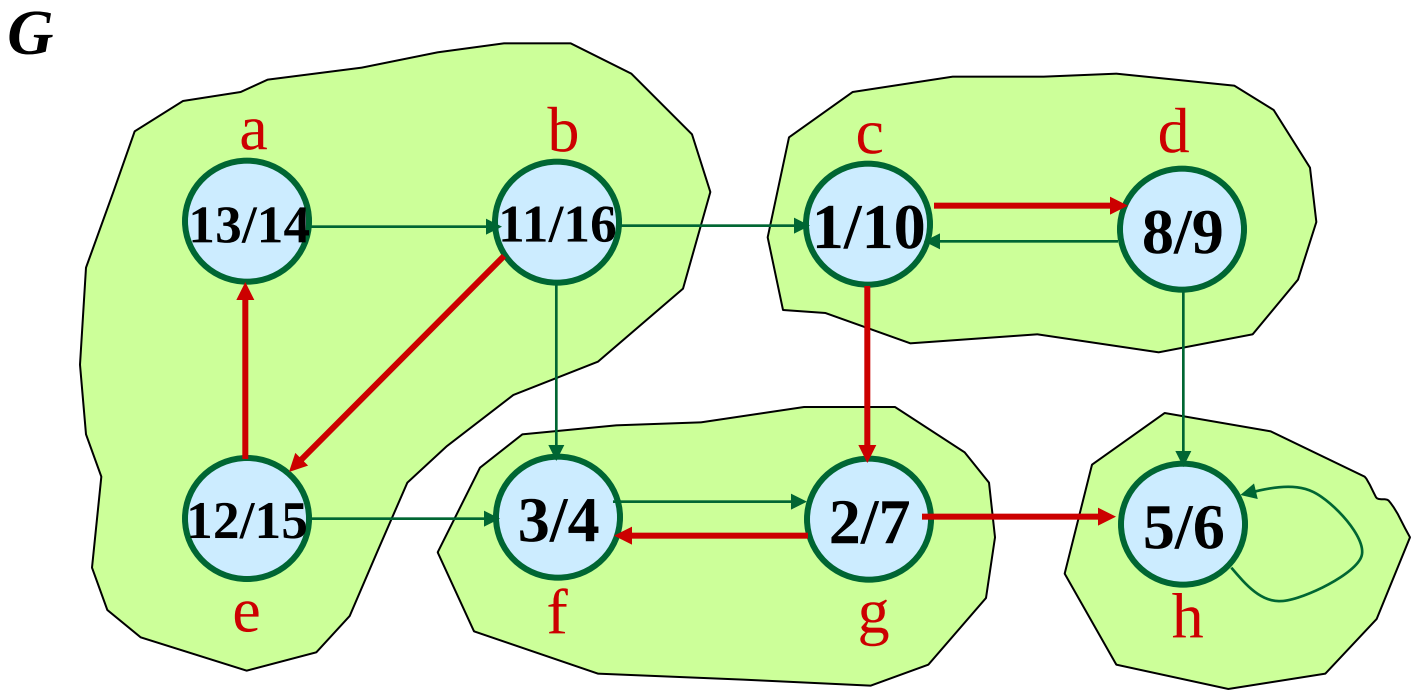
Kosaraju algorithm to determine

SCC(G)

1. call DFS(G) to compute finishing times f [u] for all u
2. compute G^T
3. call DFS(G^T), but in the main loop, consider vertices in order of decreasing f[u]. (as in line 1, topological order)
4. output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC

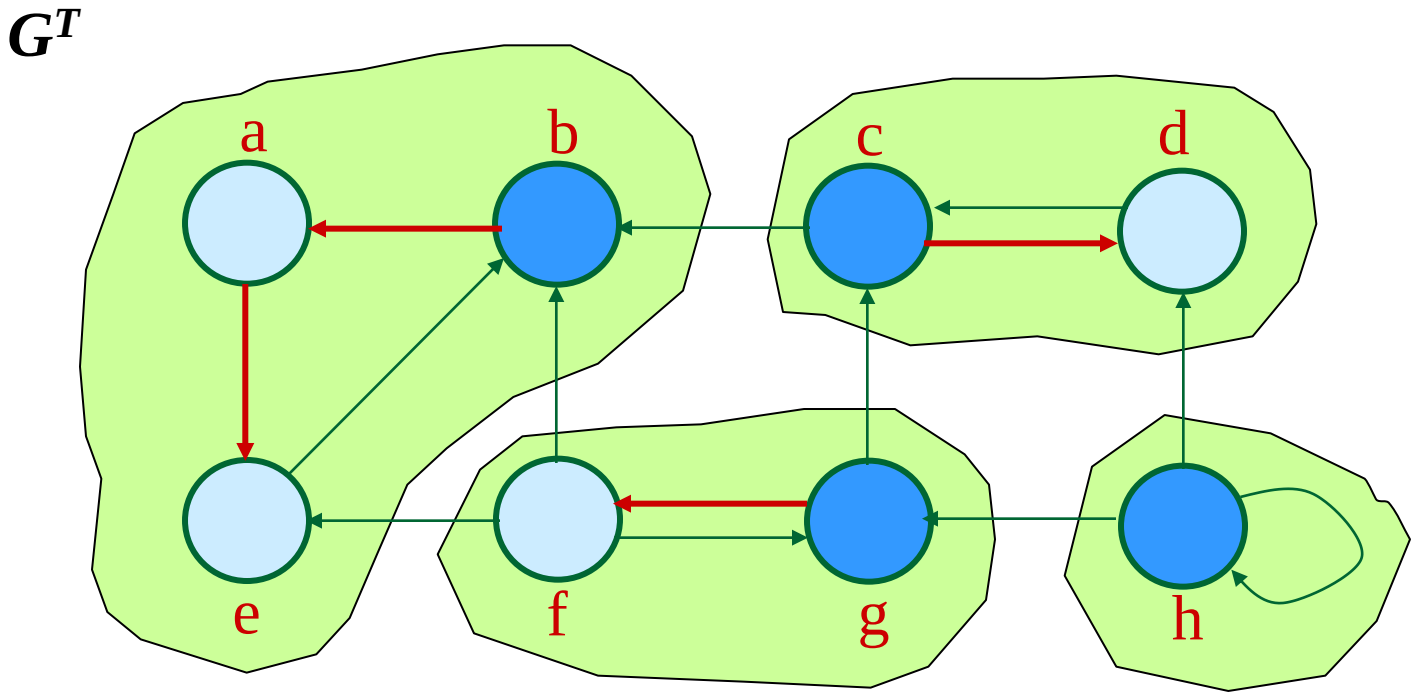
Time: $\Theta(V + E)$.

Example



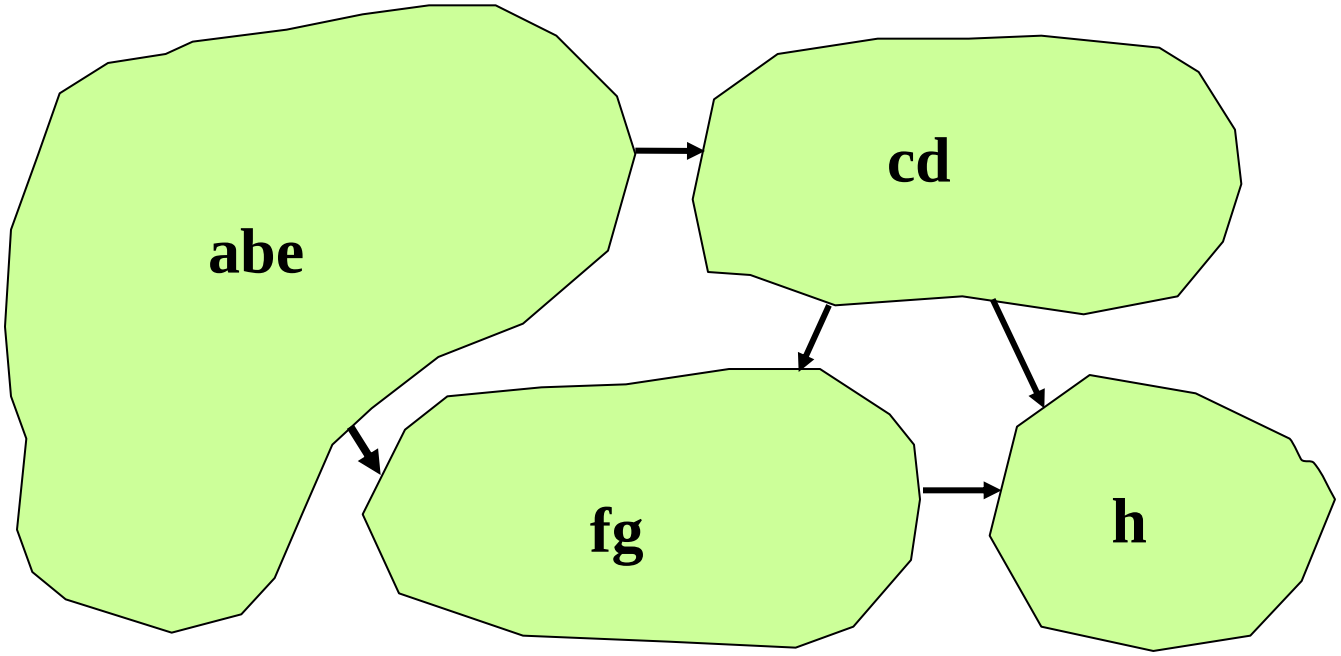
in order of decreasing f[*u*]: b, e, a, c, d, g, h, f

Example



in order of decreasing f[u]: b, e, a, c, d, g, h, f

Example



Kosaraju算法的正确性

- 当按照f值排序以后，第二次DFS 是按照SCC的拓扑顺序进行(以后所指d[u] 和f[u] 都是第一次DFS 所得到的值)
- 记d(C) 和f(C) 分别表示集合U所有元素的最早发现时间和最晚完成时间，有如下定理:
- 定理：对于两个SCC C和C', 如果C到C' 有边，则f(C)>f(C')
  – 情况一: d(C) < d(C')，考虑C 中第一个被发现的点x, 则C' 全为白色，而C 到C' 有边，故x到C' 中每个点都有白色路径. 这样, C和C' 全是x的后代，因此f(C) > f(C')
  – 情况二: d(C) > d(C'). 由于从C' 不可到达C, 因此必须等C' 全部访问完毕才能访问C. 因此f(C) > f(C')
- 推论:对于两个SCC C和C', 如果在G^T中C到C' 有边，则f(C)<f(C')

Kosaraju算法的正确性

- 首先考虑f(C) 最大的强连通分量. 显然, 此次DFS将访问C 的所有点, 问题是是否可能访问其他连通分量的点? 答案是否定的, 因为根据推论, 如果在G^T中C到另外某个C' 存在边, 一定有f(C)<f(C'), 因此第一棵DFS恰好包含C. 由数学归纳法可知, 每次从当前强连通分量出发的边一定连到f值更大的强连通分量, 而它们是已经被遍历过的, 不会在DFS 树形成多余结点

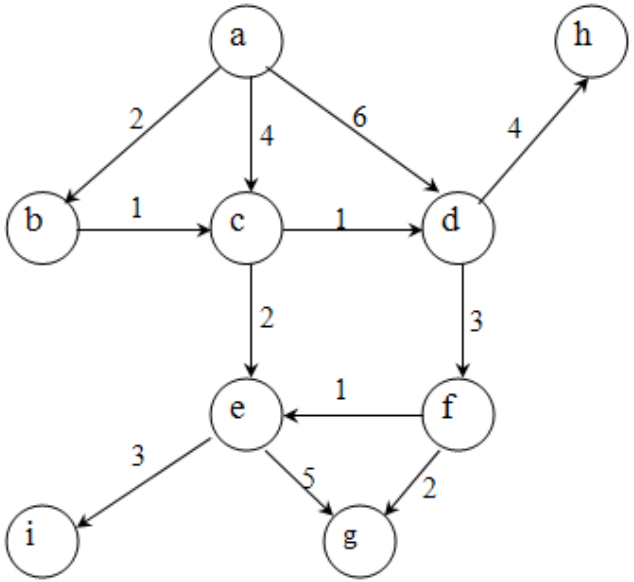
局限性

•SC C 算法的简单历史
– 第一个SCC算法: Tarjan 1972 (经典算法)
–80 年代: 精美的Kosaraju 算法(后来发现和1972年苏联发现的算法本质相同)
– 1999 Gabow 在60年代的思想上提出了第三个线性算法
• 局限性: Kosaraju 算法需要计算图的转置和两次DFS, 时间效率不如Tarjan 算法和Gabow算法

Backtracking and Depth-First Search

- Depth-first search uses essentially the same idea as backtracking.
- Both involve exhaustively searching all possibilities by advancing if it is possible, and backing up as soon as there is no unexplored possibility for further advancement.
- Both are most easily understood as recursive algorithms.

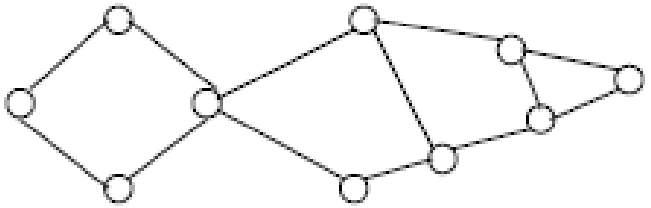
Review – DFS, BFS, Topological Sort



- Do DFS and BFS starting from vertex `a'.
- DFS/BFS traversal depends on the ordering of neighbors…
- Any valid DFS/BFS is accepted.
  - But usually we order neighbor alphabetically.
- DFS:
  - a, b, c, d, f, e, g, i, h (my default answer) OR
  - a, b, c, d, h, f, g, e, i (if you adopt other vertex ordering) OR
  - a, b, c, e, i, g, d, h, f OR other possible sequences
- BFS:
  - a, b, c, d, e, f, h, g, i (my default answer) OR
  - a, d, c, b, h, f, e, g, i (if you adopt other vertex ordering) OR other possible sequences
  - Note that a, b, c, d, f, e, h, g, i is not an acceptable sequence, why?
  - We are using a queue, e will be inside first after visiting c, so e is executed first before f…
- Homework:
  - Try doing DFS/BFS from other nodes
- Write the topological order of the vertices for the graph given above.
  - a, b, c, d, f, h, e, g, i (my default answer) OR
  - a, b, c, d, h, f, e, g, i (if you adopt other vertex ordering)

DFS Application: Articulation Vertices

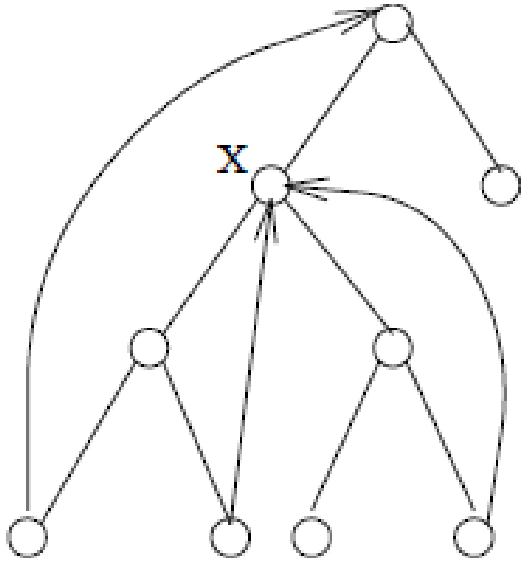
- Suppose you are a terrorist, seeking to disrupt the telephone network. Which station do you blow up?



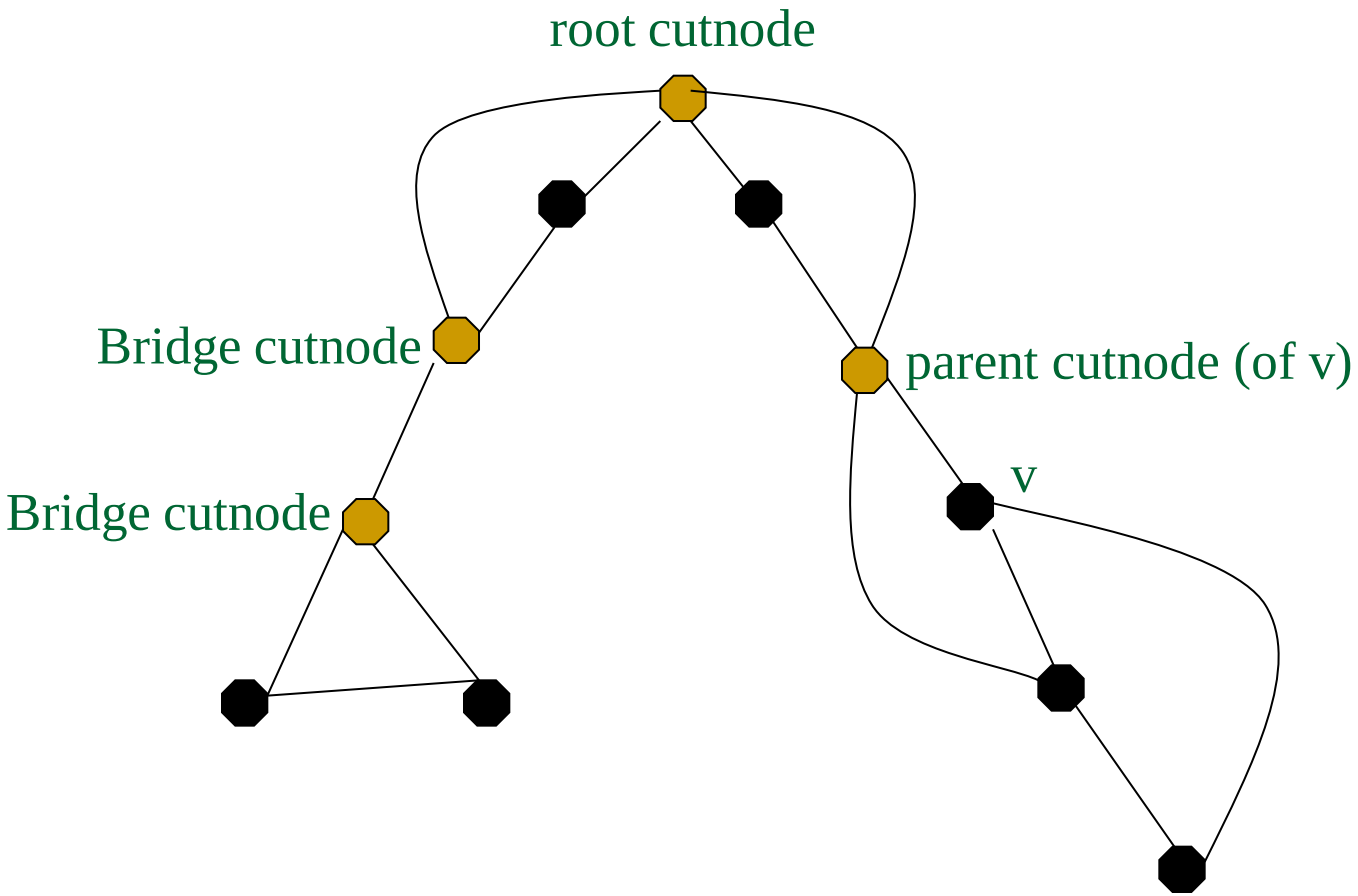
- An *articulation vertex* is a vertex of a connected graph whose deletion disconnects the graph.
- Articulation vertices can be found in O(n(m+n)) – just delete each vertex to do a DFS on the remaining graph to see if it is connected.

A Faster O(n + m) DFS Algorithm

- In a DFS tree, a vertex v (other than the root) is an articulation vertex iff v is not a leaf and some subtree of v has no back edge incident to a proper ancestor of v.
- The root is a special case since it has no ancestors.
- X is an articulation vertex since the right subtree does not have a back edge to a proper ancestor.



The three cases of articulation vertices



```
/* 无向图连通度(割)
   INIT: edge[][]邻接矩阵;vis[],pre[],anc[],deg[]置为0;
   CALL: dfs(0, -1, 1, n);
   k=deg[0], deg[i]+1(i=1…n-1)为删除该节点后得到的连通图个数
   注意:0作为根比较特殊! */
int edge[V][V], anc[V], pre[V], vis[V], deg[V];
void dfs(int cur, int father, int dep, int n)
{// vertex: 0 ~ n-1
  int cnt = 0;
  vis[cur] = 1; pre[cur] = anc[cur] = dep;
  for (int i=0; i<n; ++i) if (edge[cur][i]) {
    if (i != father && 1 == vis[i]) {
      if (pre[i] < anc[cur]) anc[cur] = pre[i];//back edge }
    if (0 == vis[i]) {           //tree edge
      dfs(i, cur, dep+1, n);
      ++cnt; // 分支个数
      if (anc[i] < anc[cur]) anc[cur] = anc[i];
      if ((cur==0 && cnt>1) || (cnt!=0 && anc[i]>=pre[cur]))
        ++deg[cur];  // link degree of a vertex
    }
  }
  vis[cur] = 2;
}
```