一、图的表示
1、邻接矩阵Adjacency
2、邻接表
二、广度优先遍历BFS
1、连通图--FIFO三染色
2色尝试
三染色--白灰黑
拓展性
正确性
2、不连通图
3、最短路径
三、深度优先搜索DFS
迭代
递归
不连通
递归--三染色建树
初始化结点
深搜建树
美妙性质
1、边的分类
2、括号化定理
3、白色路径定理

4、点颜色推断边
四、拓扑排序
判断无环
算法1
算法2：
五、强连通分量
分量图--有向无环图DAG
图转置
Tarjan's SCC算法
定位Ci的root ri

关联矩阵incidence matrix
有向图directed graph : the edge j leave/enter vertice i
联通无圈图connected acyclic graph
点vertice
度degree

一、图的表示

```
G=(V,E) |V|=n  |E|=m
```

1、邻接矩阵Adjacency

> 稠密图常用，小图常用
> 需要快速判断相邻时，最短路径算法时
> 优势：每个记录项只需要1的空间

- 无向图：

> 对称，主对角线为零
>
> 空间n*(n-1）/2

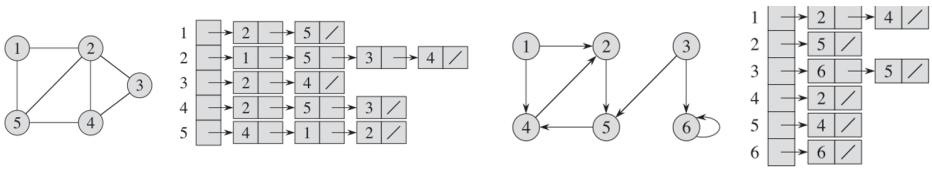
- 有向图

> 空间nn

```
判断相邻：快
找任意邻居：慢
枚举enumerate所有邻居：慢
```

2、邻接表

> 稀疏图常用：m<<nn，边很少
>
> 鲁棒性高robust



- 无向图

> 所有边出现两次，所以链表长度和为2m

- 有向图

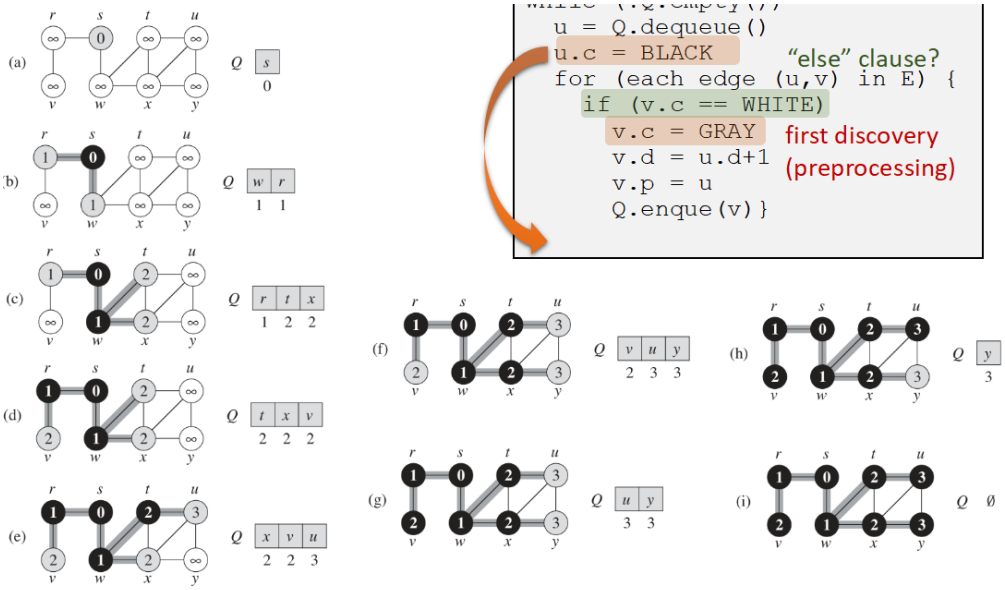
> 所有链表长度和为m

- 空间Θ（n+m）

```
判断相邻：快
判断x，y邻居：慢
枚举enumerate所有邻居：快
```

二、广度优先遍历BFS

> 按与source的距离，一层一层来



1、连通图--FIFO三染色

2色尝试

> dist计算距离
>
> 循环从队列头拿出元素：访问，遍历邻居加距离

BFSSkeleton(G,s):

```
for (each u in V)
  u.dist=INF, u.visited=false
s.dist = 0
Q.enque(s)
while (!Q.empty())
  u = Q.dequeue()
  u.visited = true
  for (each edge (u,v) in E)
    if (!v.visited)
      v.dist = u.dist+1
      Q.enque(v)
```

- 漏洞：结点的两个邻居都访问过时，此结点会被重复添加到Q

三染色--白灰黑

> 光visited不够，还需要一个discovered（灰）

```
1 白：不在队列
2 灰：放到了队尾，但未访问
3 黑：已变成队头，被拿出访问
```

> .p记录当前点到s的最短路径上的父亲

```
BFS(G,s):
for (each u in V)
  u.c=WHITE, u.d=INF, u.p=NIL
s.c=GRAY, s.d=0, s.p=NIL
FIFOQueue Q
Q.enque(s)
while (!Q.empty())
  u = Q.dequeue()
  u.c = BLACK
  for (each edge (u,v) in E)
    if (v.c == WHITE)
      v.c = GRAY
      v.d = u.d+1
      v.p = u
      Q.enque(v)
```

拓展性

```
u.c = BLACK访问结点时刻，可拓展；
              此句可移到最后面
v.c = GRAY第一次发现结点时刻，可拓展
```

- 时间Θ（n+m）

> while执行n次
> for执行2m次

- 不是连通图，要写成O（n+m）

正确性

- 法一：对距离k归纳，证明BFS访问的一一对应联通的点
- 法二：循环不变式：Q中永远是灰点集合

2、不连通图

> 对每个连通分支执行

```
BFS(G):
for (each u in V)
  u.c = WHITE, u.d = INF, u.p = NIL
for (each u in V)
  if (u.c == WHITE)//此处循环判断新连通分支
    u.c = GRAY, u.d = 0, u.p = NIL
    Q.enque(u)
    while (!Q.empty())
      v = Q.dequeue()
      v.c = BLACK
      for (each edge (v,w) in E)
        if (w.c == WHITE)
          w.c = GRAY
          w.d = v.d+1
          w.p = v
          Q.enque(w)
```

3、最短路径

> BFS可以找出最短路径

- 相邻点的路径长度关系

引理 22.1 给定 $G=(V,E)$，G 为一个有向图或无向图，设 $s\in V$ 为任意结点，则对于任意边 $(u,v)\in E$，$\delta(s,v)\leqslant\delta(s,u)+1$。

三、深度优先搜索DFS

> 访问邻居直到stuck
> 回退直到有未访问邻居

> 两次处理结点机会：白变灰，灰变黑
> 这些机会可用于统计已有结点数

迭代

> 深搜和栈惺惺相惜
> push时不访问，pop时才访问
> 你添加点的样子像一个BFS

```
DFSIterSkeleton(G,s):
Stack Q
Q.push(s)
while (!Q.empty())
  u = Q.pop()
  if (!u.visited)
    u.visited = true
    for (each edge (u,v) in E)
      Q.push(v)
```

递归

> 深搜和递归简直绝配

```
DFSSkeleton(G,s):
s.visited = true
for (each edge (s,v) in E)
  if (!v.visited)
    DFSSkelecton(G,v)
```

不连通

```
DFSAll(G):
for (each node u)
  u.visited = false
for (each node u)
  if (u.visited == false)
    DFSSkeleton(G,u)
```

递归--三染色建树

```
白：未调用DFSSkeleton(G,u)
灰：调用中DFSSkeleton(G,u)
黑：DFSSkeleton(G,u)已返回
```

初始化结点

> 初始颜色白，父节点为空
> 遍历调用建树操作

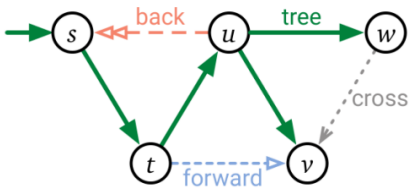
```
DFSAll(G):
for (each node u)
  u.color = WHITE, u.parent = NIL
for (each node u)
  if (u.color == WHITE)
    DFS(G,u)
```

深搜建树

```
DFS(G,s):
s.color = GRAY
for (each edge (s,v) in E)
  if (v.color == WHITE)
    v.parent = s
    DFS(G,v)
s.color = BLACK//深搜特点：变黑之前，孩子一定全黑
```

美妙性质

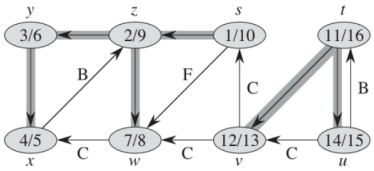
1、边的分类

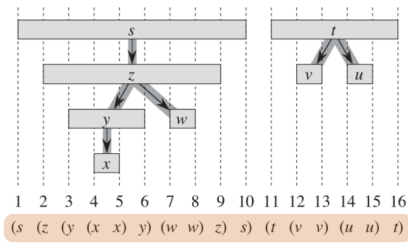



```
树边
后向边：指向祖先
前向边：指向后代
横向边：其余的：兄弟、不同树之间
```

2、括号化定理

- 用时间戳画出的括号匹配





- 覆盖性质

- 区间[u.d，u.f]和区间[v.d，v.f]完全分离，在深度优先森林中，结点u不是结点v的后代，结点v也不是结点u的后代。
- 区间[u.d，u.f]完全包含在区间[v.d，v.f]内，在深度优先树中，结点u是结点v的后代。
- 区间[v.d，v.f]完全包含在区间[u.d，u.f]内，在深度优先树中，结点v是结点u的后代。

- v是u的后代当且仅当u.d<v.d<v.f<u.f

3、白色路径定理

- v是u的后代当且仅当发现u（u.d）时，存在一条由u到v的白色路径

4、点颜色推断边

- 有向图

> 第一次探索边（u，v）时

```
v白：树边
```

```
2 v灰：后向边
3 v黑：前向边，横向边
```

- 无向图

> 只有树边、后向边

四、拓扑排序

Directed Acyclic Graphs (DAG)

> 有向无圈图DAG的线性排序
> 有圈，做不到
> 可能很多种

判断无环

- 有向图无环《==》深搜无后向边

> 证明用白色路径

- （u，v）则DFS中u.f>v.f

算法1

- DFS，计算finish time
- 结点finish时加到list表头
- 无back edge则成功

算法2：

- 找到一个源点并删除，删除所有出边

> 找到一个source，输出并删除后，必有新的source出现

- 重复直到空

- 源点source：入度为0

> 可当作第一个结点
> max finish time 必是

- 汇点sink：出度为1

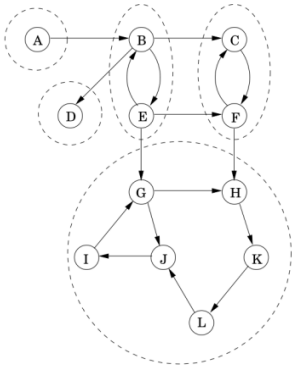
> min finish time 必是

- 每个DAG至少有一个源点和汇点

五、强连通分量

strongly connected components (SCC)

> 有向图点集的子集，此子集内任意u,v，存在u->v，v->u
> 双向路径才算联通



分量图--有向无环图DAG

component graph

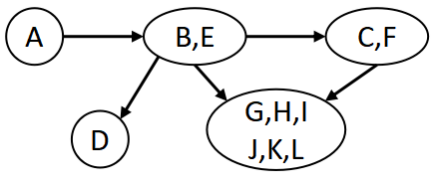
- G^C=(V^C,E^C)

> 分量之间单向相连，一定可以表示

- 深搜实际是以拓扑排序次序访问分量图结点

> 这里D和GHIJKL都属于sinkSCC

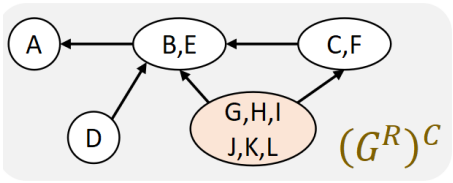
- 存在 (vi,vj)∈E^C 说明存在(u,v)∈E满足u∈Ci and v∈Cj

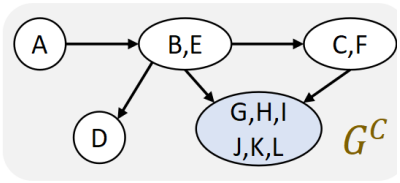


- d（U）f（U）：结点集合U中最早发现时间、最晚完成时间

图转置

- 方向相反，SCC不变





- 在GR中max finish time的是source

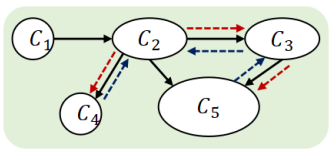
Lemma: for any edge $(u, v) \in E(G^R)$, if $u \in C_i$ and $v \in C_j$, then $\max_{u \in C_i}\{u.f\} > \max_{v \in C_j}\{v.f\}$.

Tarjan's SCC算法

- 调用的时候这样写

Find strongly connected components $C_1, C_2, \ldots, C_k$ and component graph $G^C = (V^C, E^C)$ with tarjan's SCC algorithm in linear time;

For each SCC C_i, let r_i be its root. If we push a node to a stack when it is discovered, when DFS returns from r_i, all nodes above r_i in the stack are in C_i.



- First node in C_2 (root of C_2)
- Some nodes in C_2
- First node in C_3 (root of C_3)
- Some nodes in C_3
- First nodes in C_5 (root of C_5)
- All other nodes in C_5 (C_5 is a sink SCC)
- All other nodes in C_3 (C_3 becomes a sink SCC by then)
- Some nodes in C_2
- First nodes in C_4 (root of C_4)
- All other nodes in C_4 (C_4 is a sink SCC)
- All other nodes in C_2 (C_2 becomes a sink SCC by then)
- First node in C_1 (root of C_1)
- All other nodes in C_1 (C_1 becomes a sink SCC by then)

- sink SCC：从这里出发的，一定能回来，因为这个SCC没有出度

> 这里C4和C5都属于sinkSCC
>
> 当DFS开始遍历C3中所有其他结点时，C3变成sinkSCC

定位Ci的root ri

- Cv：v所在的SCC
- low（v）:Cv中min发现时间
- v是root《==》low（v)=v.d

```
Tarjan(G):
time = 0
Let S be a stack
for (each node v)
  v.root = NIL
  v.visited = false
for (each node v)
  if (!v.visited)
    TarjanDFS(v)
```

```
TarjanDFS(v):
v.visited = true, time = time+1
v.d = time, v.low = v.d
S.push(v)
for (each edge (v,w))
  if (!w.visited) // tree edge
    TarjanDFS(w)
    v.low = min(v.low, w.low)
  else if (w.root == NIL) // non tree edge in C_v
    v.low = min(v.low, w.d)
if (v.low == v.d)
  repeat
    w = S.pop(), w.root = v
  until (w==v)
```