图结构和基本问题

刘汝佳

目录

1 预备知识

1.1 图的基本概念
1.2 路径、圈和连通性
1.3 生成树、完全图和补图
1.4 特殊图类
1.5 有向图和带权图
1.6 图的ADT
1.7 邻接矩阵
1.8 邻接表和前向星

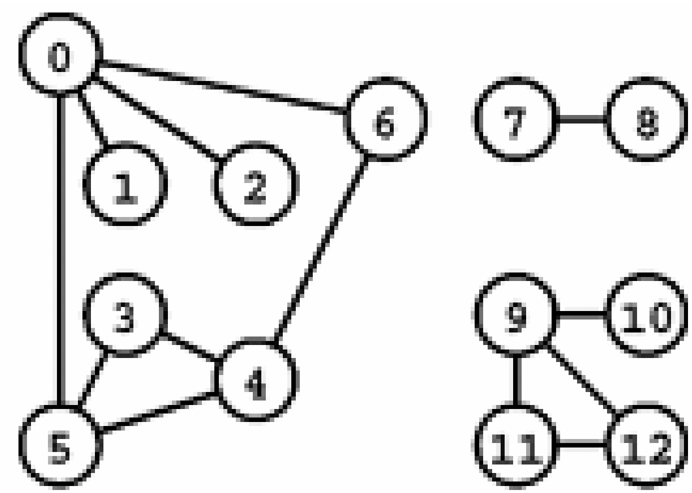
图的基本概念

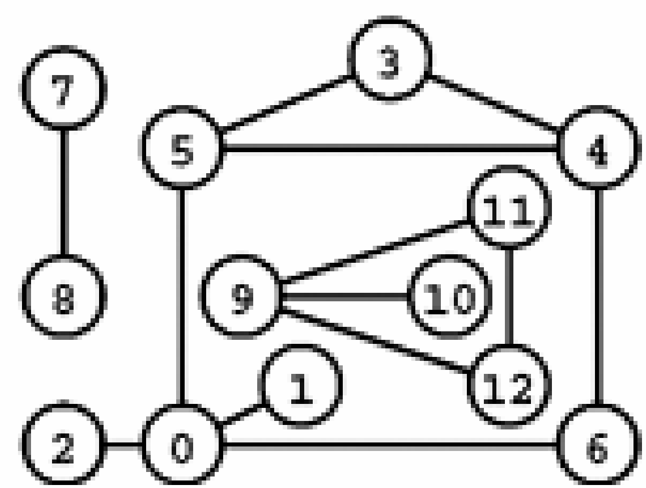
- 图G=(V, E), 即顶点集和边集组成的二元组, 它描述了顶点集的***相互关系***
- 本文只考虑简单图
  - 边的两端不是同一个顶点(没有自环self-loops)
  - 一对结点最多被一条边连接(没有重边parallel edges)
- 注意: 自环和重边在有些情况(见后)很有用
- 图的数学表示
  - 点: 用整数0, 1, 2, …, V-1表示
  - 边: 用无序数对(u, v)表示, 或者表示成u-v
- 边数E不超过V(V-1)/2

图形表示

- 此二图是同一个图的不同表示. 图中并不定义各个结点的位置以及边的形态(直线?曲线?), 关键是有哪些点，哪些点相连
- 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	





其他术语

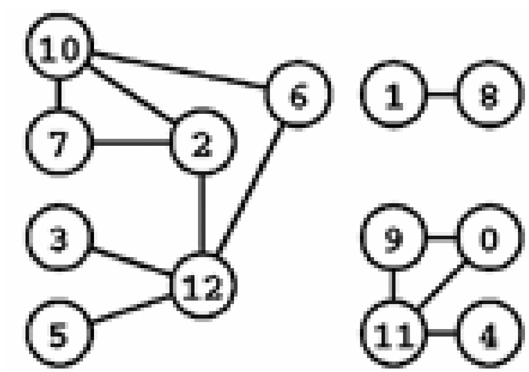
- 其他名称
  - 结点=顶点: vertex, node
  - 边=弧: edge, arc, link
- 对于边e=(u, v)
  - u和v邻接(adjacent)
  - e和u、v关联(incident)
  - 点的度数(degree)是与它关联的边的数目
- 子图
  - 子图(subgraph): 边的子集和相关联的点集
  - 导出子图(induced graph): 点的子集和相关联的边集

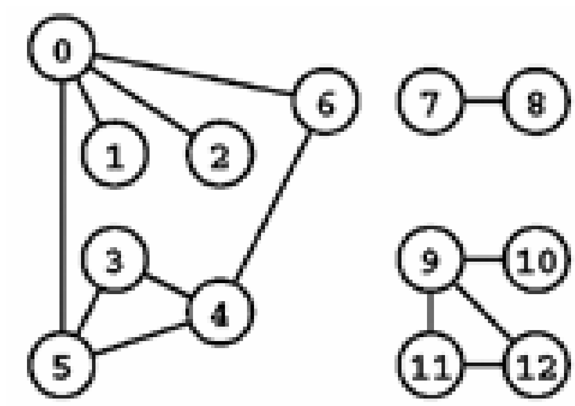
图绘制

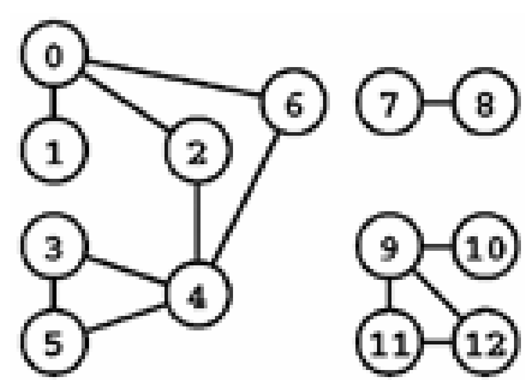
- 图绘制(Graph Drawing): 顶点赋予几何坐标, 边绘制成直线或者曲线
- 平面图(planar graph): 可以绘制成边不相交的形式
- 欧几里得图(euclidean graph)的绘制中,顶点的位置和边是有意义的, 例如地图或者电路图
- 本文的多数算法不使用几何信息

同构

- 上面两个图同构, 但不和下面的图同构







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

连通性

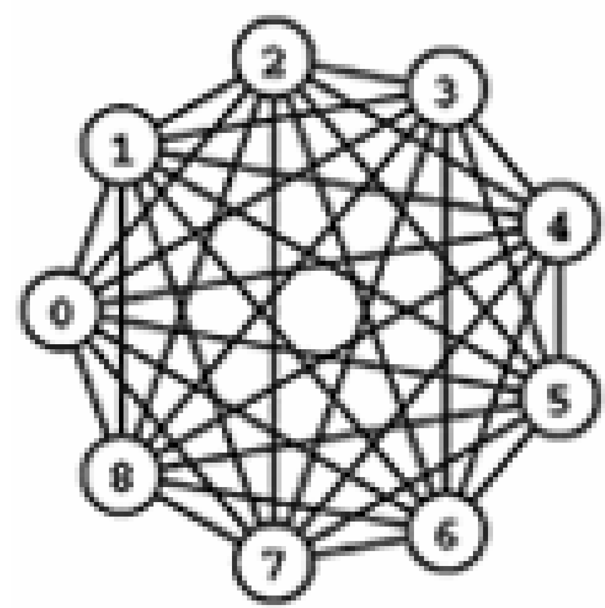
- 如果任意两点都有路径, 则称图是连通(connected)的, 否则称图是非连通的.
- 非连通图有多个连通分量(connected component, cc), 每个连通分量是一个极大连通子图(maximal connected subgraph), 因为任意加一个结点以后将成为非连通图

生成树

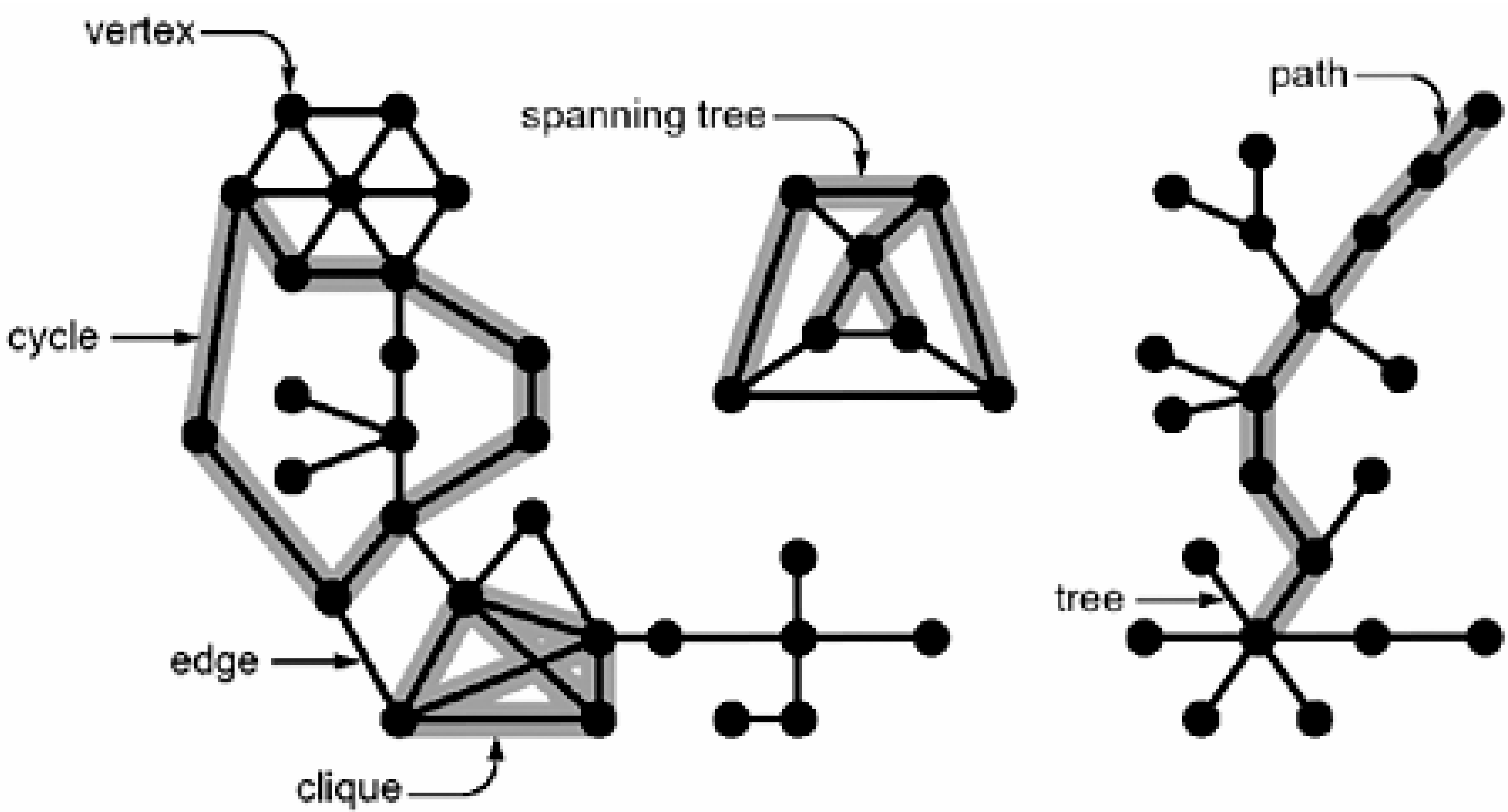
- 连通去圈图成为树(tree)
- 树的集合称为森林(forest)
- 生成树: 包含某图G所有点的树
- 生成森林: 包含某图G所有点的森林
- 一个图G是树当且仅当以下任意一个条件成立
  - G有V-1条边, 无圈
  - G有V-1条边, 连通
  - 任意两点只有唯一的简单路径
  - G连通, 但任意删除一条边后不连通
- 还有其他条件,可类似定义

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

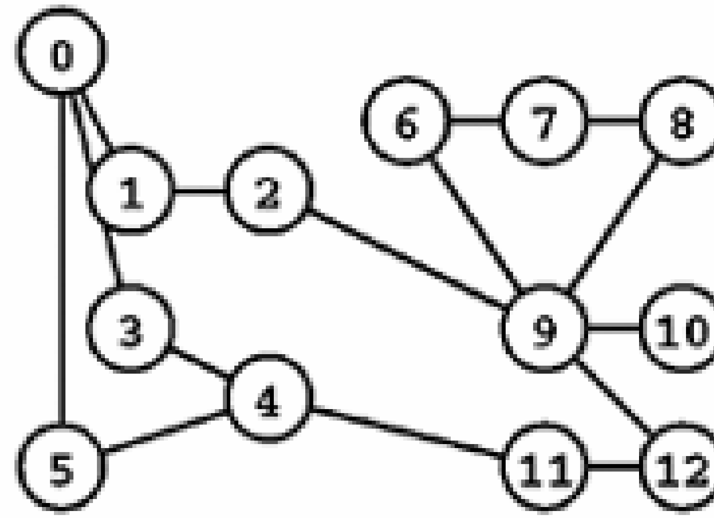


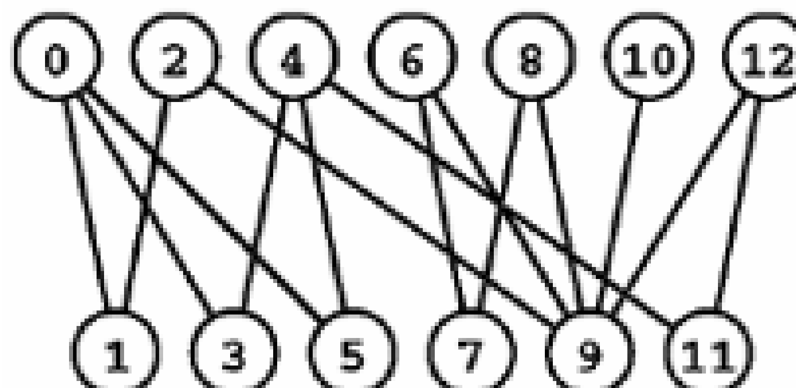
稀疏图和稠密图

- 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)

二分图

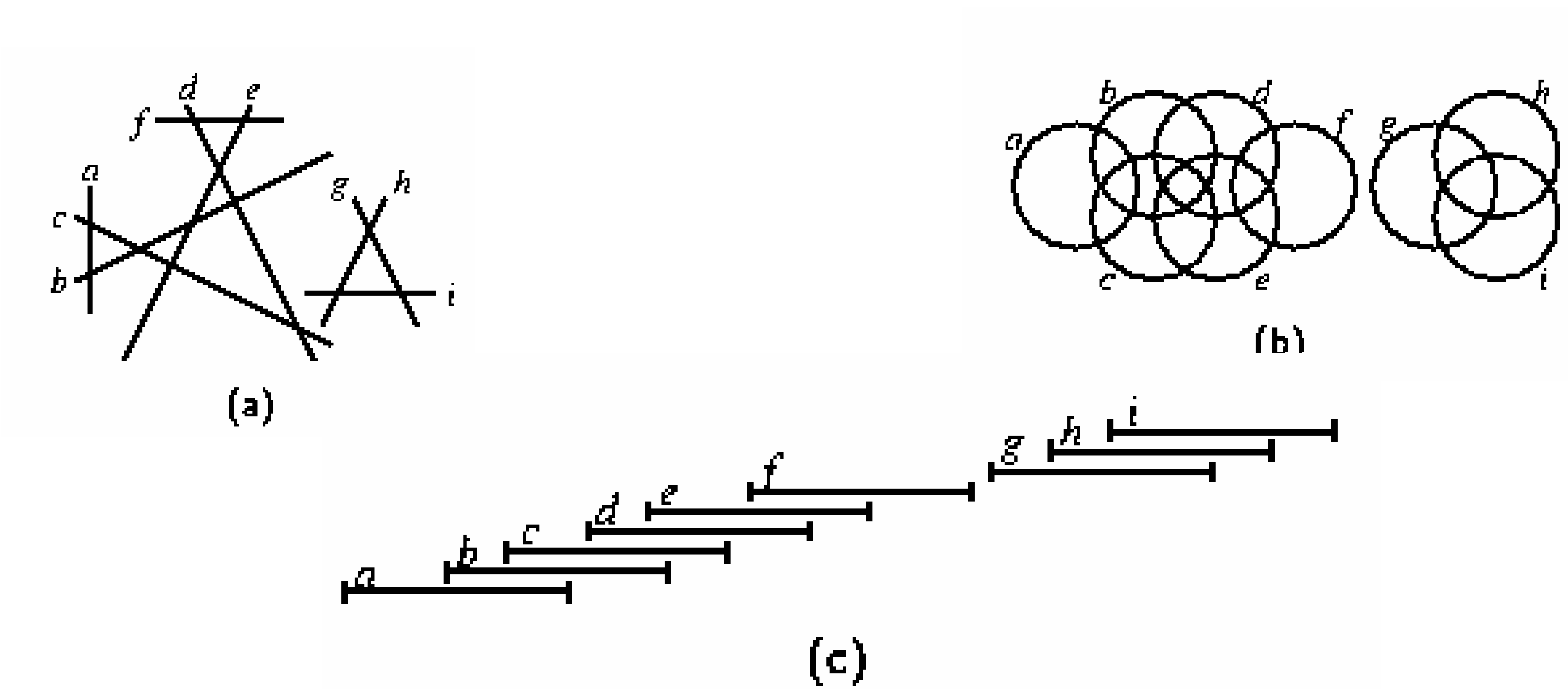
- 可以把结点分成两部分, 每部分之间没有边. 这样的图只有奇圈 (odd-cycle), 即包含奇数条边的圈
- 许多困难问题在二分图上有有效算法





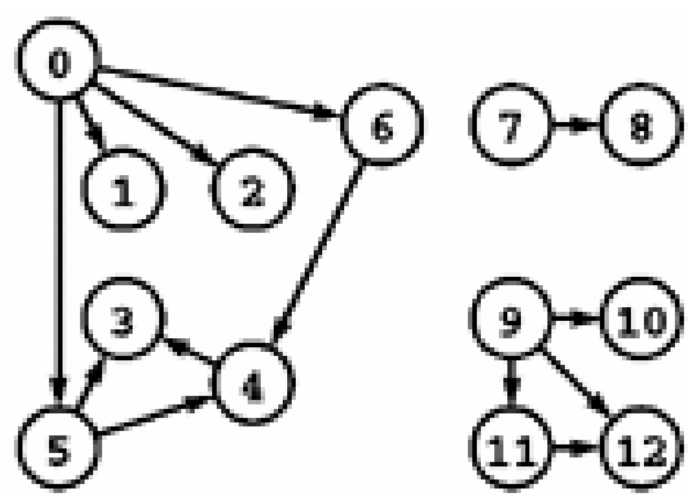
相交图、区间图

- 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



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



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

图的ADT

- 图的ADT如下:
  - V(), E(): 返回顶点数和边数
  - bool directed(): 当且仅当有向图返回真
  - insert(Edge) / remove(edge): 增加/删除边
  - bool edge(int, int): 邻接测试
  - AdjList getAdjList(int): 返回邻居列表
- Edge包含u, v两个成员
- AdjList支持取beg, nxt和end测试

图IO的ADT

- 当图需要进行IO的时候，通常需要支持以下操作
  - scanEZ: 读入边列表(顶点用0,1,…V-1表示)
  - scan: 读入边列表(顶点用符号表示)
  - show: 输出边列表

基本图问题

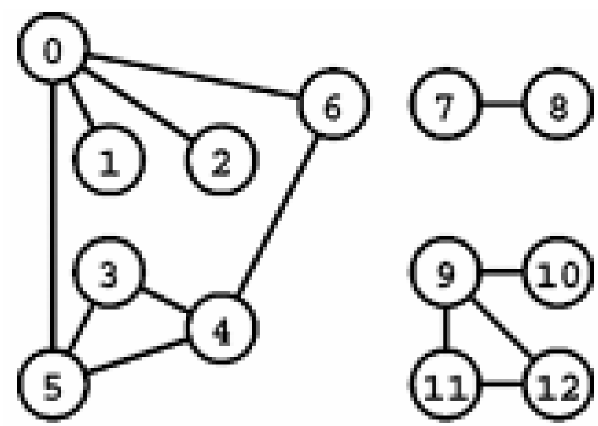
- 和图有关的问题通常有三种
  - 计算图的某种度量值
  - 计算某个子图(边的子集)
  - 回答关于图属性的询问
- 动态问题: 插入/删除边和询问交替

基本表示方法

- 邻接矩阵(adjacency-matrix)
- 邻接表(adjacency-list)
- 前向星(forward star)

基本概念

- A[i,j]=0表示(u,v)不邻接, 1表示邻接



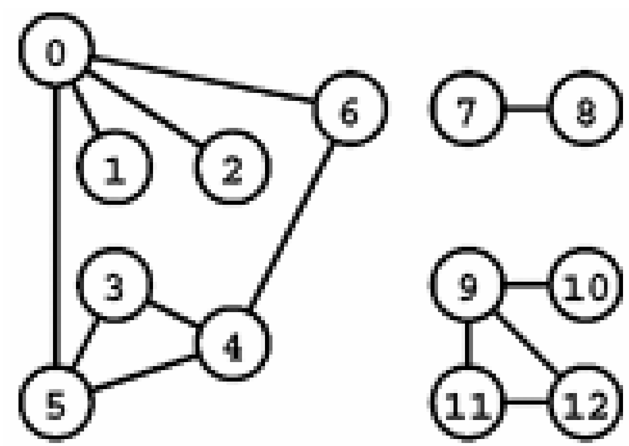
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0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

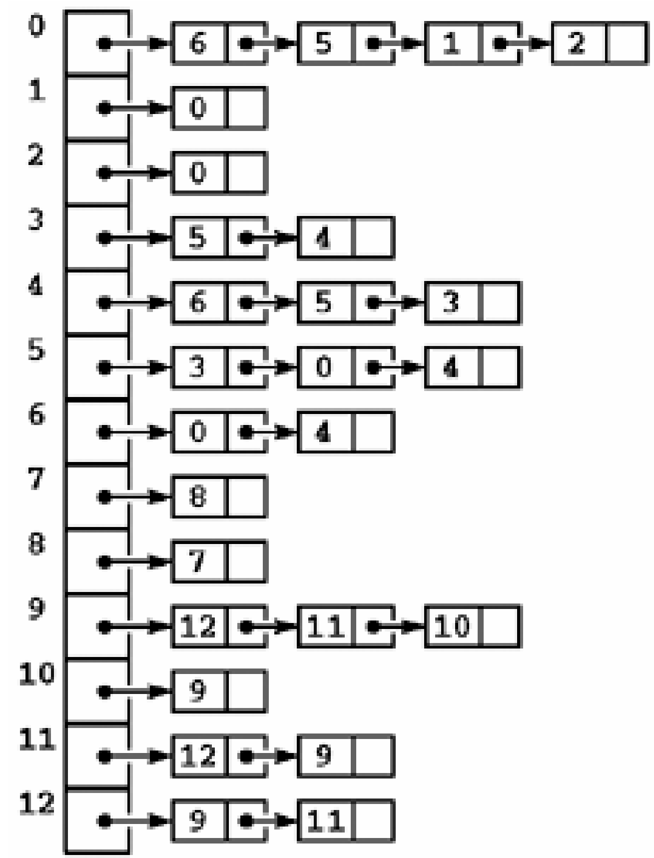
相关问题

- 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?

邻接表

- 每个结点的邻居形成一个链表





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

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

练习

- 给定邻接表, 如何统计每个点的出度和入度?
- 对于有向图的邻接表和邻接矩阵, 分别如何计算图的转置?转置即把所有有向边(u,v)变成(v,u)
- 对于邻接表和邻接矩阵, 设计算法计算图G的平方G^2. 如果G有边(u, v)和(v, w), 则G^2中有边(u, w), 即G^2中的边对应于G中恰走两步可以到达的点对
- 给出有向图的邻接矩阵, 判断是否存在汇(sink), 即入度为|V|-1, 出度为0. 时间复杂度应为O(V)
- 关联矩阵B是一个|V|*|E|的矩阵, 若j是i的出边, 则b_{ij}=-1, 若为入边, b_{ij}=1, 否则为0. 解释BB^T的含义

2 图的遍历及其应用

2.1 宽度优先遍历

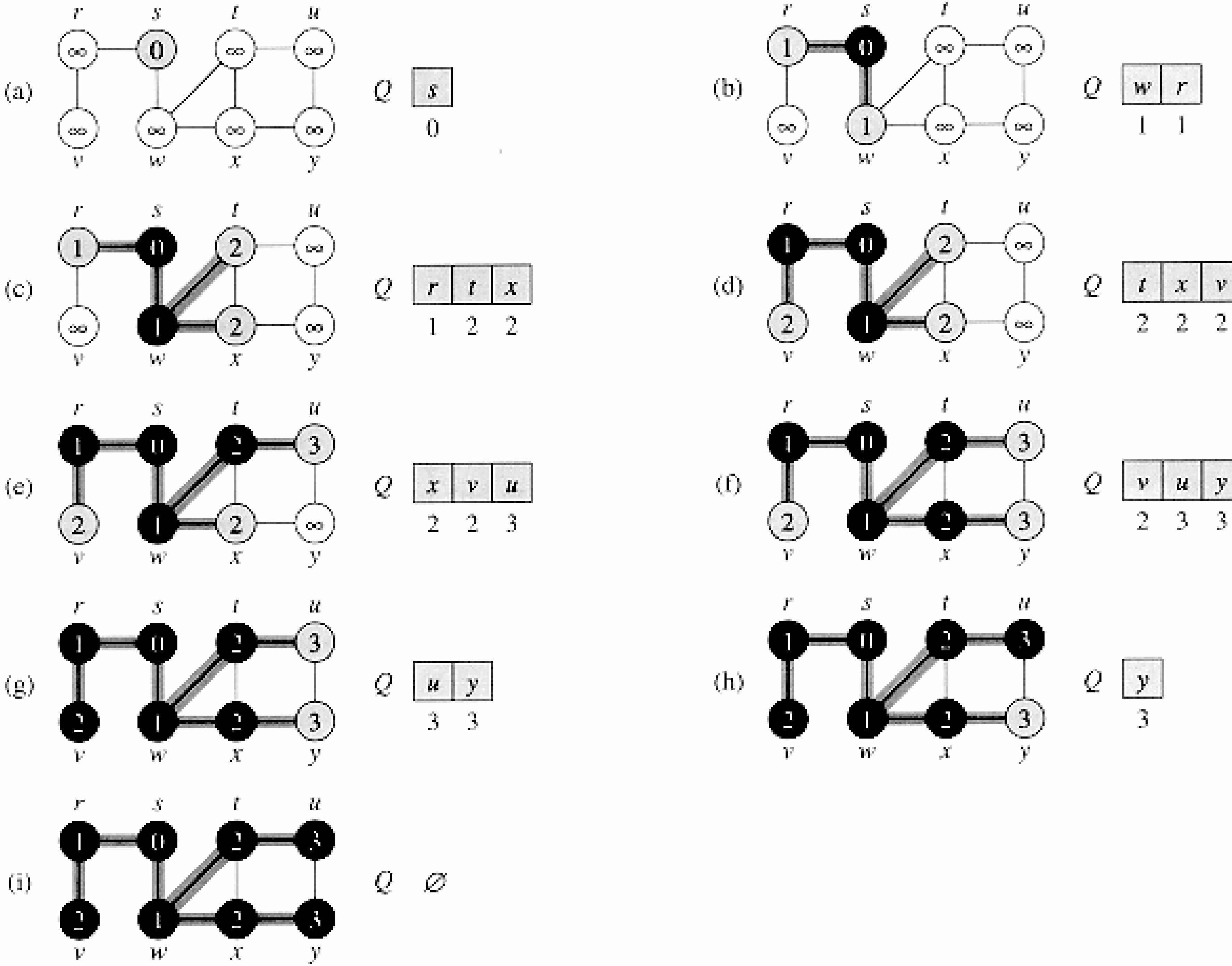
2.2 深度优先遍历和边分类

2.3 拓扑排序

2.4 欧拉回路

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: 没有考虑过的点
  - 黑色: 已经完全考虑过的点
  - 灰色: 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
- 整棵树的根为s


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
r s t u
v w x y
Q

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

练习

- 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?

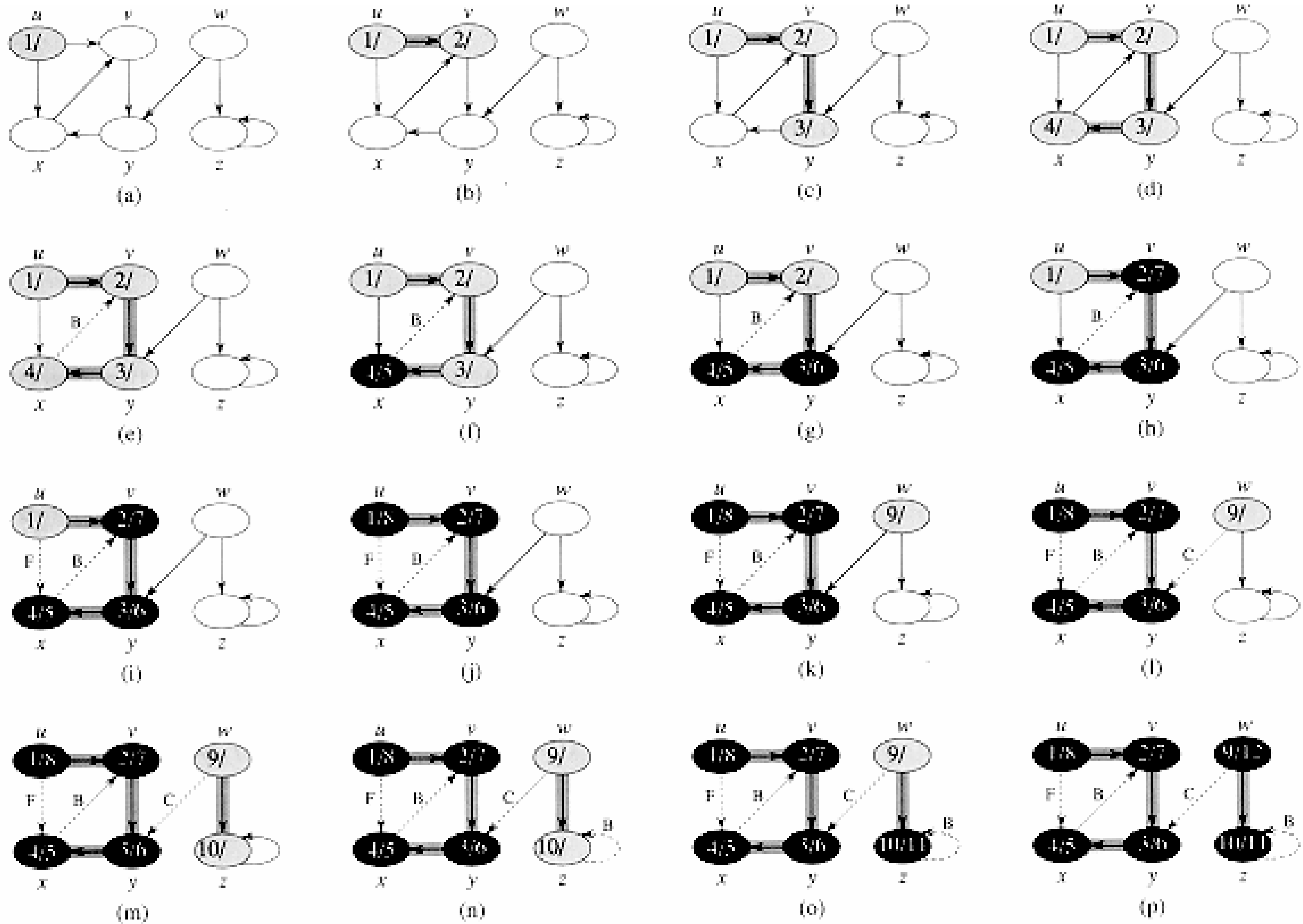
DFS基本算法

- 新发现的结点先扩展
- 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d[v]: 变灰的时间
  - 结束时间f[v]: 变黑的时间
  - 1<=d[v] < f[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-VISIT算法

- 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)
- 时间复杂度为O(n+m)

```
DFS-VISIT(u)
color[u] ← GRAY                      ▷ White vertex u has just been discovered.
d[u] ← time ← time + 1
for each v ∈ Adj[u]                  ▷ Explore edge (u, v).
     do if color[v] = WHITE
           then π[v] ← u
                DFS-VISIT(v)
color[u] ← BLACK                     ▷ Blacken u; it is finished.
f[u] ← time ← time + 1
```



(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)
(m)
(n)
(o)
(p)

DFS树的性质

- 括号结构性质
- 对于任意结点对(u, v), 考虑区间[d[u], f[u]]和[d[v], f[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$d[u]<d[v]<f[v]<f[u]$, 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d[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

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

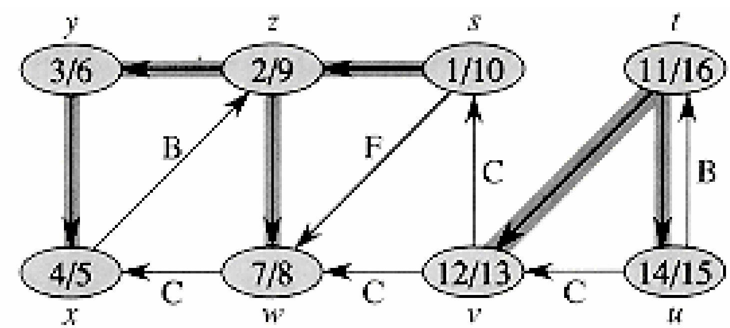
边分类算法

- 当(u, v)第一次被遍历, 考虑v的颜色
  - 白色, (u,v)为T边
  - 灰色, (u,v)为B边 (*只有它的祖先*是灰色)
  - 黑色: (u,v)为F边或C边. 此时需要进一步判断
    - $d[u]<d[v]$: F边 (v是u的后代, 因此为F边)
    - $d[u]>d[v]$: C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T边和B边 (易证)

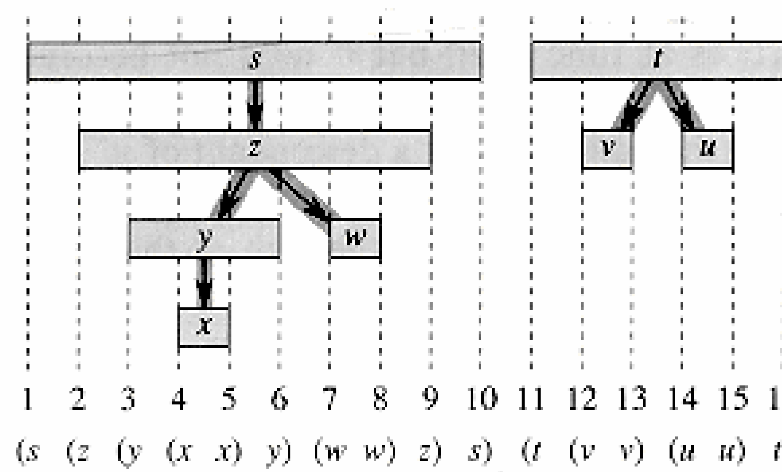
DFS树的性质演示

- 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

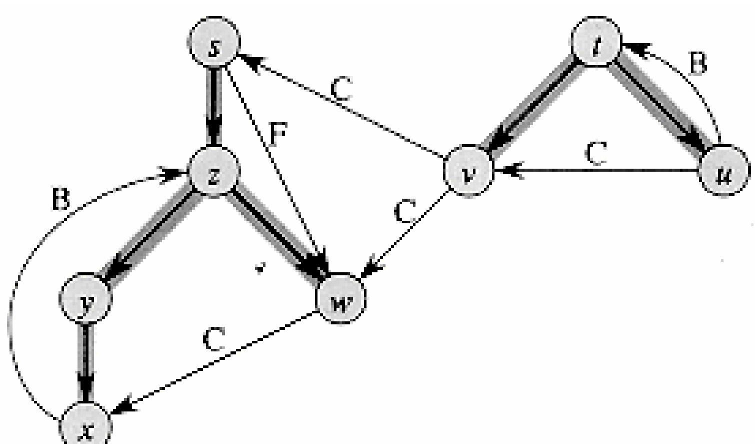
(a)



(b)



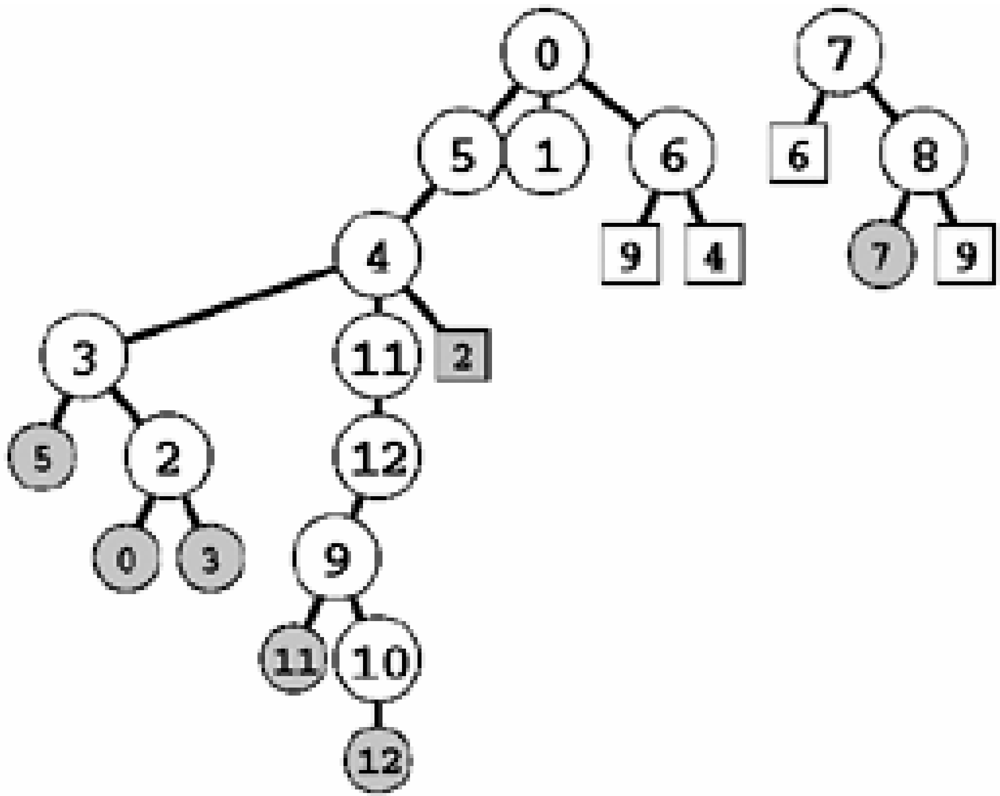
(c)

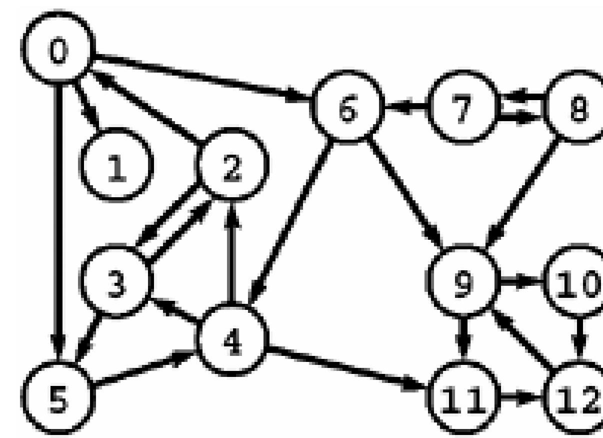


实现细节

- 颜色值以及时间戳可以省略, 用意义更加明确的pre数组和post代替d和f数组, 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] == -1) dfs(v); //树边, 递归遍历
  - else if (post[v] == -1) show(“B”); //后向边
  - else if (pre[v] > pre[u]) show(“F”); // 前向边
  - else show(“C”); // 交叉边
- pre和post的初值均为-1, 方便了判断

使用pre和post的DFS





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

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d[u]<d[v]<f[v]<f[u]
  - B边当且仅当d[v]<d[u]<f[u]<f[v]
  - C边当且仅当d[v]<f[v]<d[u]<f[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]

练习

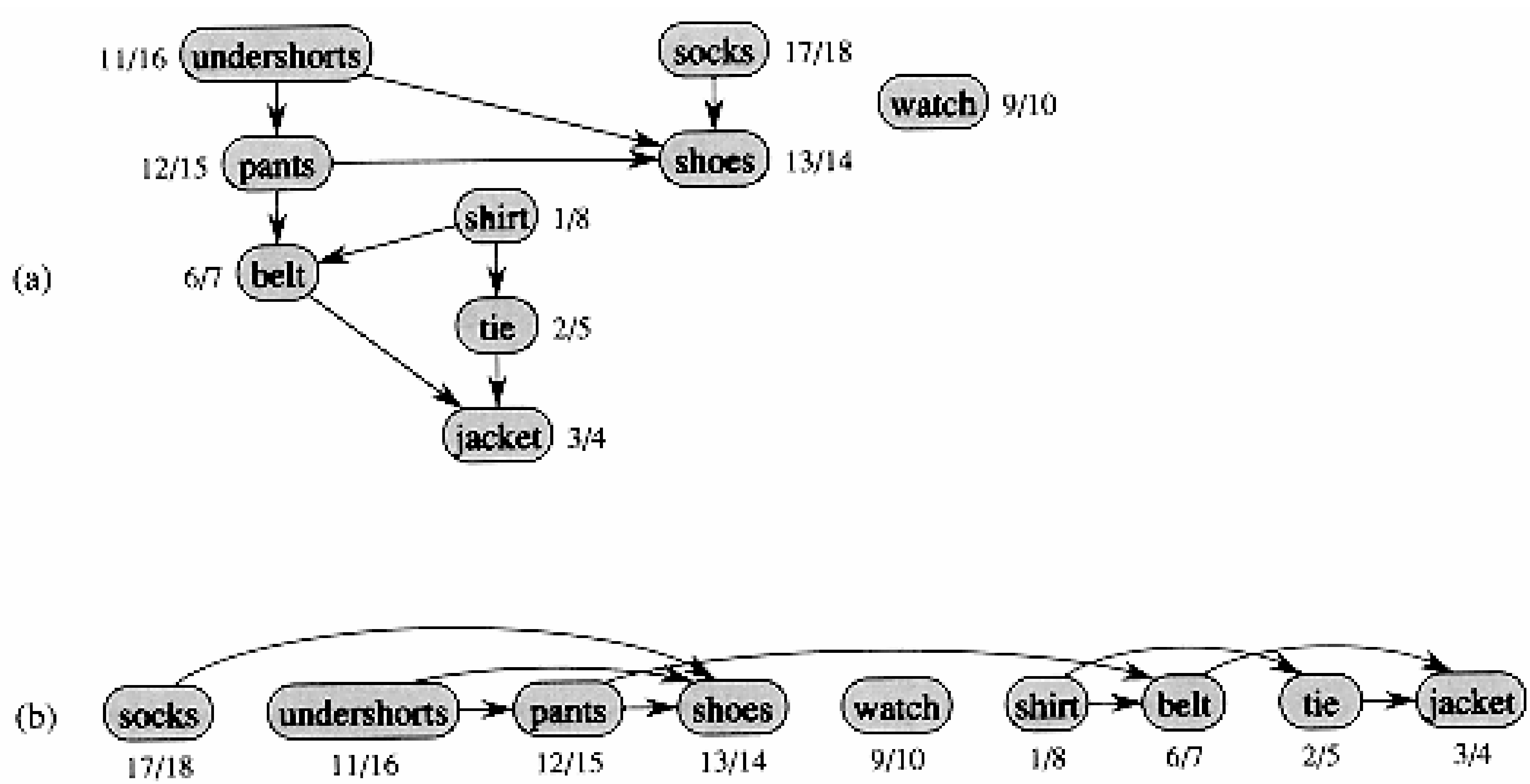
- 考虑无向图的边分类.
  - 证明只有T边和B边
  - 两次分类有可能是不一样的. 一种方案是只取第一次分类的结果, 另一种方案是按照类型优先级(T先于B). 证明两种方案等价
- 一个有向图是单连通的, 如果u可达v, 则u到v只有唯一一条简单路径. 设计一个判断一个图是否为单连通的算法

参考资料

- CLRS 22.3: Depth-first Search
- Algorithms in Java, Third Edition, 19.2: Anatomy of DFS in Digraphs

拓扑顺序

- 穿衣服的顺序



(a)
11/16 undershorts
socks 17/18
watch 9/10
12/15 pants
shoes 13/14
shirt 1/8
6/7 belt
tie 2/5
jacket 3/4
(b)
socks 17/18
undershorts 11/16
pants 12/15
shoes 13/14
watch 9/10
shirt 1/8
belt 6/7
tie 2/5
jacket 3/4

拓扑排序算法

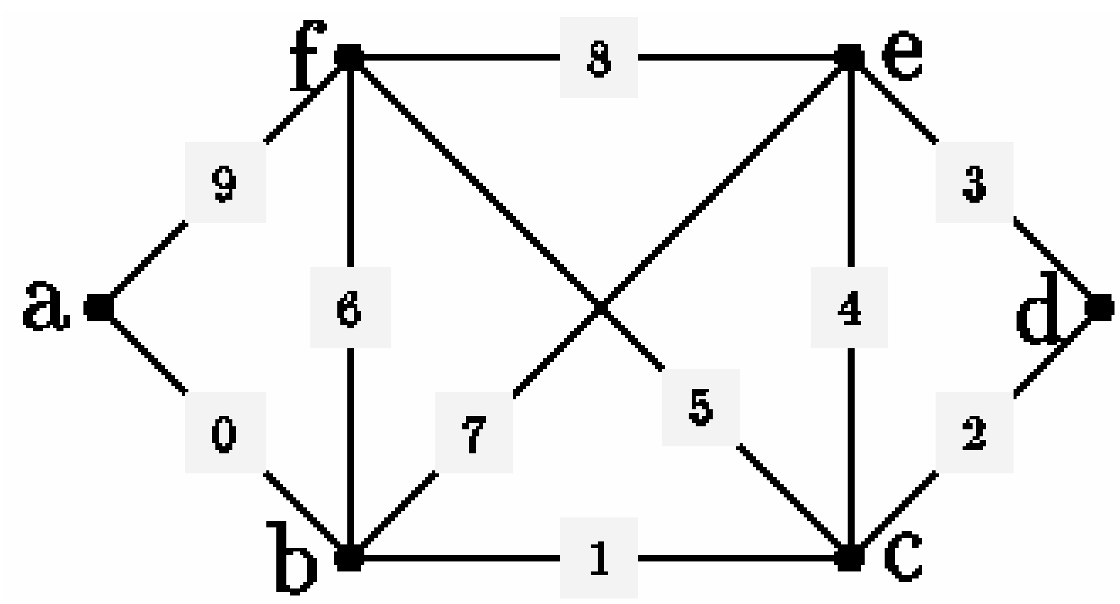
- 对图G使用DFS算法, 每次把一个结点变黑的同时加到链表首部
- 定理1: 有向图是DAG当且仅当没有B边
  - 如果有B边, 有环(易证)
  - 如果有环c, 考虑其中第一个被发现的结点v, 环中v的上一个结点为u, 则沿环的路径v→u是白色路径, 有白色路径定理, u是v的后代, 因此(u, v)是B边
- 定理2: 该算法正确的得到了一个拓扑顺序

练习

- 举例说明存在一个图G和它的一个拓扑顺序, 拓扑排序算法无法得到此序(不管结点排列顺序如何)
- 设计一个算法判断无向图是否含圈. 时间复杂度应为O(V), 和E无关
- 每次找0度点, 用队列设计一个O(V+E)时间的算法

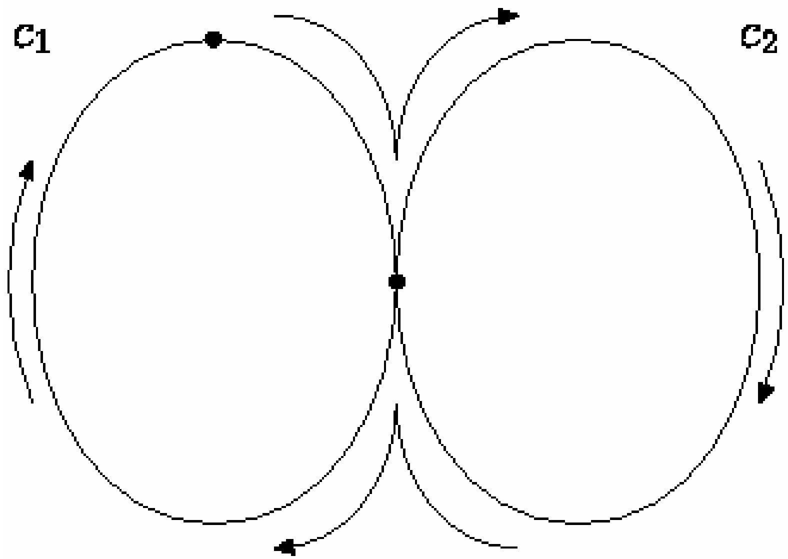
欧拉回路

- 如下图, 每条边经过一次且仅一次的称为欧拉回路(euler cycle, euler circuit).



欧拉回路

- 存在欧拉回路的充要条件: 每个点的度数都是偶数, 且图连通
- 算法: 套圈
  - 删除一个圈后, 剩下的图每个连通分量都有欧拉回路
  - 至少有一个公共点的圈可以合并



算法

- 算法: 找一个圈(用DFS), 然后把边上的边都删除, 求剩下各连通分量的欧拉回路, 最后合并起来

```
procedure Euler-Route(start);
begin
  for v ∈ adj(v) do
  if not marked(start,v) then begin
      mark_edge(start,v);
      mark_edge(v,start);
      Euler-Route(v);
      Result.Push(start,v);
  end
end
```

3 有向图：强连通分量和传递闭包

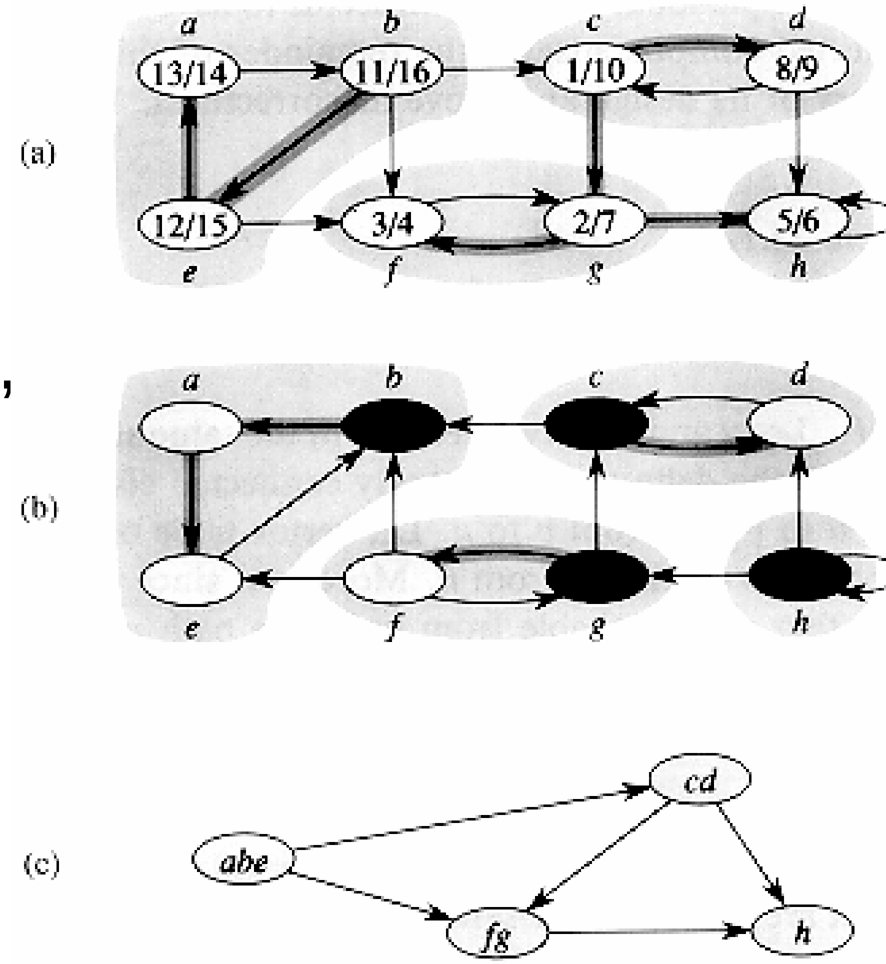
3.1 SCC的概念

3.2 Kosaraju算法

3.3 Tarjan和Gabow算法

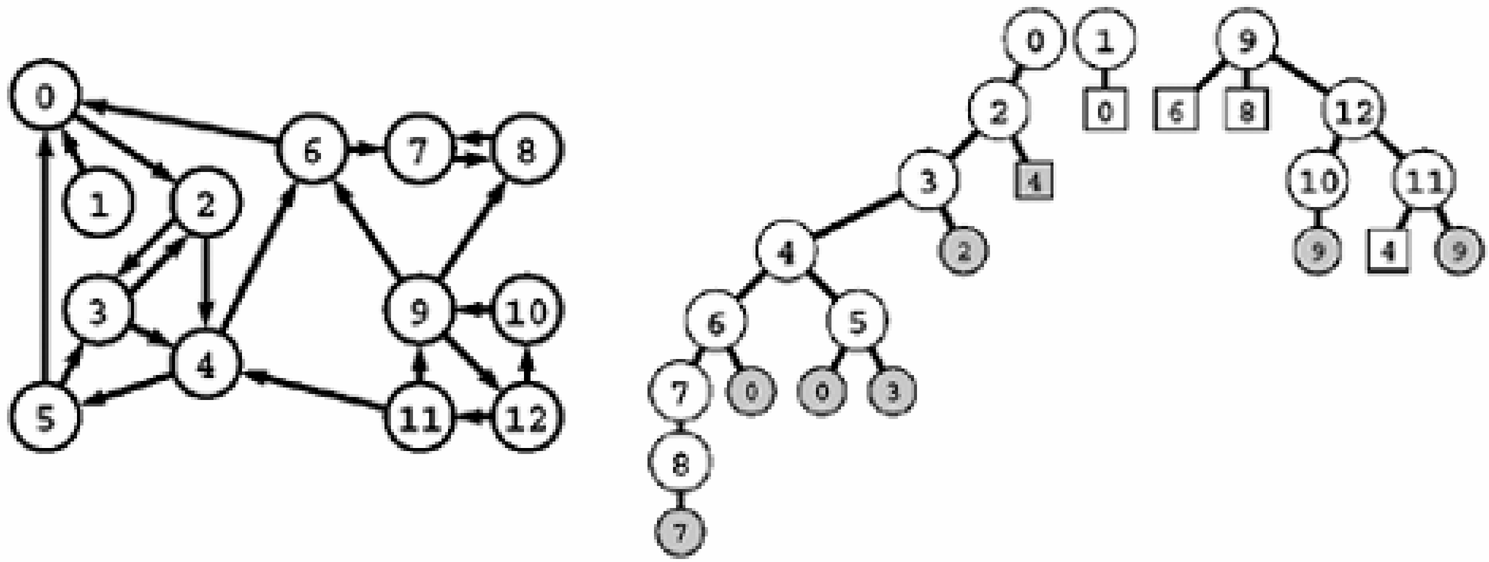
SCC的概念

- 有向图中, u可达v不一定意味着v可达u. 相互可达则属于同一个强连通分量(Strongly Connected Component, SCC)
- *<u>有向图和它的转置的强连通分量相同</u>*
- 所有SCC构成一个DAG

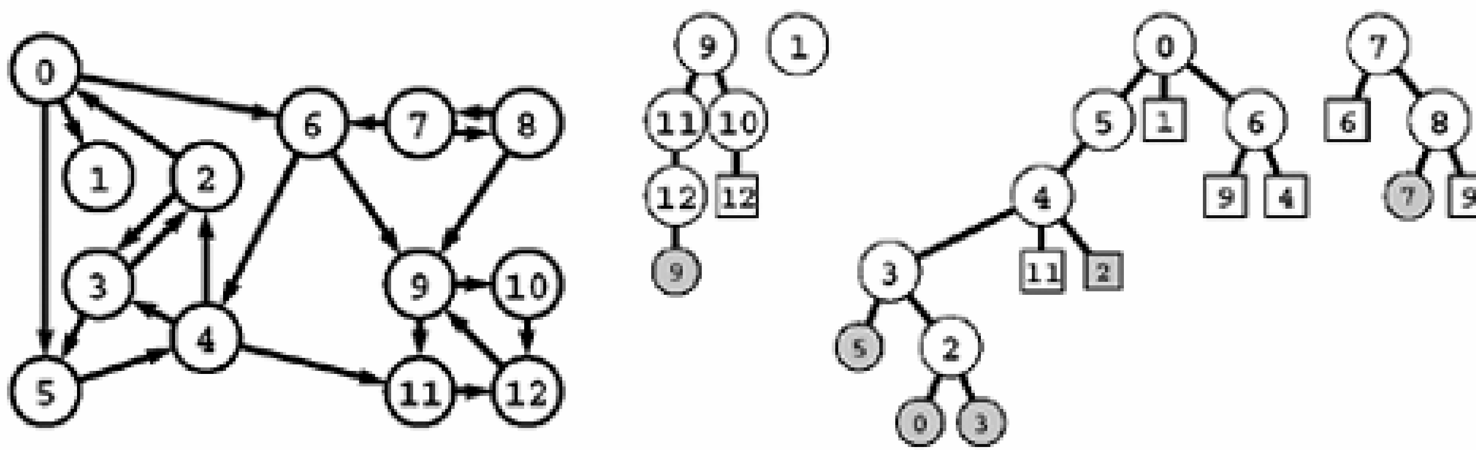


Kosaraju算法

- 算法步骤
  - 调用DFS(G), 计算出每个结点的f[u]
  - 计算G^T
  - 调用DFS(G^T), 在主循环中按照f[u]递减的顺序执行DFS-VISIT, 则得到的每个DFS树恰好对应于一个SCC
- 运行时间：O(n+m)
- 算法示例: 先把f[u]排序成postI数组, 然而在G^T上DFS



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



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

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

SCC图

- 把SCC看成点, 则DFS的同时可以得到SCC图. 在第二次DFS中, 每新开始一次DFS-VISIT, 即新找到一个SCC, 当前编号加1.
- DFS-VISIT某个SCC C时, 如果出现了指向一个已访问过的SCC C’的交叉边, 则在SCC图中设置边(C’, C), 因为在转置图中存在C到C’的边

局限性

- SCC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

基于DFS的SCC算法

- 结点放在栈S中
  - 当一个点结束后，它所在的SCC就访问完成了
  - 但是若任意点都可以输出SCC, 会引起重复
  - 我们把输出任务交给SCC中pre最小的一个
- 如何判断输出任务由谁完成呢?
  - 条件: 如果u或者u的后代存在一条反向边(w, v)到达u的祖先v, 那么u和父亲属于同一个SCC, 且由于pre[v]<pre[u], 任务不可能需要u完成
  - Tarjan算法: 使用LOW函数测试该条件
  - Gawbow算法: 使用另一个栈

Tarjan算法

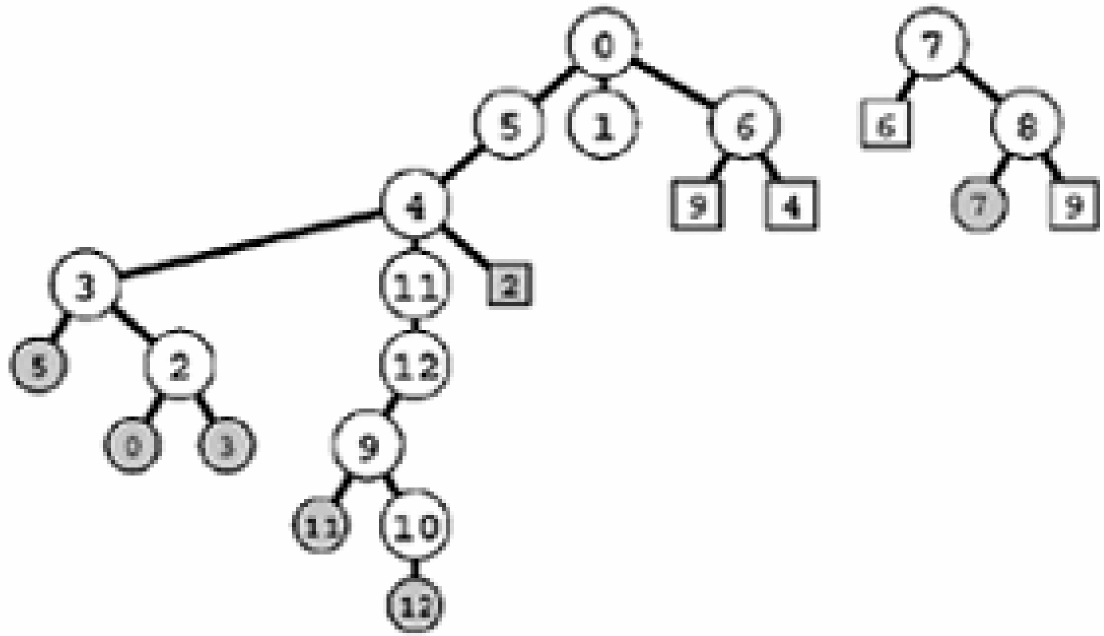
- Tarjan算法: 定义辅助函数low[u]为u及其的后代所能追溯到的最早(最先被发现)祖先点v的pre[v]值, 则可以在计算low的同时完成SCC计算
- 检查完儿子和检查反向边时更新low函数, 交叉边可以通过特殊手段避免

LOW函数的计算

```
min = low[u] = pre[u] = cnt++; //初始为pre[u]
S.push(u);
for u的邻居v {   //计算min
    if (pre[v] == -1) dfs-visit(v);
    if (low[v] < min) min = low[v];
}
if (min < low[u]) { low[u] = min; return; }  //任务不用u完成
do { id[v = S.pop()] = scnt; low[v] = n; } while (v != u);
scnt++;
```

- 注意: 输出SCC后需要设置low值均为最大值n以防止交叉边影响结果

LOW函数的计算



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

Gabow算法

- Gabow算法使用另一个栈P保留当前路径中的结点, 发现反向边(u,v)后不断出栈, 只保留v

```
pre[u] = cnt++;
S.push(u); P.push(u);
For u的每个邻居v
    if (pre[v] == -1) dfs-visit(v);
    else if (id[v] == -1) while (pre[P.top()] > pre[v]) P.pop();
if (P.top() == u) P.pop(); else return;
do { id[v = S.pop()] = scnt; } while (v != u); scnt++;
```

Tarjan和Gabow算法过程

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

应用: 传递闭包

- 对于图G的任意一个结点u, u可达的结点集合称为u的传递闭包
- 无向图: u的传递闭包为和u处于同一连通分量的点集
- 有向图: 先求SCC图, 则u的传递闭包为u所处SCC和它的所有后代SCC中的结点

参考资料

- CLRS 22.5: Strongly connected components
- Algorithms in Java, Third Edition, 19.8: Strong Components in Digraphs

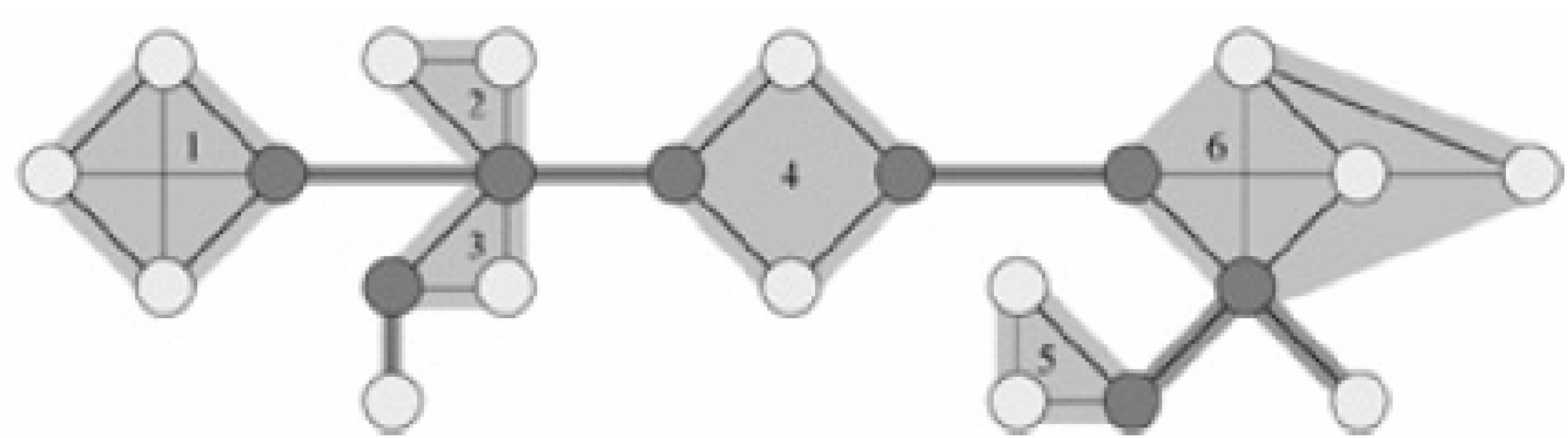
4 无向图：割顶、桥和双连通分量

4.1 割顶和桥

4.2 连通性和双连通分量

割顶和桥

- 对于无向连通图G
  - 割顶是去掉以后让图不连通的点
  - 桥是去掉以后让图不连通的边



无向图的LOW函数

- 定义辅助函数low[u]为u及其的后代所能追溯到的最早(最先被发现)祖先点v的pre[v]值
- 类似有向图的计算方式, 注意无向图只有T和B边

```
low[u] = pre[u] = cnt++;
for u的不等于u的邻居v{   //不考虑自环
    if (pre[v] == -1) {  // 白色点
        dfs-visit(v);
        if (low[u] > low[v]) low[u] = low[v];
    } else
        if (low[u] > pre[v]) low[u] = pre[v];
}
```

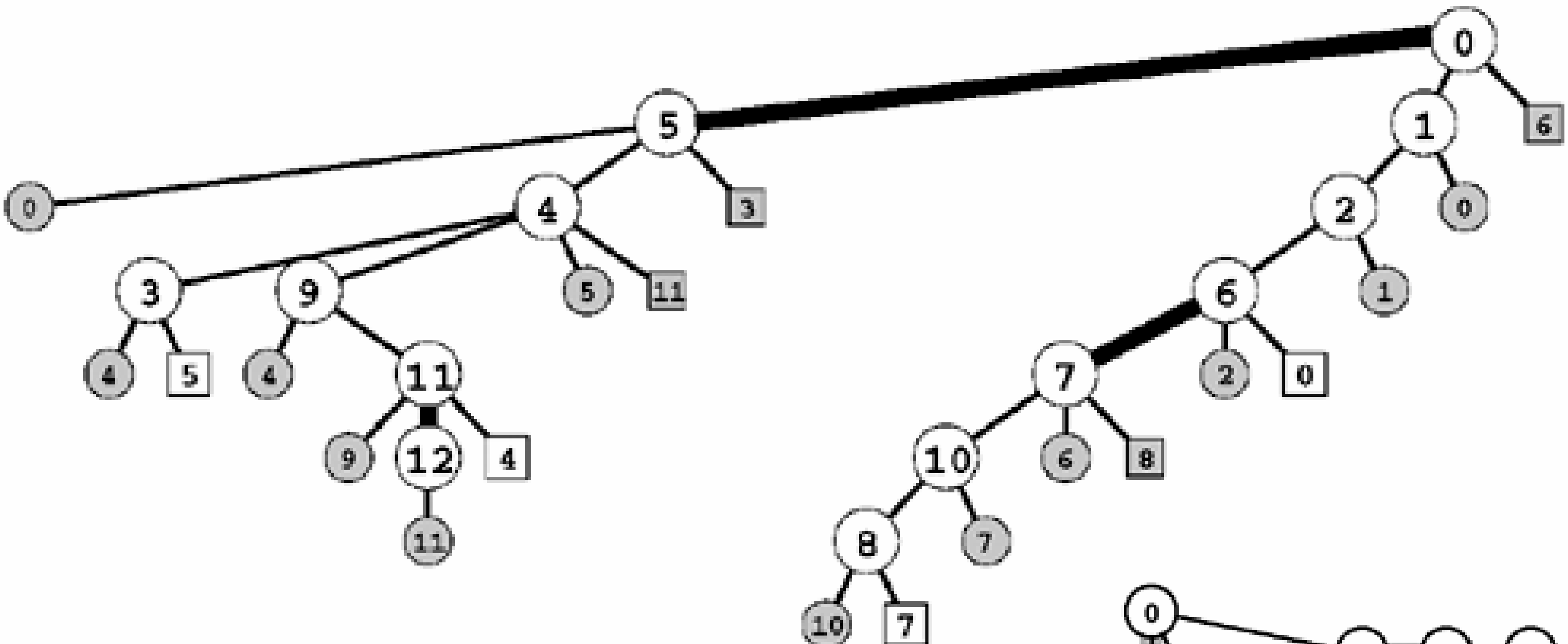
割顶的判定

- 在一棵DFS树中
  - 根root是割顶当且仅当它至少有两个儿子
  - 其他点v是割顶当且仅当它有一个儿子u, 从u或者u的后代出发没有指向v祖先(不含v)的B边, 则<u>*删除**v**以后**u**和**v**的父亲不连通*</u>, 故为割顶
- 割顶判定算法:
  - 对于DFS树根, 判断度数是否大于1
  - 对于其他点u, <u>*如果不是根的直接儿子*</u>, 且LOW[u] >= d[P[u]], 则它的父亲v=P[u]是割点

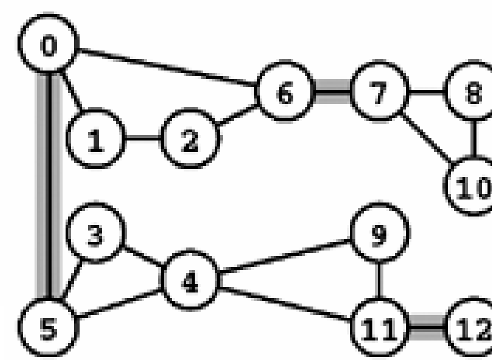
桥的判定

- 边(u,v)为桥当且仅当它不在任何一个简单回路中
- 发现T边(u,v)时若发现v和它的后代不存在一条连接u或其祖先的B边, 则<u>*删除***(u,v)***后***u***和***v***不连通*</u>, 因此(u,v)为桥
- 桥的判定算法: 发现T边(u, v)时若LOW[v]>d[u](注意<u>*不能取等号*</u>), 则(u,v)为桥
- 实现: 用pre数组代替d数组, 取消f数组. 实际代码是测试若low[v]=pre[v]则(u,v)是桥

桥判定举例



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



参考资料

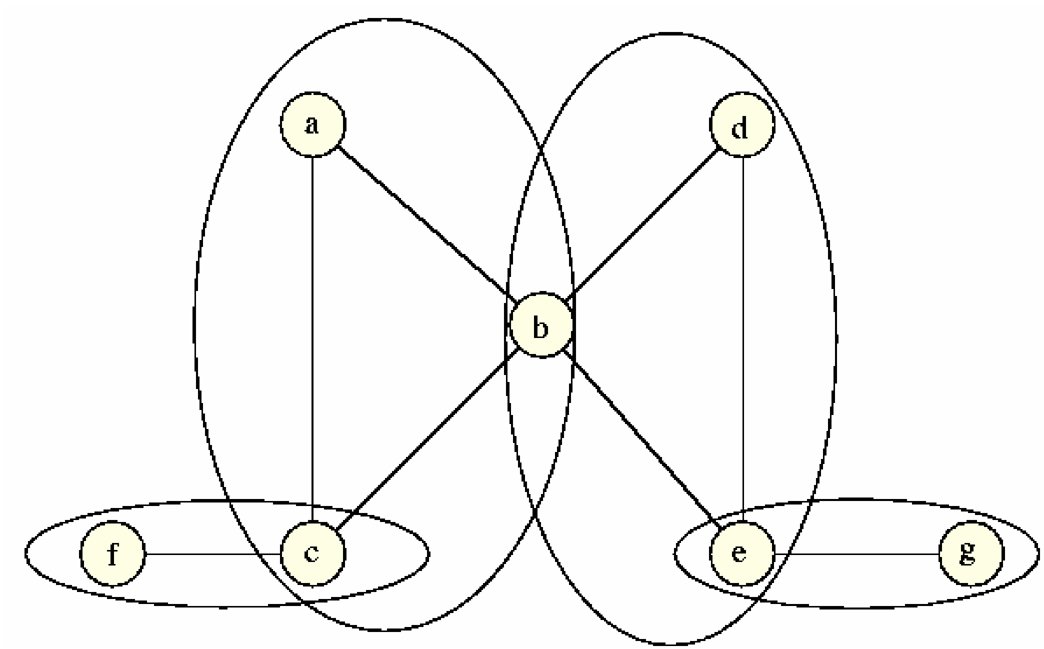
- Algorithms in Java, Third Edition, 18.6: Separability and Biconnectivity

连通性定义

- 点连通度 (等价性: Whitney定理)
  - 定义1: 把图变非连通所需删除的最少点数
    - 1-连通: 一般连通
    - 2-连通: 双连通, 删除一个点后仍连通(无割顶)
  - 定义2: 任意两个点至少有k个不含相同结点的路(vertex-disjoint path).
- 边连通度 (等价性: Menger定理)
  - 定义1: 把图变非连通所需删除的最小边数
  - 定义2: 任意两个点至少有k个不含相同边的路(edge-disjoint path).

BCC

- 一个图的块(biconnected component, bcc)是双连通的极大子图
- 块间没有公共边, 以割顶相连



BCC的性质

- 性质1. 每条边都包含在某个BCC中. 证明: 对于(u, v), {u, v}是双连通的, 这是某BCC的一部分
- 性质2. 不同的两个BCC最多有一个公共结点, 此结点是原图的割顶
- 性质3. 不同的两个BCC不含公共边. 证明: 如果有相同边, 则含有两个公共结点.
- 根据性质2和性质3可知: BCC是G的边划分

BCC算法

- 对于点v, 考虑v的父亲u
  - u是根, (u,v)是新BCC的第一条边
  - 否则设u的父亲为f. 若删除u后, v和f不连通, 则{f, u, v}不是双连通的, 因此(u, v)是新BCC的第一条边, 否则(u, v)和(f, u)处于同一个BCC中
- 实现: 可以用栈, 类似于SCC的Tarjan算法
- 注意: 自环没有编号, 因为在无向图中自环被忽略

收缩图

- 把每个BCC看成点b, 每个割顶a也看成点. 则b和a相连当且仅当b包含a. 这样得到的图叫BCC收缩图
- BCC收缩图是一棵树

其他算法

- 修改的Gabow算法可以计算无向图的桥、割顶、块和边连通分量
- 最大流可以用来计算任意图的点连通度和边连通度