第五章 图问题与基本周游方法

主要参考：

14. Sara Baase, Allen Van Gelder, “Computer Algorithms: Introduction to Design and Analysis (Third Edition)”, Higher Education Press & Pearson Education Asia Limited., 2001. ***Chapter 7***

1．引言

很多问题可归结为关于图的问题。这些问题不只是根源于计算机领域，也贯穿了各个学科与工商业界。针对许多图问题已发展出高效算法，这极大地提高了人们解决实际问题的能力。然而，还有许多重要的图问题，迄今仍无有效方法求解。对于另外一些问题，则拿不准现存的求解方法已经是最有效的还是可有进一步改进的余地。

在本章，我们首先回顾图的定义和一些基本性质，然后详述有效周游图的基本方法，即深度优先检索法和宽度优先检索法。以这两种检索法的算法为基础，可扩展出关于许多典型图问题的高效算法，所谓高效即在线性时间内可求解。粗略地说，这样的图问题称为“易解”的图问题，易解的意思不是说容易编制算法或程序，而是指编制的算法可以非常高效地求解问题，在实践中可解决很大规模的实例，比方说，可处理有数百万结点的图。

依此粗略观点来看，中等难度图问题包括那些可在多项式时间内

求解的图问题，比如最优检索树问题等，这些问题虽然不能在线性时间内求解，但它们的时间复杂度往往受限于n^2或n^3这样的渐近增长率，在实际中可解决成千上万之规模的实例，也可令人接受。“困难”的图问题，是没找到多项式时间算法的问题，这样的问题就实际应用来说几乎是不可求解的，因为很小规模的实例就可能耗费几年时间。就目前所知，看不到发现有效算法的希望，但还无人证明确实不存在多项式算法。它们被称为 NP 难度的问题。

图问题的一大诱人之处在于稍微改变问题的提法就会导致难度类型的变化。因此熟悉现存问题的难度及其原因将有助于处理碰到的新问题。

2．图的概念和数据结构表示

2.1 图的基本概念和术语

大凡牵涉到研究离散对象间某种结构的问题总能用图模型来表示。譬如，交通运输线路图，程序流程图，计算机网络，电子线路等等，对应于这些结构的问题就转化为典型的图问题。

抽象地说，图为离散对象集合上的二元关系提供了一种描述方法。

图(graph)分为无向图(undirected graph)和有向图(directed graph, digraph)。任何图由二要素组成：结点集V和边集E。$G=(V, E)$。无向图与有向图的区别是，前者的边是无向边，后者的边是有向边。无向图表示了一种**对称的**二元关系，在大多数情况下可看成是一种特殊的有向图：**对称有向图**, 它是将无向图的每条边换

成两条方向相反的有向边得到的。

有限简单图：包含有限个结点，不含平行边和自环的图。结点集

$$V=\{v_1,v_2,\cdots,v_n\}$$

边集

$$E=\{e_1,e_2,\cdots,e_m\}$$

任何边可表示为$e_k=v_iv_j$，其中v_i与v_j是边e_k的端点，说v_i与v_j相邻，边e_k与v_i和v_j相关联；在有向图中，v_i是起点（tail），v_j是终点（head）， 说v_i是v_j的内邻点，v_j是v_i的外邻点。

图的其他重要概念：子图，路径，道路，回路，连通图(无向图)与**连通分支**, 强连通图(有向图)与**强连通分支**， **反圈图**， **赋权图**。

连通关系：无向图G的结点集V上的一个二元关系，称结点u与v是连通的，如果它们之间存在道路。连通关系是等价关系。若无向图G的任何两个结点都是连通的，则称G是连通图。若G的子图G_1是连通图，则称G_1是G的连通子图。

连通分支：无向图的结点集上连通关系的等价类；也是无向图的极大连通子图。

强连通关系：有向图G的结点集V上的一个二元关系。称结点u与v是强连通的，如果G中既存在u到v的道路，又存在v到u的道路。强连通关系是等价关系。若有向图G的任何两个结点都是连通的，则称G是强连通图。 若G的子图G_1是强连通图，则称G_1是G的强连通子图。

强连通分支：有向图的结点集上强连通关系的等价类，是有向图

的极大强连通子图。

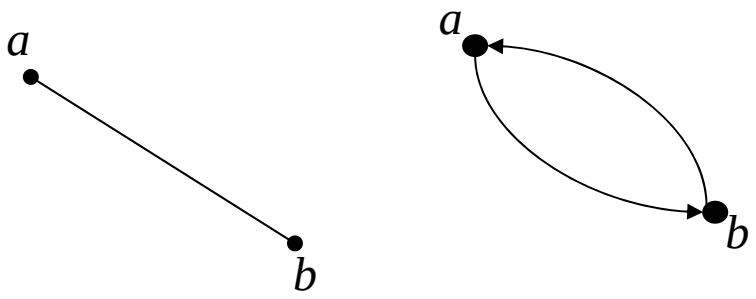
反圈图（Acyclic Graph）：不包含回路的图。

无向反圈图（Undirected Acyclic Graph）：称为无向森林（Undirected forest），如果还是连通的，称为无向树（Undirected tree）。

有向反圈图（Directed Acyclic Graph, 简记为 DAG）：不含回路的有向图。

转置图：设G是一个有向图，将它的每条边的方向改变为相反方向，就得到它的转置图G^t。

注：在涉及无向图的问题中，如果回路起重要作用，则不可将其看作对称有向图。譬如无向图ab没回路，但它对应的对称有向图有两条有向边：ab和ba，形成一个回路。



赋权图：三元组(V, E, W)，其中，W是定义在边集E上的实值（有些情况下，限制为整数、有理数是适当的）函数。权函数可有多种实际含义。

2.1 图的数据结构表示：

邻接矩阵

n阶图G可被如下定义的n阶方阵$A=(a_{ij})$所表示：

$$a_{ij}=\begin{cases}1 & \text{if } v_iv_j\in E\\ 0 & \text{else}\end{cases}\qquad 1\le i,j\le n$$

对无向图来说，邻接矩阵是对称的，只有上三角（或下三角）是必须存储的。如果$G=(V,E,W)$是赋权图，可以修改邻接矩阵使之表示出权来。

$$a_{ij}=\begin{cases}w(v_iv_j) & \text{if } v_iv_j\in E\\ c & \text{else}\end{cases}\qquad 1\le i,j\le n$$

其中c是常数，依赖于权的实际含义和待求问题的提法。比如说，如果权是路程花费，则可取$c=\infty$；如果权是网络线路的容量，则可取$c=0$.

有些问题需要处理每一条边，如果用邻接矩阵来表示图，需验证每一可能的边，可能的边数为$n(n-1)$（有向图）或$n(n-1)/2$（无向图），所以算法复杂度的下界是n^2。

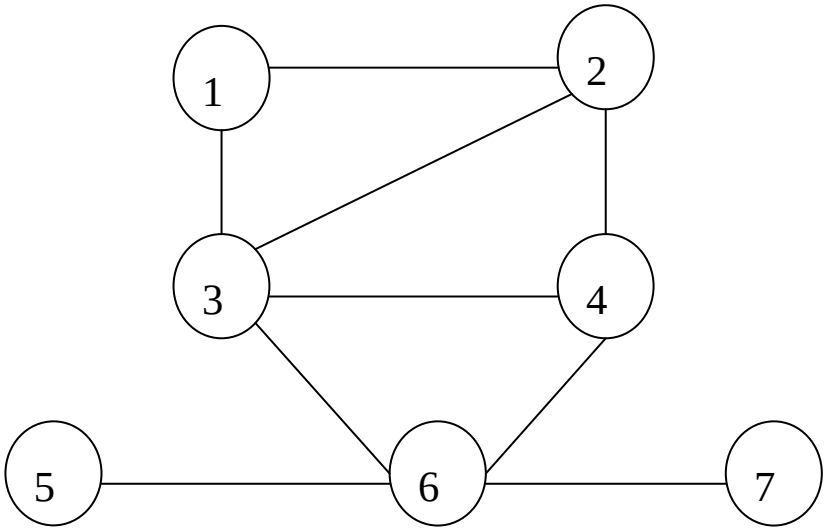
邻接表

n个链表，每个结点v_i对应一个链表i，表示结点v_i的所有外邻点与关联的边，每个链表有一个头指针$H(i)$，它指向表示v_i的第一个外邻点的链结点，每个链结点包含若干个域，其中有两个域分别表

示外邻点和指向下一个外邻点的指针， 其他域可表示对应边的信息，例如边的权值等。

邻接表是邻接矩阵的压缩表示，链结点的个数为图的边数m（有向图）或$2m$（无向图）。对很多问题的求解来说，可提高运算效率，有可能使算法时间复杂度不超过$O(n+m)$。

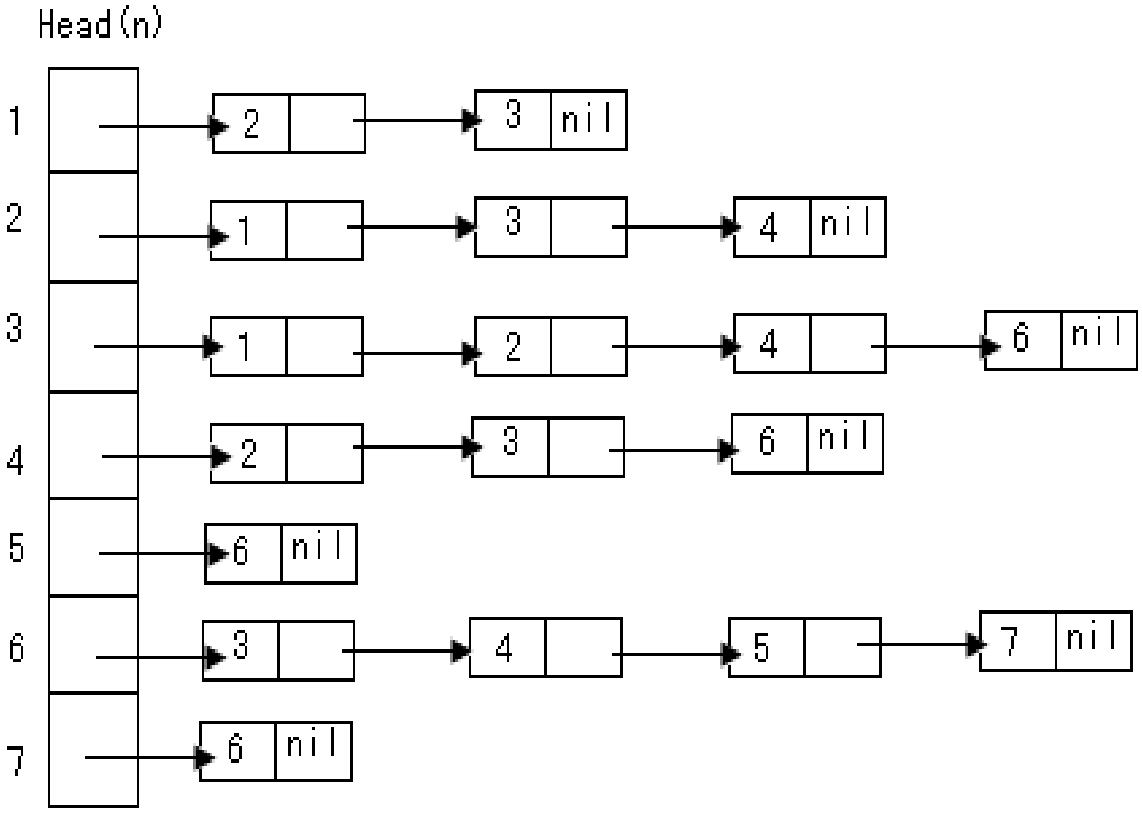
举例：



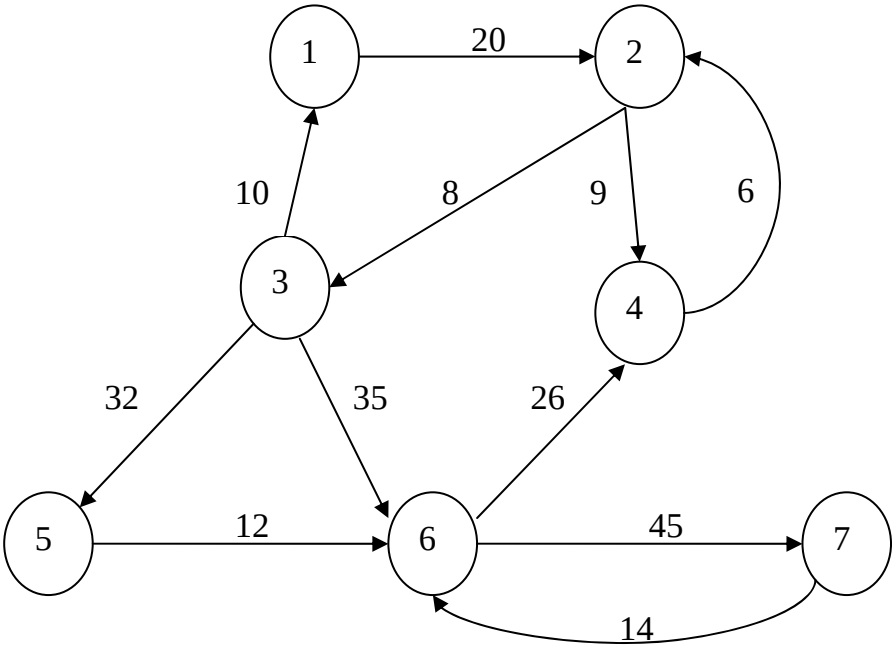
无向图示例

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

邻接矩阵


Head(n)
1
2
3
4
5
6
7
2
3
nil
1
3
4
nil
1
2
4
6
nil
2
3
6
nil
6
nil
3
4
5
7
nil
6
nil

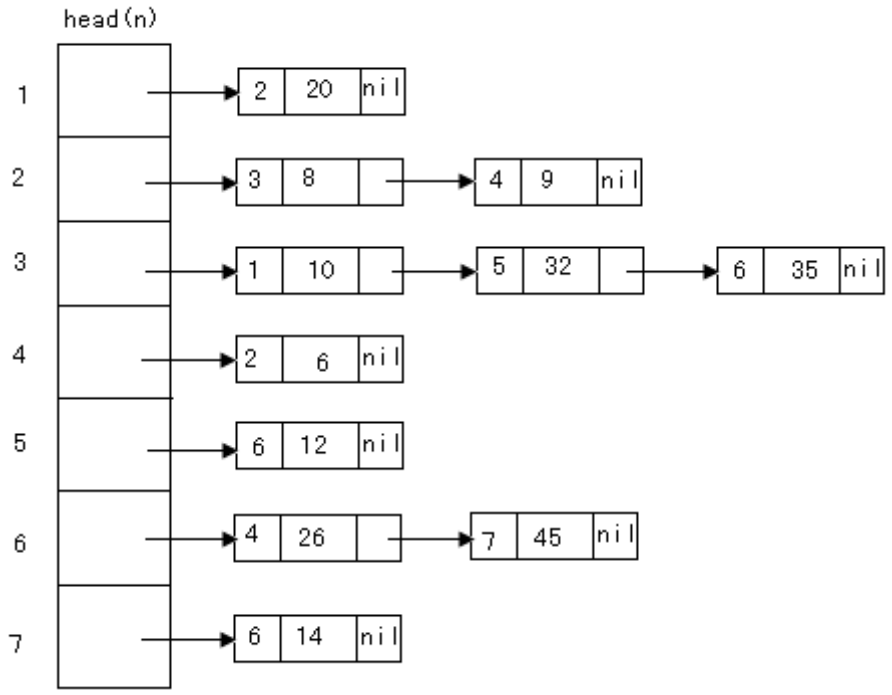
邻接表


1
2
3
4
5
6
7
20
10
8
9
6
32
35
26
12
45
14

赋权有向图示例

$$\begin{pmatrix} 0 & 20 & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 8 & 9 & \infty & \infty & \infty \\ 10 & \infty & 0 & \infty & 32 & 35 & \infty \\ 6 & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & 12 & \infty \\ \infty & \infty & \infty & 26 & \infty & 0 & 45 \\ \infty & \infty & \infty & \infty & \infty & 14 & 0 \end{pmatrix}$$

赋权矩阵



赋权图邻接表

3. 图的基本周游方法

许多图问题的求解需要检查或处理（访问）图的每个结点和每条边。宽度优先检索（breadth-first search, breadth-first traveral）和深度优先检索（depth-first search, depth-first traveral）是两种基本

检索方法，可有效地访问图的每个结点和每条边(不重复)。不同的问题需选择采用不同的检索方法求解，能得到线性复杂度($\Theta(n+m)$)的算法。

我们首先以有向图为例，介绍这两种检索方法。

3.1 深度优先检索概述

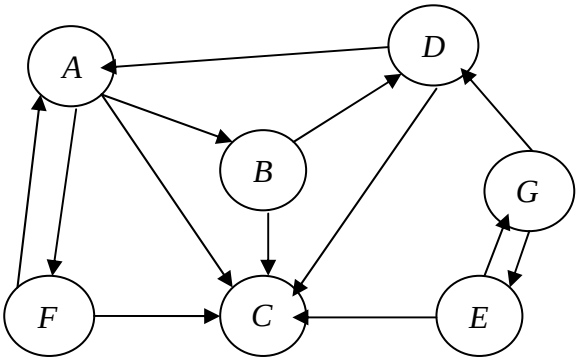
深度优先检索是树周游方法（先根周游，中根周游，后根周游）的推广，契合了递归算法的结构，所以它比宽度优先检索的应用范围更广，借以生成许多重要算法，后面几节会看到一些例子。

检索起点由问题本身决定或随意选取。深度优先检索可看作“旅游”某景点区。

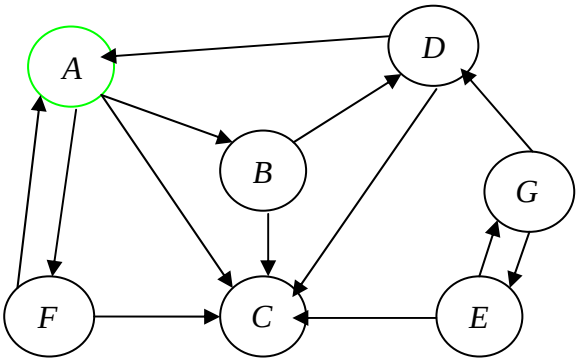
把有向图想象成以桥连接的丛岛。桥是单向交通道，但步行者不受此限，可来回走。让我们来周游此丛岛。我们采取的策略是，从某个岛出发，我们首次穿过某桥时，总是沿着交通方向走，这称为“探索（exploring）”一条边。所以当我们逆着交通方向通过某桥时，必然回到以前到过的地方， 这称为“回溯（backtracking）”。深度优先检索的要旨是尽可能地“探索”，“探索”不成才作“回溯”，以寻找其他“探索”的机会。直到“回溯”到起点，无法再探索为止。

术语：“探索(exploring)”，“回溯(backtracking)”，“发现（discover）”,“检查(checking)”，“未发现点（undiscovered）”，“活动点（active）”，“死点(dead end)”，“完成点(finished)”。

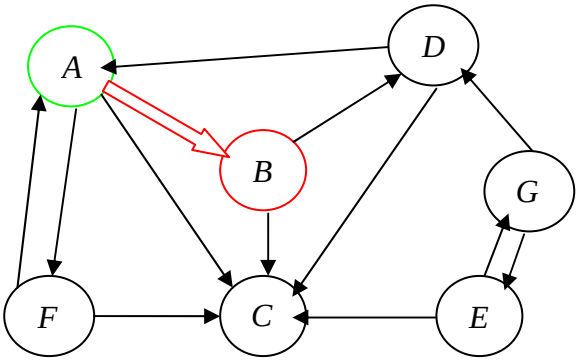
例.



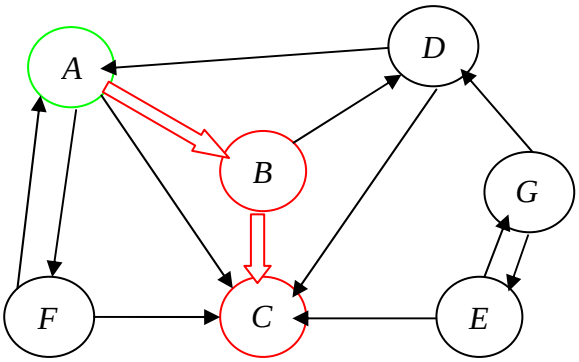
有向图示例



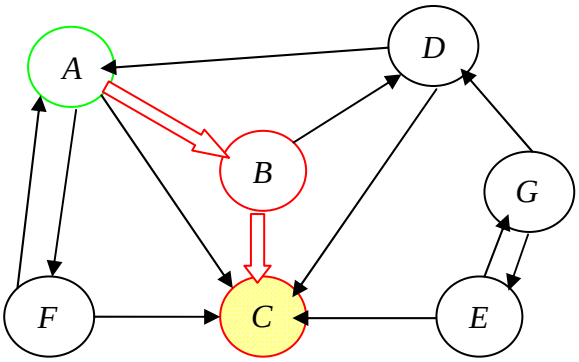
深度优先检索从结点 A 开始



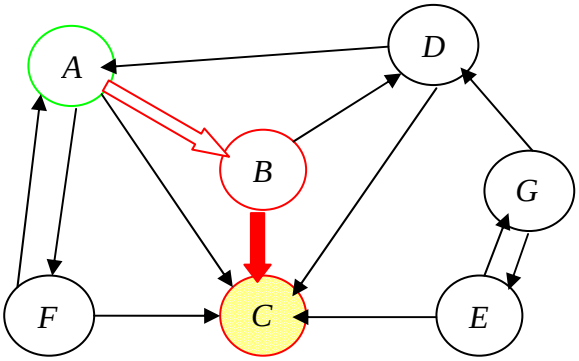
探索边 AB，发现结点 B



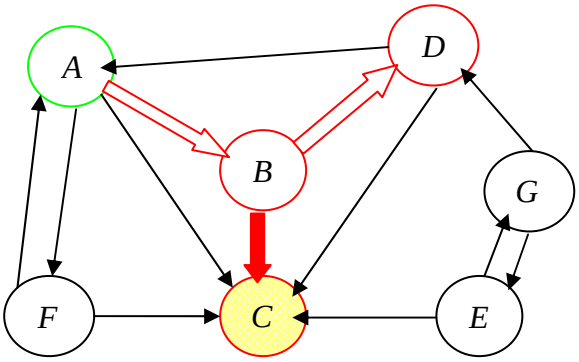
探索边 BC，发现结点 C



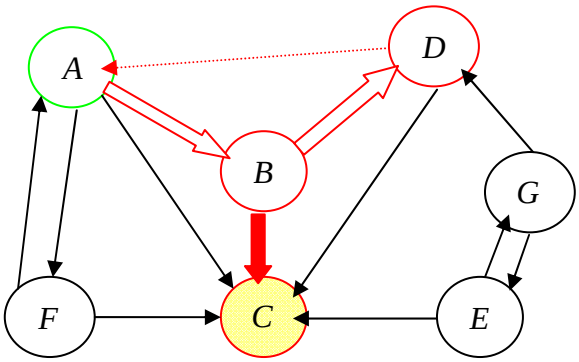
从结点 C 无路可走，称为“死角（dead end）”



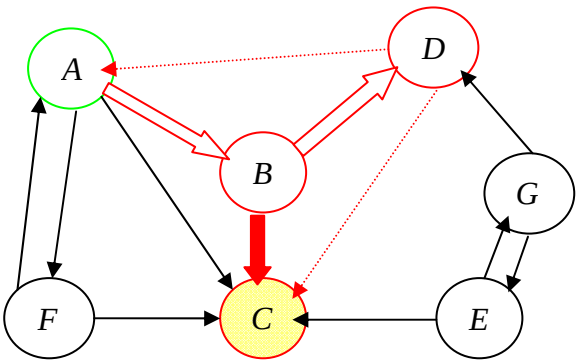
C 称为完成点（finished），从 C 回溯到 B。



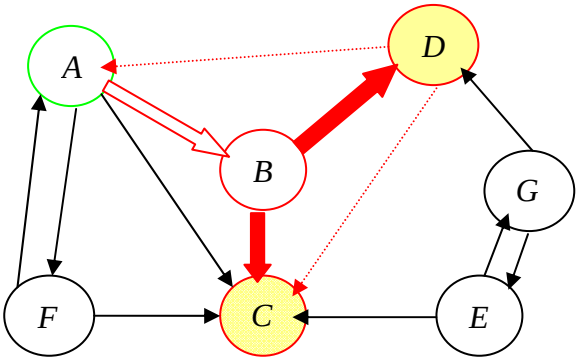
探索边 BD,　发现结点 D



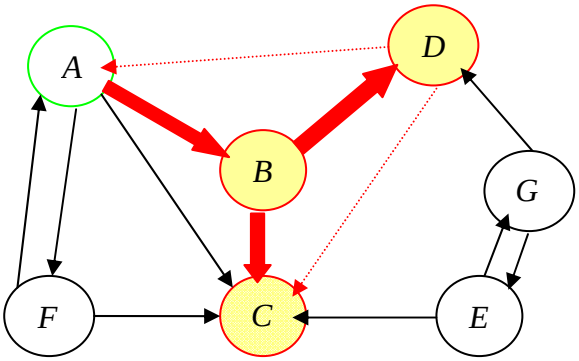
检查边 DA（A 是已发现的结点，　无需探索）



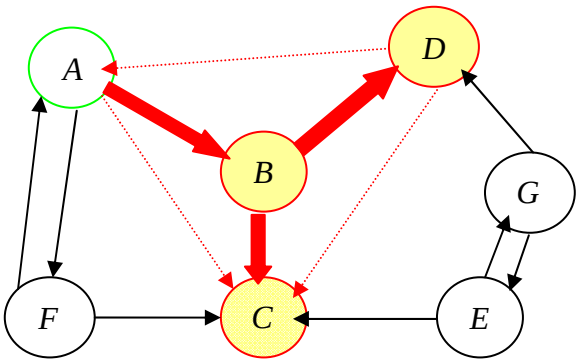
检查边 DC



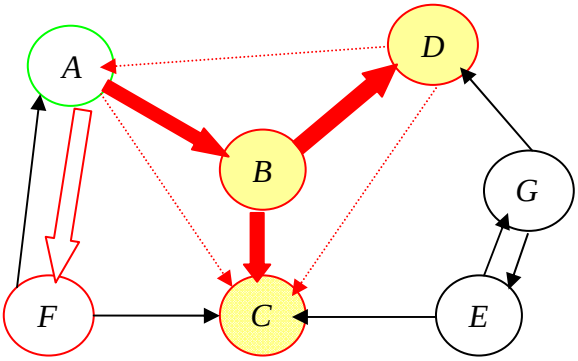
D 变成完成点，从 D 回溯到 B。



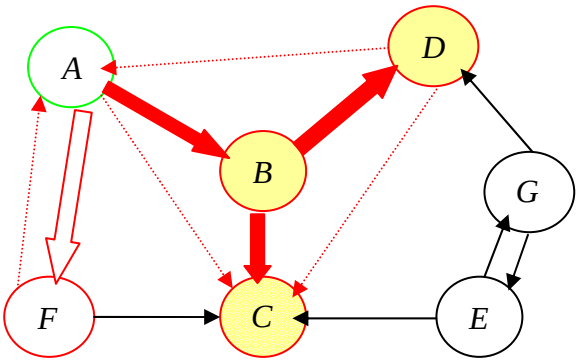
B 变成完成点，从 B 回溯到 A。



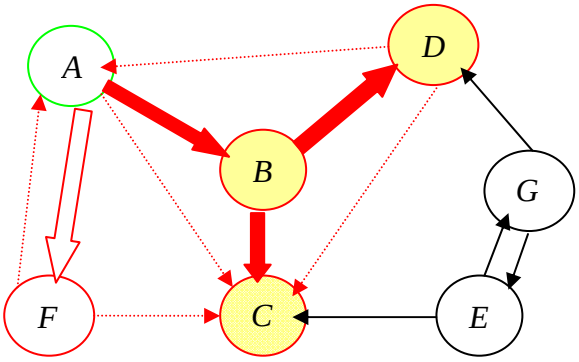
检查边 AC



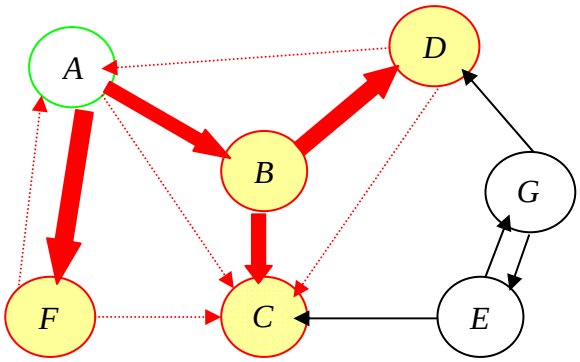
探索边 AF，发现结点 F



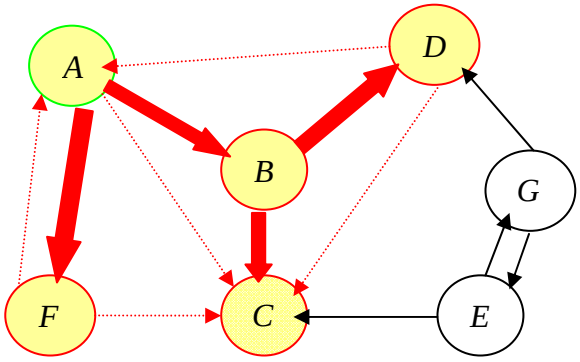
检查边 FA



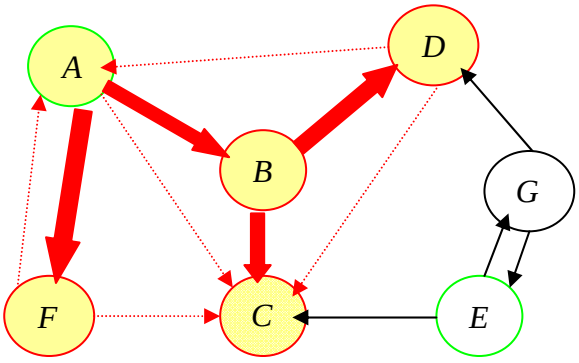
检查边 FC



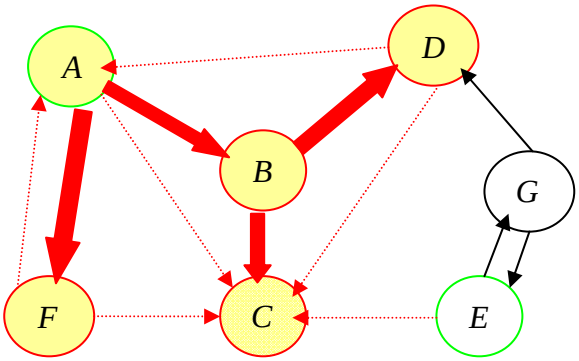
F 变成完成点，从 F 回溯到 A。



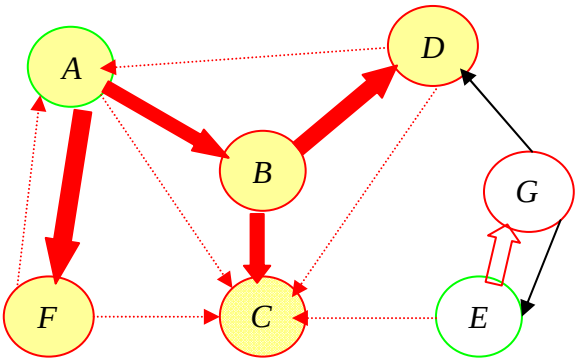
以 A 为起点的深度优先检索至此完成



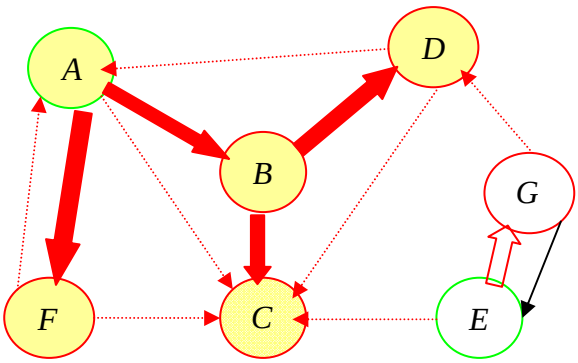
接着以 E 为起点进行深度优先检索



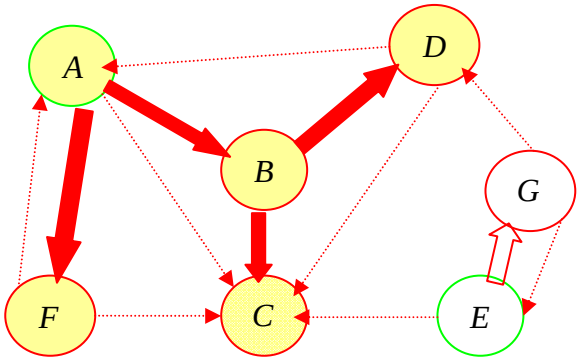
检查边 EC



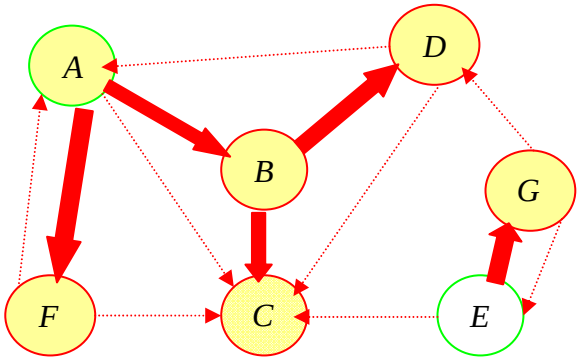
探索边 EG，发现结点 G



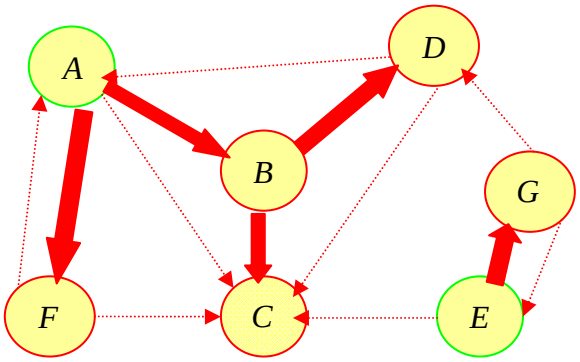
检查边 GD



检查边 GE



G 变成完成点，从 G 回溯到 E。



E 变成完成点。至此图的深度优先周游完成

深度优先检索树(depth-first spanning tree**)**：由起始点为树根，由探索边作为树枝形成。非树枝边有特殊的意义，往后有细致讨论。

深度优先检索和周游的概略描述

1．以v为起点的深度优先检索：

dfs(G, v)

Mark v as “discovered”.

For (each vertex w such that edge vw is in G) do

if (w is undiscovered) then

{dfs(G, w); that is, explore vw, visit w, explore from there as much as possible, and backtrack from w to v.}

Otherwise:

{ “Checking” vw without visiting w.}

endif

Mark v as “finished”.

repeat

end dfs(G, v)

2. 对图G的深度优先周游：

Procedure dfsSweep(G)

Initialize all vertices of G to “undiscovered ”.

For each vertex $v \in G$, in some order do

if (v is undiscovered) then

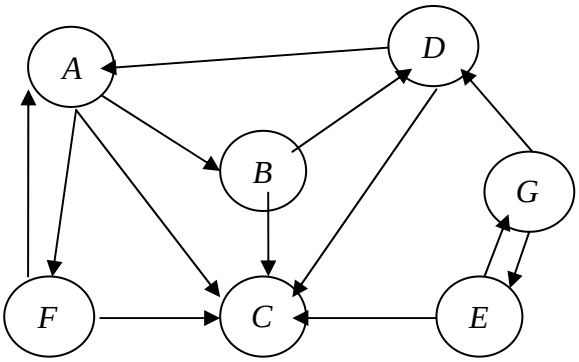
{ dfs(G, v) ; that is, perform a depth-first search beginning (and ending) at v; any vertices discovered during an earlier depth-first search visit are not revisited; all vertices visited during this dfs are now classified as “discovered”.}

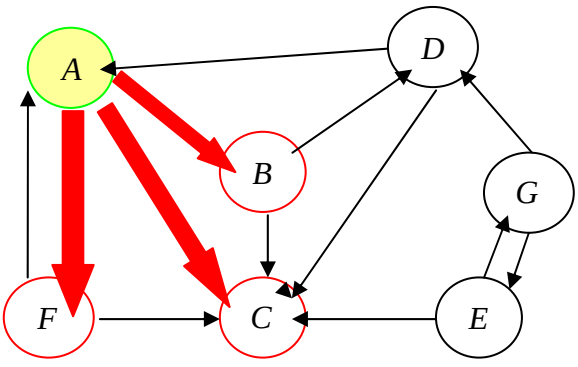
endif

end dfsSweep

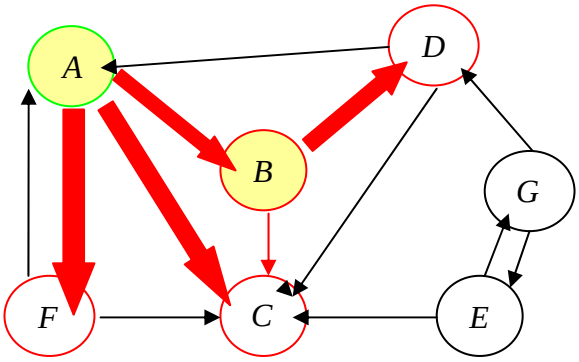
3.2 宽度优先检索概述

宽度优先检索可看作传染病的传播。所以从图的某个结点开始，首先传播到与它相邻的所有结点，传播途径是边。然后再从这些被传染的结点出发，以同样的方式继续传播。

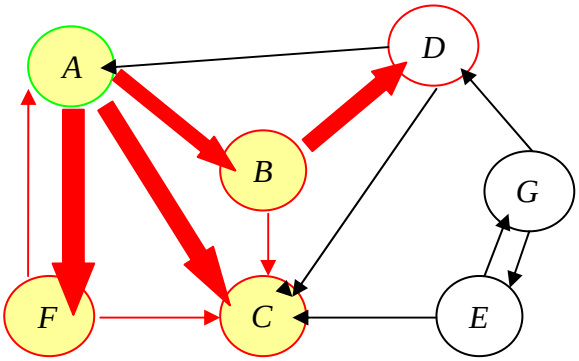




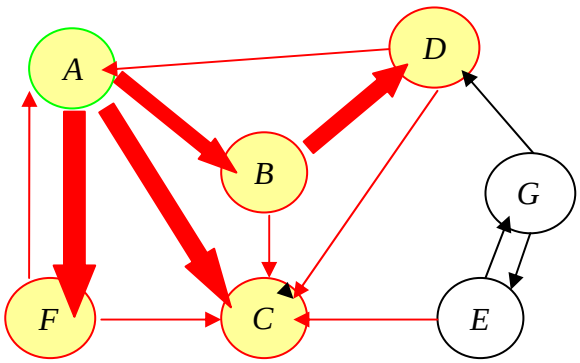
从结点 A 为起点分别沿边 AB, AC, AF 传染到结点 B，C，F。



从结点 B 检查边 BC（C 已被传染），沿边 BD 传染到结点 D。

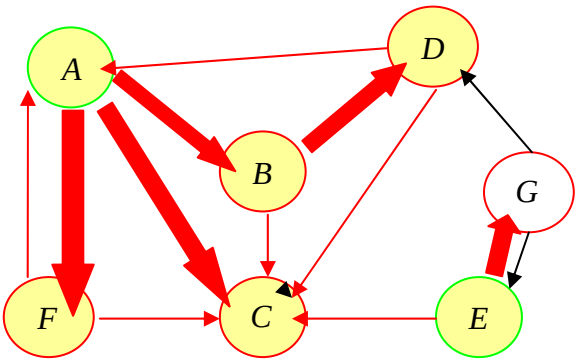


在结点 C 无传播路径。

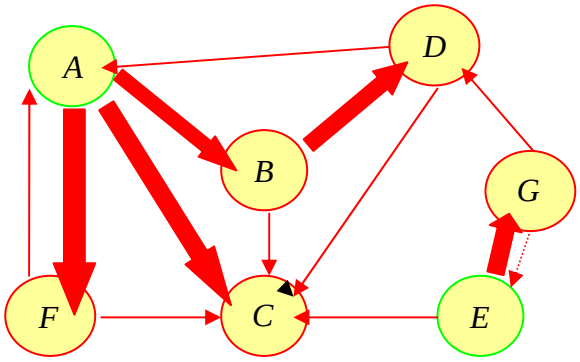


在结点 D 处，检查 DA，DC，以 A 为起点的宽度

优先检索至此完成。



从结点 E 处，检查边 EC， 沿边 EG 传染到结点 G



在结点 G 处检查边 GD，GE。至此整个图的宽度优先周游完成

宽度优先生成树(breadth-first spanning tree): 由起始点S为树根，传染边作为树枝形成。特征：在该树中根S到任何其他点V的路径是图G中S到V的最短路径。

算法 7.1 宽度优先检索

数据结构：

链表数组：$adj(n)$，

点状态数组：$color(n)$，

生成树结点的父亲数组： $parent(n)$

bfs$(n, adj, color, s, parent)$

Queue Q

$parent(s) \leftarrow -1$

$color(s) \leftarrow gray$

enqueue(Q, s)

while(Q is nonempty) do

$v \leftarrow$ front(Q)

dequeue(Q)/delete v///

for (each vertex w in the list $adj(v)$) do

if($color(w) == white$) then

{$color(w) \leftarrow gray$

enqueue(Q, w) //w enters in Q//

$parent(w) \leftarrow v$ //process tree edge vw//}

endif

repeat

$color(v) \leftarrow black$

repeat

end bfs

bfsSweep$(n, adj, color, parent)$

Initialize $color(1), color(2), \cdots, color(n)$ to $white$

for $v \leftarrow 1$ to n do

if ($color(v) == white$) then

call bfs($n, adj, color, v, parent$)

endif

repeat

end bfsSweep

3.3 深度优先检索与宽度优先检索的比较

数据结构：深度优先检索：栈；宽度优先检索：队列。

生成树：宽度优先检索树的树枝有特征意义，非树枝无任何意义；而深度优先检索树的非树枝比树枝更有意义，这是后话，此处先指出这一点。

深度优先检索存在“后处理”决定了它的广泛应用，后面各节有较细致的介绍；而宽度优先检索的简单“一次性”处理过程相对来说只有较少的应用范围，典型的应用如找**无向图的连通分支**，**求解所有边权为单位值的赋权图的单源最短路径问题**。事实上，以对称有向图来表示无向图，算法 7.1 中的数组 $parent(n)$ 能够表示结点 s 到其他结点的最短路径。

在无向图中，任何一次宽度优先检索过程中检索到的所有结点构成一个连通分支，以起始点 v 作为这些结点的标记，就得到了无向图的所有连通分支

算法 7.2 利用宽度优先检索求无向图的连通分支（同样以对称有向图来表示无向图）

数据结构：

链表数组：$adj(n)$，点状态数组：$color(n)$

结点所在连通分支标记：$ConComp(n)$

bfs_con$(n, adj, color, s, ConComp)$

Queue Q

$ConComp(s) \leftarrow s$

$color(s) \leftarrow gray$

enqueue(Q, s)

while(Q is nonempty) do

$v \leftarrow$ front(Q)

dequeue(Q)/delete v///

for (each vertex w in the list $adj(v)$) do

if($color(w) == white$) then

{$color(w) \leftarrow gray$

enqueue(Q, w)

$ConComp(w) \leftarrow s$}

endif

repeat

repeat

end bfs_con

```
bfsSweep_con(n,adj,color,ConComp)
Initialize color(1),color(2),⋯,color(n) to white
for v←1 to n do
 if (color(v) == white) then
  call bfs_con(n,adj,color,v,ConComp)
 endif
repeat
end bfsSweep_con
```

4. 有向图上的深度优先检索

我们开始细致处理有向图上的深度优先检索过程。 我们首先描述一种一般化的深度优先检索框架，它可用来求解许多问题，我们将以几个典型问题来说明这一点。

探究无向图上的深度优先检索过程较之有向图更为复杂，这是因为在无向图的表示中，每条边的表示在所用数据结构中出现两次，我们必须对边的这两次出现加以区别对待。我们将在第 6 节单独论述。当然在有些情况下，可以将无向图当作对称有向图来处理，此时，可应用关于有向图的深度优先检索过程。从原理上说，**只要无向图上的深度优先检索可忽略非树枝边**，就可这样做。

关于**转置图**：很多以有向图作为数学模型表示的问题中，采用边的两种方向都是可行的，比如说，有向边 “VW”表示父子关系的话，我们采用相反的方向就表示子父关系。哪种方向作表示更方便就

采用哪种。这两个边方向相反的有向图互称为转置图。

深度优先检索与递归调用的关系：深度优先检索算法可简单地写成一个递归算法；而任何递归算法都可以表示成对一棵树的深度优先检索。以计算斐比那契(Fibonacci)数的递归算法：$F_0 = F_1 = 1$，$F_n = F_{n-2} + F_{n-1}$为例，取$n = 5$。

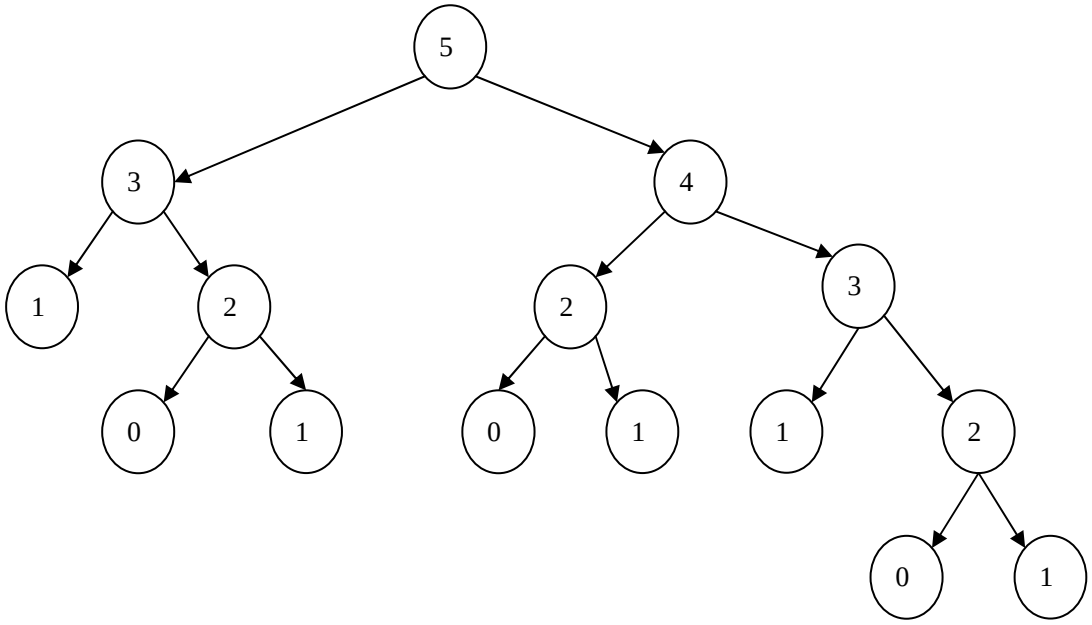
斐比那契数的直接递归算法：

```
Procedure Fibonacci(n, x)
// x表示待求的第n个斐比那契数//
Integer y, z
If(n = 0 or n = 1)then x ← 1
Else
  {
    Call Fibonacci(n − 2, y)
    Call Fibonacci(n − 1, z)
    x ← y + z
  }
Endif
End Fibonacci
```

上述算法的时间复杂度$t(n)$满足下列递归式

$$t(n) = \begin{cases} 1, & n = 0,\ 1 \\ t(n-2) + t(n-1) + 1, & n \geq 2 \end{cases},$$

容易算出$t(n)$是指数复杂度的。



(a) 第 5 个斐比那契数的递归计算调用结构图。标记数字i的结点表示求解第i个斐比那契数。

当然这里出现了许多重复运算，把算法改写成关于一个有向图（这样的图总是有向反圈）的深度优先检索将更为有效。

斐比那契数的优化递归算法

Procedure OpFibonacci(n, x)

Integer $F(0:n)$// 全局数组//

Boolean $Q(0:n)$// 全局数组//

For $i \leftarrow 1$ to n do

$Q(i) \leftarrow$ *false*

repeat

Call Fib(n)

$x \leftarrow F(n)$

End OpFibonacci

Procedure Fib(k)

If($k = 0$ or $k = 1$) then $F(k) \leftarrow 1;\ Q(k) \leftarrow true$

Else

{

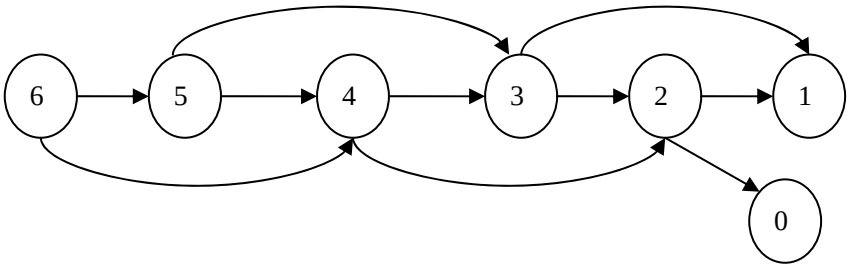
If ($Q(k-2) = false$) then call Fib($k-2$) endif

If ($Q(k-1) = false$) then call Fib($k-1$) endif

$F(k) \leftarrow F(k-2) + F(k-1);\ Q(k) \leftarrow true$

}

Endif Fib



(b) 深度优先检索实际上是将中间运算结果存储起来，避免了重复运算。

去掉重复运算后，算法复杂度降低为线性的。再应用我们后面介绍的有向反圈的反拓扑序，算法可进一步改进为非递归算法。

Procedure SimFib(n, x)

Integer $F(0:n)$// 全局数组//

If($n=0$ or $n=1$)then $x \leftarrow 1$; return endif

$F(0) \leftarrow F(1) \leftarrow 1$

For $i \leftarrow 2$ to n do

$F(i) \leftarrow F(i-1)+F(i-2)$

Repeat

$x \leftarrow F(n)$

end SimFib

深度优先检索的简单应用之一： 找出无向图的连通分支。 还是以对称有向图来表示无向图。

以深度优先检索算法为基本框架，从某结点出发，能探索到的所有结点刚好构成一个连通分支，以出发点的编号作为该连通分支每个结点的标记，

算法 7.3 深度优先检索法找出无向图的连通分支

procedure ConnectedComponents$(adj, n, comp)$

int $color(n)$

int v

Initialize $color$ array to $white$ for all vertices

For $v \leftarrow 1$ to n do

```
    If (color(v) = white) then
      call Compdfs(adj,color,v,v,comp)
    endif
  repeat
end ConnectedComponents

Procedure Compdfs(adj,color,v,compnum,comp)
Int w
Int temadj
color(v) ← gray
comp(v) ← compnum
temadj ← adj(v)
while(temadj ≠ nil ) do
  w ← first(temadj)
  if (color(w) = white) then
  call Compdfs(adj,color,w,compnum,comp)
  endif
  temadj ← rest(temadj)
repeat
//color(v) ← black //
end Compdfs
```

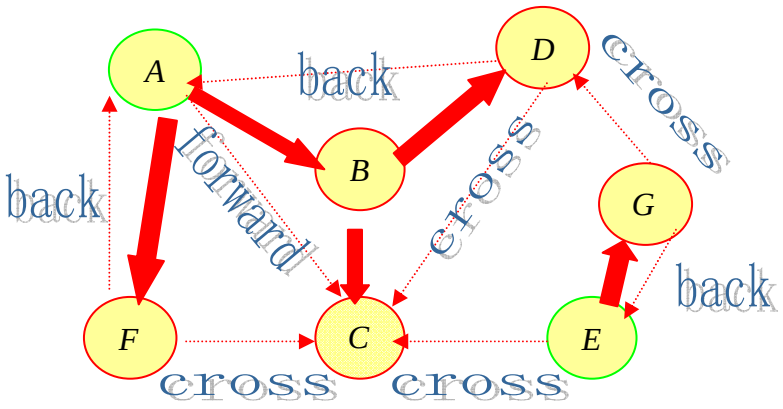
注：不需要 Postorder vertex processing 和 inorder vertex processing。

注：时间复杂度$\Theta(n+m)$。

注：可用链表表示这些连通分支的结点集和边集。

深度优先检索树对于深度优先检索过程提供了非常重要的视角，很多问题虽然不需要明显地建构深度优先检索树，但它作为一个分析方法起作用。

深度优先检索树有关的术语：祖先，父亲，树枝(tree edge)，向后边(back edge)，向前边(forward edge)，交叉边(cross edge)。



非树枝边的分类

算法 7.4 完整细化的深度优先周游过程：

procedure DfsSweep($n, adj, color, ans \cdots$)

int *ans* // answer//

Allocate *color* array and initialize to *white*.

For each vertex v of G, in some order:

```
  If(color(v) = white) then
   {
      call Dfs(adj,color,v,vAns,⋯)
     Process vAns
   }
  endif
repeat
process ans
end DfsSweep

procedure Dfs(adj,color,v,vAns,⋯)
 int w
 int temadj
 int temans
 color(v) ← gray
 Preorder processing of vertex v
 temadj ← adj(v)
 while(temadj ≠ nil ) do
    w ← first(temadj)
    if(color(w) = white) then
     {
       Exploratory processing for tree edge vw
```

call Dfs($adj, color, w, wAns, \cdots$)

Backtrack processing for tree edge vw, using $wAns$

(like inorder processing of vertex v)

}

else

{ Checking (i.e. processing) for nontree edge vw}

endif

$temadj \leftarrow rest(temadj)$

repeat

Postorder processing of vertex v, including final computation of $vAns$

$color(v) \leftarrow black$

end Dfs

深度优先检索的性质

对深度优先周游过程中的两个重要操作：探索到新结点和从完成点回溯，进行记时，即进行一次这样的操作，时钟走动一次。这样就可以得到每个结点的活动时间段。活动时间段对于分析深度优先检索的性质和特征来说，是非常方便好用的概念。

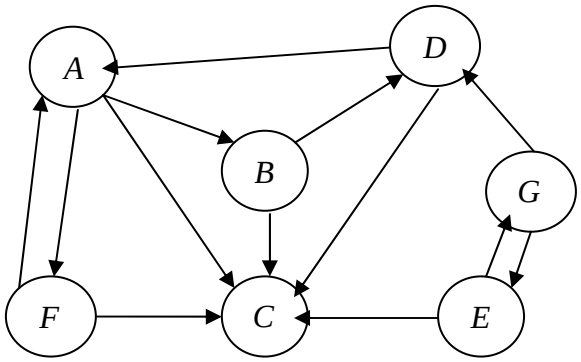
结点v的活动时间段：

$$active(v) = discovertime(v), \cdots, finishTime(v),$$

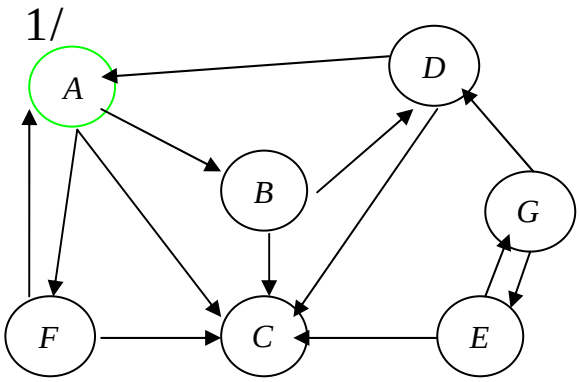
其中$discovertime(v)$是结点v被发现的时间 $finishTime(v)$是

结点V完成了检索的时间（即回溯时刻）

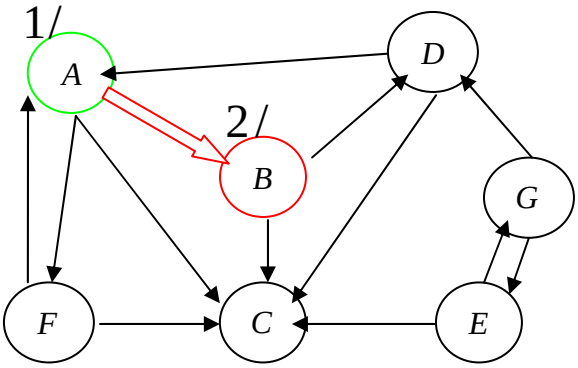
例子说明



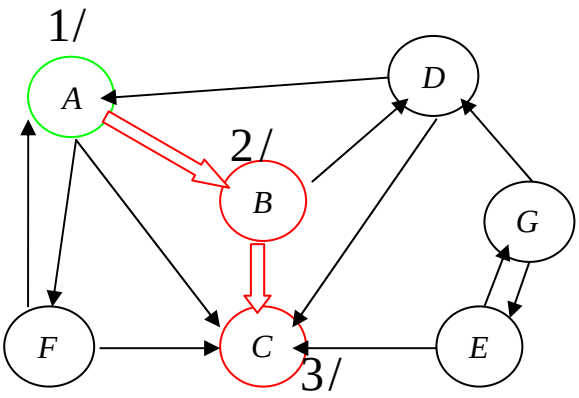
有向图示例



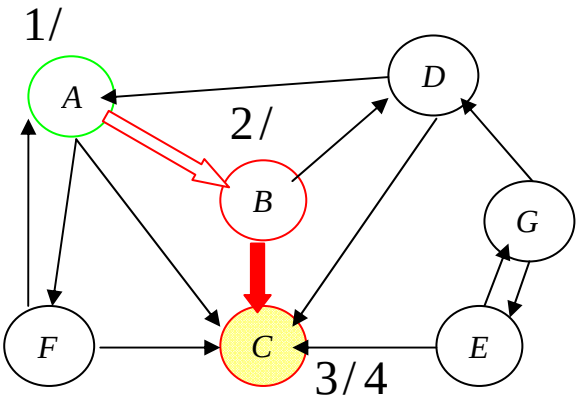
深度优先检索从结点 A 开始



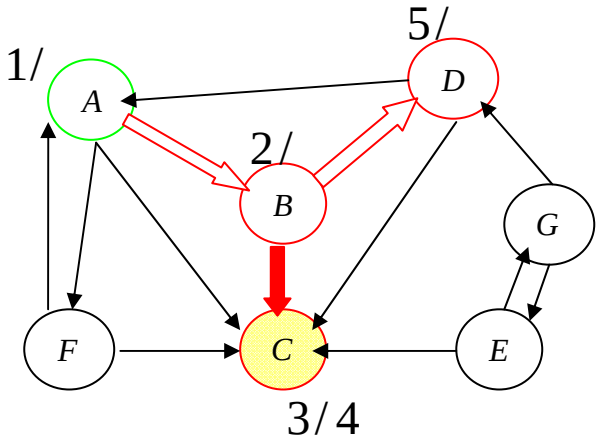
探索到结点 B



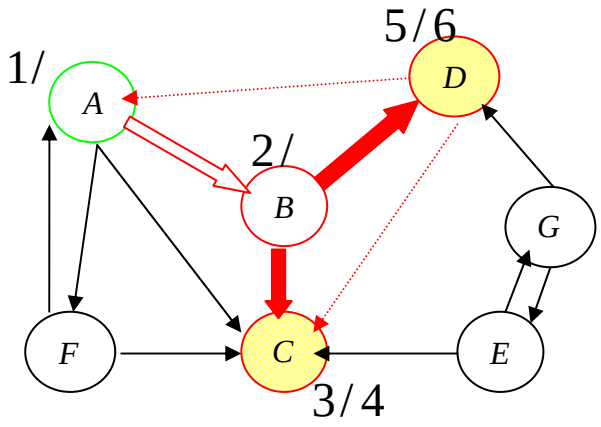
探索到结点 C



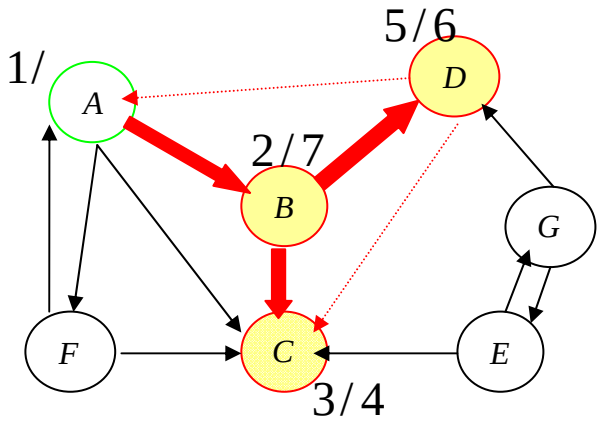
C 变成完成点，从 C 回溯到 B



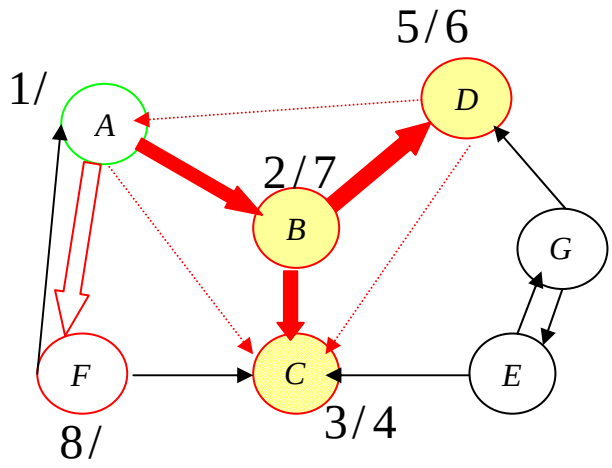
从 B 探索到 D



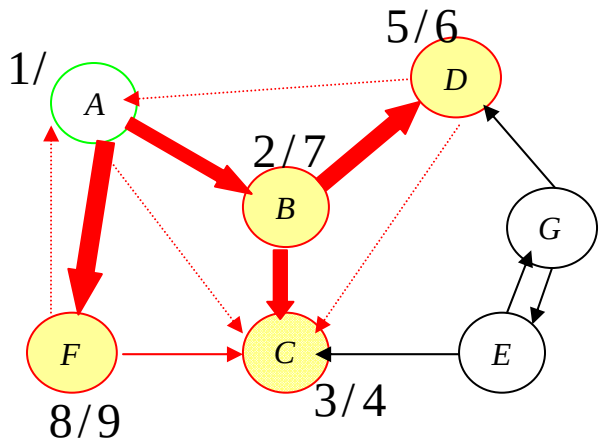
(检查边 DA ， DC)，D 变成完成点，从 D 回溯到 B。



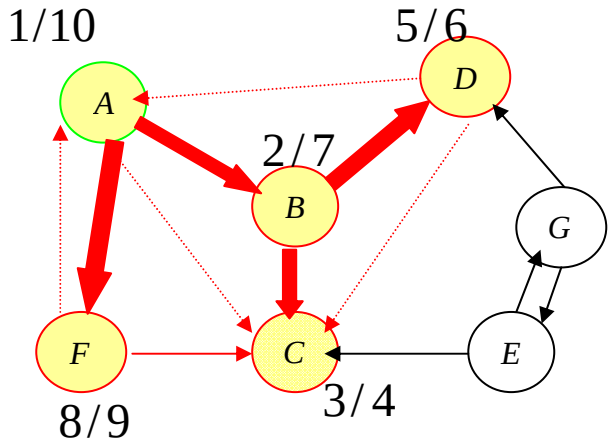
B 变成完成点，从 B 回溯到 A。



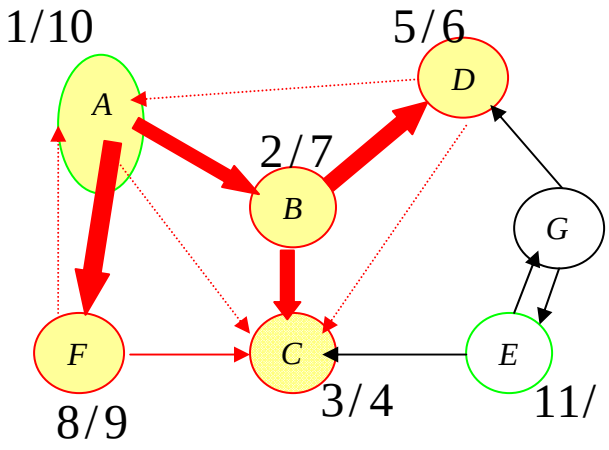
（检查边 AC），探索到结点 F。



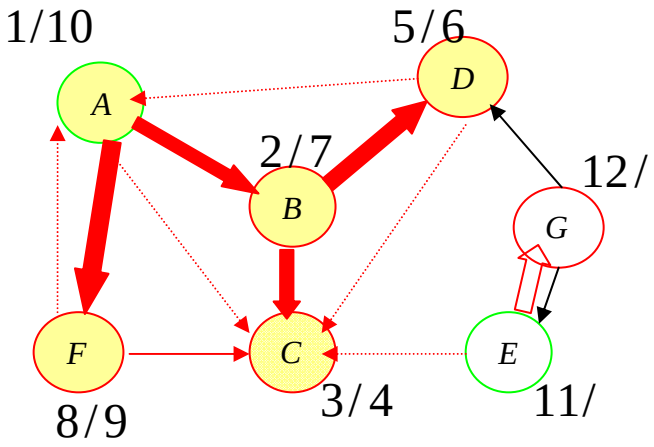
（检查边 FA，FC），F 变成完成点，从 F 回溯到 A。



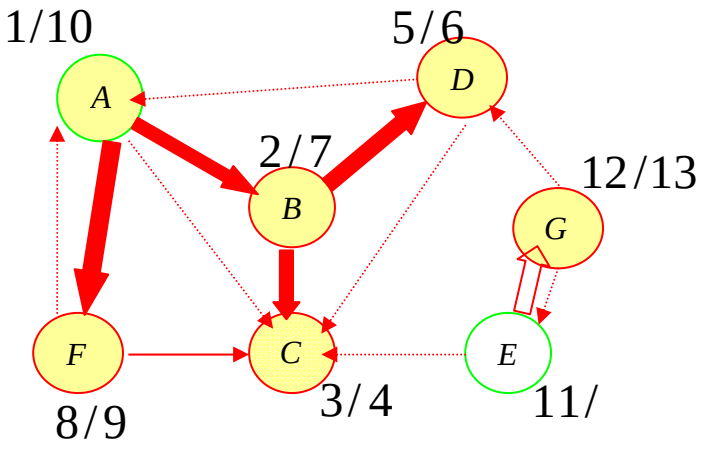
以 A 为起点的深度优先检索至此完成



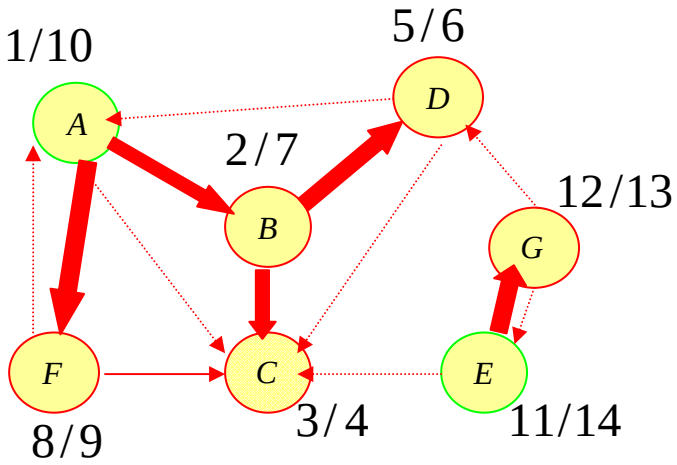
接着以 E 为起点进行深度优先检索



（检查边 EC），探索到结点 G。



（检查边 GD，GE），G 变成完成点，从 G 回溯到 E。



E 变成完成点。至此图的深度优先周游完成。

算法 7.5　算法 7.4 的细化(加入树和活动时间)

procedure　DfsSweep($n, adj, color, ans,$

$parent, dis\,\mathrm{cov}erTime, finishTime\cdots$)

int ans // answer//

int $time$

Allocate $color$ array and initialize to $white$.

$time \leftarrow 0$

For each vertex v of G, in some order:

If($color(v) = white$) then

{

$parent(v) \leftarrow -1$

call Dfs($adj, color, v, vAns, \cdots$)

Process $vAns$

}

endif

repeat

process　ans

end DfsSweep

procedure Dfs($adj, color, v, vAns, \cdots$)

int w

int $temadj$

int $temans$

$color(v) \leftarrow gray$

$time++;\ discoverTime(v) \leftarrow time$

Preorder processing of vertex v

$temadj \leftarrow adj(v)$

while($temadj \neq nil$) do

$w \leftarrow first(temadj)$

if($color(w) = white$) then

{

$parent(w) \leftarrow v$

Exploratory processing for tree edge vw

call $\mathrm{Dfs}(adj, color, w, wAns, \cdots)$

Backtrack processing for tree edge vw, using $wAns$

(like inorder processing of vertex v)

}

else

{ Checking (i.e. processing) for nontree edge vw }

endif

$temadj \leftarrow rest(temadj)$

repeat

$time++;\ finishTime(v) \leftarrow time$

Postorder processing of vertex v, including final computation

of $vAns$

$color(v) \leftarrow black$

end Dfs

定理 7.1 假定$active(v)$如上定义，则

性质 1. w是v的子孙$\Leftrightarrow active(w) \subseteq active(v)$。此时，如果$w \neq v$，则$active(w) \subset active(v)$。

性质 2. 如果v与w没有祖孙关系，则

$$active(v) \cap active(w) = \varnothing$$。

性质 3. 如果$vw \in E$， 则

a. vw是交叉边$\Leftrightarrow active(w)$完全在$active(v)$之先；

b. vw是向前边$\Leftrightarrow$存在另一个结点x，使得

$$active(w) \subset active(x) \subset active(v)$$；

c. vw是树枝$\Leftrightarrow active(w) \subset active(v)$，但不存在另一个结点$x$，使得$active(w) \subset active(x) \subset active(v)$；

d. vw是向后边$\Leftrightarrow active(v) \subset active(w)$。

推论 7.2 在结点v的活动时间段里发现的所有结点恰是它的所有子孙。

另一个容易看到的事实： w是v的子孙当且仅当v没被发现前存在一条v到w的道路，其上所有点都尚未被发现。严格的证明形成下面的“白道定理”。

定理 7.3（**白道定理** White Path Theorem）在对一个图G的任何深

度优先检索中，结点w是结点v在相应的深度优先检索树中的子孙当且仅当，在结点v被发现的时刻（恰在其被染成灰色之前），G中有一条从v到w的道路，其上的结点全是白结点。

证明：（仅当，必要性）如果w是v的子孙，由深度优先检索树的生成知道，树中从v到w的道路在v被发现的时刻是一条白道路。

（当，充分性）对从v到w的白道路的长度k施行归纳，基础情形是$k=0$，此时$v=w$，定理成立。对于$k\geq 1$，设$P=(v,x_1,\cdots,x_k)$，其中$x_k=w$，是从v到w的长度为k的白道路。设x_i是白道上在v活动时间段内最早被发现的结点。这样的假定是合理的，因为至少x_1能在此期间被发现。这样，x_i是v的子孙。把道路P分成从v到x_i的一段P_1和从x_i到w的一段P_2，如此看来，当x_i被发现时，P_2是从x_i到w的白道路，由归纳假设，w是x_i的子孙，从而w是v的子孙。

关于有向反圈的讨论：

有向反圈之重要性在于

1．工程调度问题由有向反圈模型来表征：某工程由若干任务所构成，这些任务之间存在依赖关系，即每个任务的执行必须在若干任务完成之后方可进行。

2．许多有向图上的问题（指数复杂度）在有向反圈（线性复杂度）上更易解决。

3．任何有向图都唯一对应于一个有向反圈（**收缩图**，condensation

graph）

有向反圈在数学上对应于结点集上的一个偏序。 有向反圈G的边vw可解释为$v \prec w$，当G中有从v到w的道路时，也认为$v \prec w$（为使其满足传递性）。

拓扑序（Topological Order）：当我们考虑某些有向图上的问题时，可能出现这样的念头：“如果有向图存在一种画法能使边上的方向都是从左到右（或从上到下，象根树一样），这将有助于解决问题吗？” 当然，如果有向图包含回路，这种画法显然不可能。但是，如果有向图不包含回路，也就是说，是反圈，这种画法能够做到，该画法中结点形成一个顺序，称为图的拓扑序。

拓扑序的定义：设$G=(V,E)$是n阶有向图。G的一个拓扑序是指用整数$1,\cdots,n$给G的n个不同结点进行编号（称为结点的拓扑数，topological numbers）使得对G的任何边vw来说，v的拓扑数总小于w的拓扑数。G^T的拓扑序称为G的反拓扑序（reverse topological order）。

引理 7.4. 如果图G有圈，则G没有拓扑序。

注：在某种意义下，拓扑序是反圈的基本问题，有了拓扑序，许多问题就会变得简单明了。在有些情况下，仅仅意识到拓扑序的概念就可能引导我们得到问题的有效解。

注：以深度优先检索算法为基础，就可得到求解拓扑序的算法，我们给出求反拓扑序的算法，因为在应用问题中经常用到的是反拓扑序。

算法 7.6 求反拓扑序数 Reverse Topological Ordering

在 $DfsSweep$ 框架里另外需要的数据结构：$topo(n)$，$toponum$

输出：全局数组 $topo(n)$ 被赋以对应结点的反拓扑序

Strategy: Modify the $DfsSweep$ and Dfs skeleton as follows :

1. In $DfsSweep$, initialize $toponum$ to 0;
2. In Dfs, at postorder processing, insert

$$toponum++;\ topo(v) \leftarrow toponum$$

注：如果是求解拓扑序，相应地，

1. In $DfsSweep$, initialize $toponum$ to $n+1$;
2. In Dfs, at postorder processing, insert

$$toponum--;\ topo(v) \leftarrow toponum$$

以求反拓扑序数为例

procedure DfsSweep_topOrder($n, adj, color, topo$)

int $toponum$

Allocate $color$ array and initialize to $white$.

$toponum \leftarrow 0$

For each vertex v of G, in some order:

  If($color(v) = white$) then

    call Dfs_topOrder($n, adj, color, v, topo$)

  endif

repeat

end DfsSweep_topOrder

procedure Dfs_topOrder($n, adj, color, v, topo$)

int w

int $temadj$

$color(v) \leftarrow gray$

$temadj \leftarrow adj(v)$

while($temadj \neq nil$) do

  $w \leftarrow first(temadj)$

  if($color(w) = white$) then

    call Dfs_topOrder($n, adj, color, w, topo$)

  endif

  $temadj \leftarrow rest(temadj)$

repeat

$toponum++$; $topo(v) \leftarrow toponum$

$color(v) \leftarrow black$

end Dfs_topOrder

定理 7.5. 上述算法给出了反圈G的一个反拓扑序。

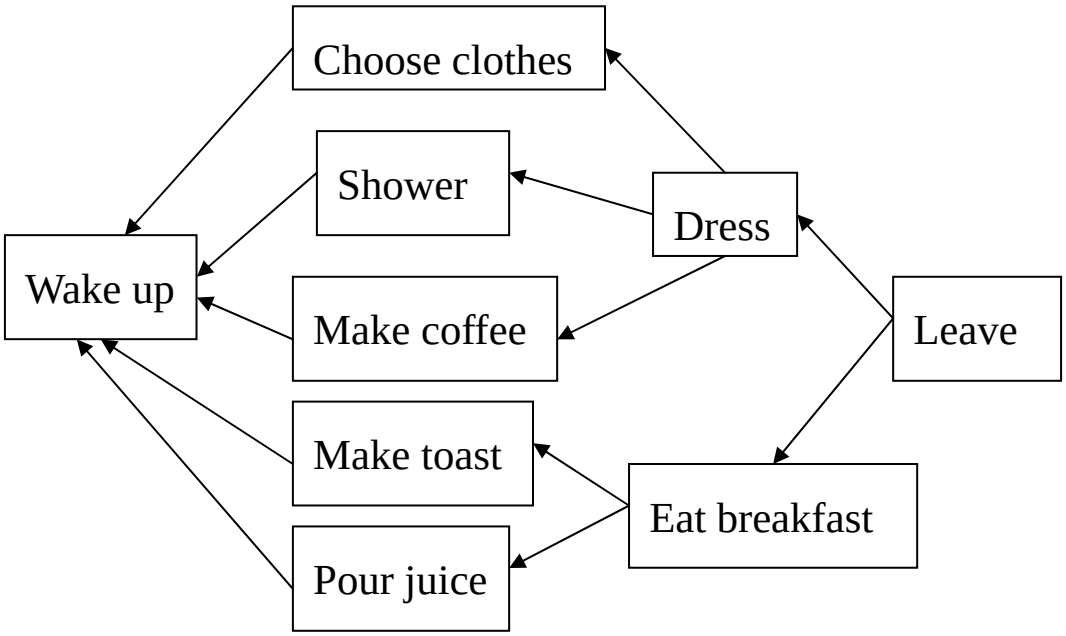
证明：因为只是在结点刚处于完成状态（即回溯）时，被赋予拓扑数。而结点一旦成为完成状态，将不会被再次访问。所以结点被赋拓扑数这种操作对每个结点只会发生一次，n个结点的拓扑数都不同，且取值范围是$1, \cdots, n$这n个数。下面只需要说明，对G的每条边vw来说，都有$topo(v) > topo(w)$。以深度优先检索对边的分类来看，反圈没有“向后边”，而“树枝”，“向前边”，“交叉边”，都满足上面

的不等式。

依赖性任务构成的工程调度问题：描述各任务之间依赖关系的最自然方式是列表，表中的每一行列出某个任务直接依赖的所有任务。这儿有一个简单例子：一个人早上起床到离开家所干事项之间的依赖关系，把这些事项作为任务。这些任务的编号按字母顺序来做。

Task and Number	Depends on
Choose clothes 1	9
Dress 2	1,8
Eat breakfast 3	5,6,7
Leave 4	2,3
Make coffee 5	9
Make toast 6	9
Pour juice 7	9
Shower 8	9
Wake up 9	-

用有向图G来表示这些任务之间的依赖关系：每个结点表示一项任务，边vw表示任务v直接依赖任务w。则上面的表给出了图G的外邻点表。图G是反圈，成为依赖图。依赖图的反拓扑序给出工程中任务进行的先后顺序。



工程各任务的依赖关系示例

图G的一个反拓扑序：

深度优先周游

深度优先检索的起点：Choose clothes, 得序：Wake up: 1, Choose clothes: 2 ;

深度优先检索的起点: Dress, 得序：Shower: 3 , Dress: 4;

深度优先检索的起点: Eat breakfast, 得序：Make coffee: 5, Make toast: 6, Pour juice: 7, Eat breakfast:8;

深度优先检索的起点: Leave, 得序; Leave: 9.

按照这个序来进行这些任务，可使得在进行任何任务时，它依赖的任务已经完成。

关键路径（Critical Path）分析

关键路径分析与求拓扑序有关，但它本身是一个最优化问题，即要求反圈中的最长道路。以上述调度问题为例，一个工程包括一组任务，这些任务之间有依赖关系。现在，假定完成每个任务有一个时间量，此外，假定每个任务都可在它所依赖的任务完成的同时开始进行，也就是说有足够的工人可供支配。当然后一个假定在许多实际情况中是有问题的，但是初步考虑问题时我们先作此简化。

假定工程在0时刻开始。我们可以定义每个任务的最早完成时间。

最早开始时间，最早完成时间，关键路径

假定一个工程包括一组任务，编号为$1, \cdots, n$，对于每个任务v，有一个它所直接依赖任务的列表，一个非负实数表示完成该任务所花费的时间（duration）$d(v)$。任务v的**最早开始时间**$est(v)$定义为

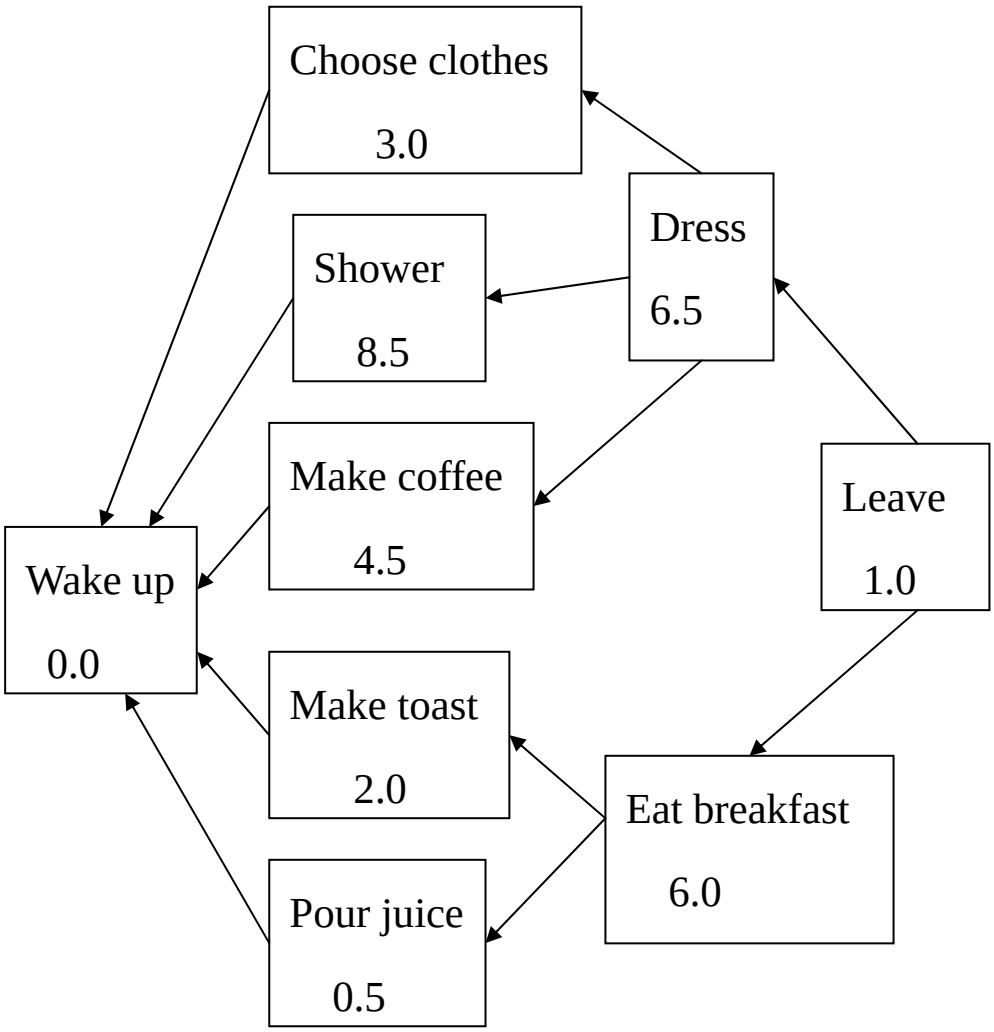
1. 如果任务v不依赖任何任务，则$est(v)=0$；
2. 否则，$est(v)$是所依赖任务最早完成时间的最大值。

任务v的**最早完成时间**$eft(v)$则是$est(v)$加上$d(v)$。

工程的一个**关键路径**是指一个任务序列$v_0, v_1, \cdots, v_k$使得

1. v_0无依赖的任务；
2. 对于任意$1 \le i \le k$，v_{i-1}是v_i的一个直接依赖任务，且$est(v_i)=eft(v_{i-1})$；
3. $eft(v_k)=\max\limits_{1\le v\le n} eft(v)$。

关键路径没有“松弛部分（slack）”，即路径上，一个任务的完成和下一个任务的开始之间没有停顿。换句话说，如果在关键路径上，v_i紧跟着v_{i-1}，则v_{i-1}的最早完成时间是v_i的所有直接依赖的任务中最大的。所以v_{i-1}是v_i的关键依赖，意味着v_{i-1}的任何延误将造成v_i的延误。以另外一个观点看，假如我们要通过加快某个任务的完成时间来加快所有任务(整个工程)的完成时间，如果该任务不在所有关键路径上，则不可能做到这一点。研究关键路径的实际意义正在于此。

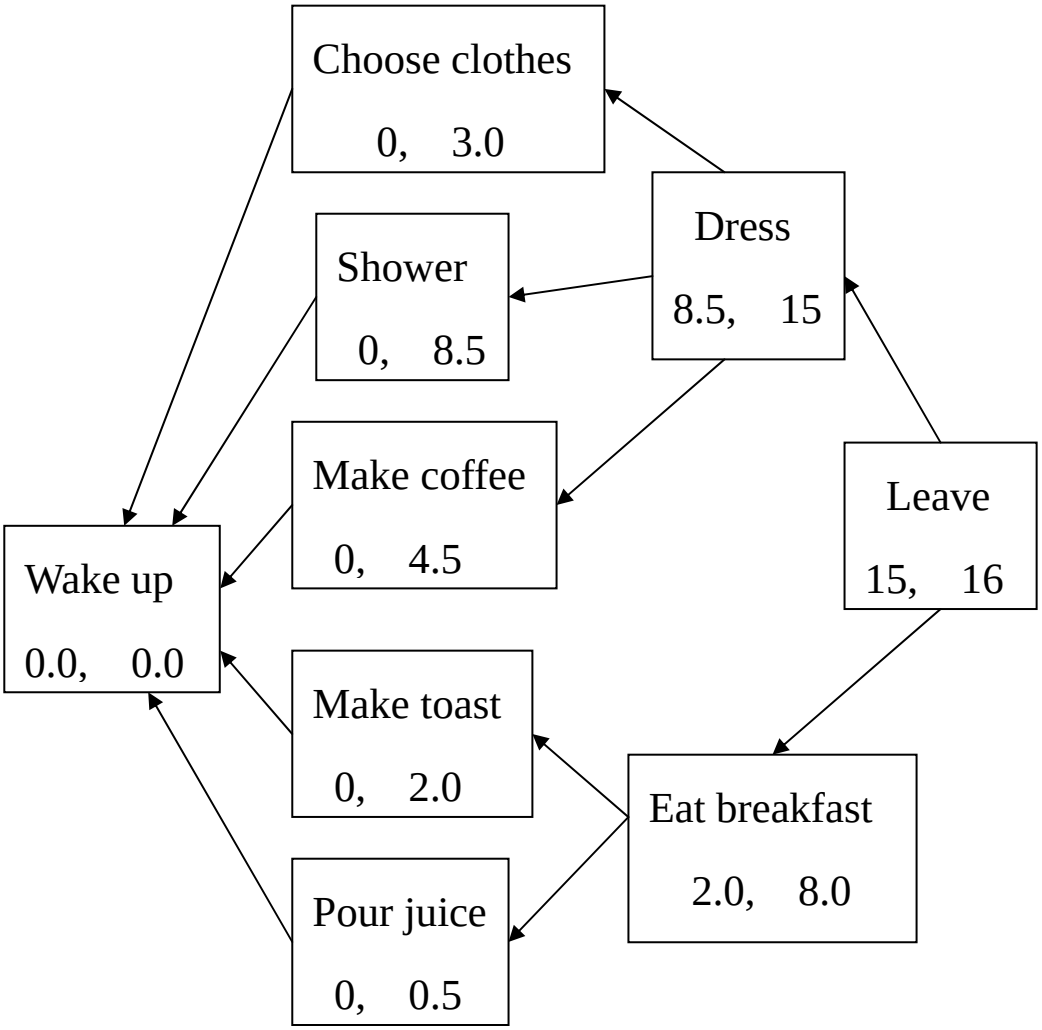


关于各任务花费时间的描述

为简单起见，我们假定每个任务有固定的花费时间。在很多实际情形下，可以分配更多的资源给某任务以减少其花费时间，这或许要减少不在关键路径上的某任务的资源。

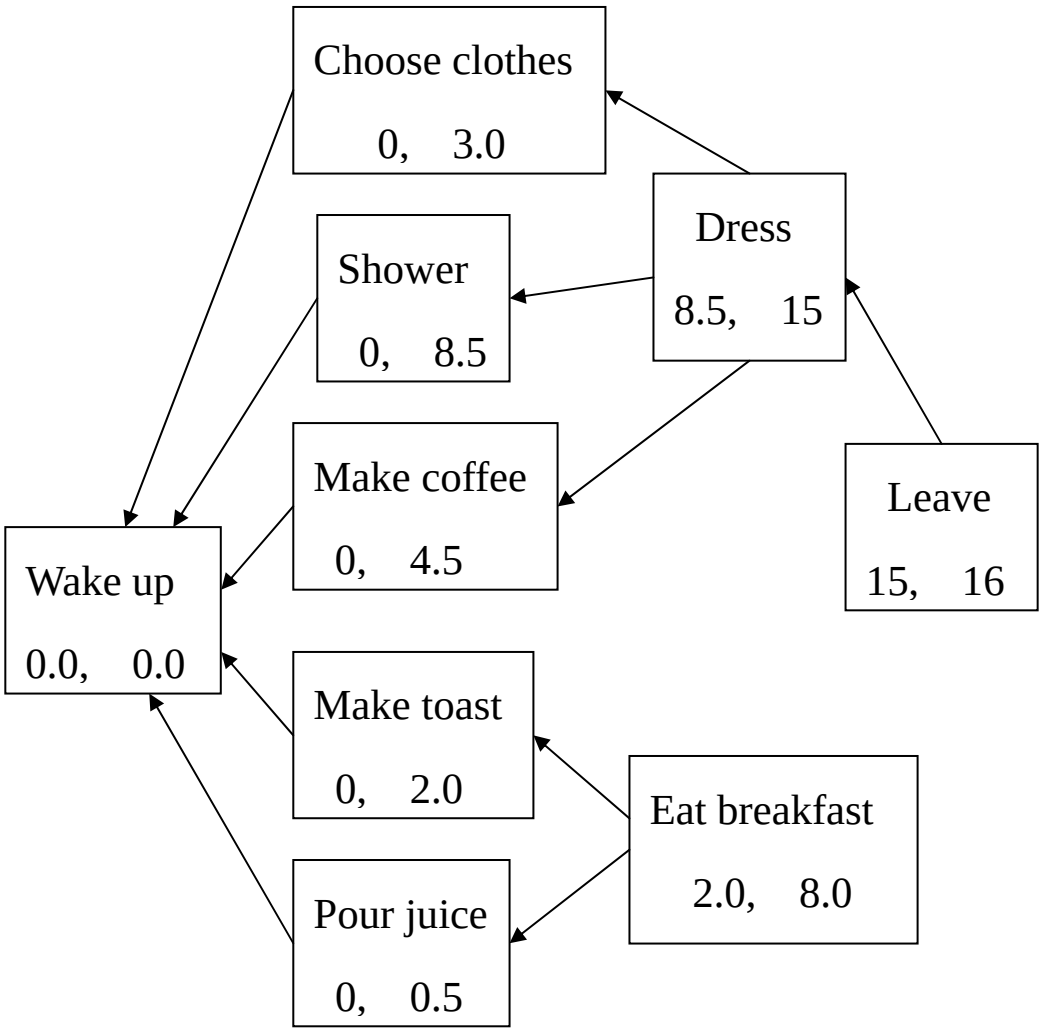
来计算上面例子的关键路径。假定任意多的任务可以同时进行，只要它们所依赖的任务已经完成。如下图所示，各任务的最早开始时间和最早完成时间列在表示结点的方框中。

关键路径：wake up→shower→Dress→leave.



各任务的最早开始时间和最早完成时间

关键依赖：在每个任务v所依赖的任务中，其中有个任务w的最早完成时间最大，即有：$est(v)=eft(w)$。w称为v的关键依赖任务。



各任务的关键依赖任务

关键路径是点加权反圈上的最优化问题。为运算方便起见，对图做点加工：即增添一个结点$done$，它直接依赖的任务是那些不被任何任务直接依赖的任务，它花费的时间量当然为0，其反拓扑数为$n+1$。如此以来，关键路径序列的末点总是结点$done$，即总有

$eft(n+1)$最大。可以通过深度优先检索框架来计算$eft(v)$和关键路径。

算法需增添的数据结构：

数组$d(n+1)$：每个任务花费的时间量。

数组$eft(n+1)$：存放每个任务的最早完成时间。

数组$critDep(n+1)$：存放每个任务的关键依赖任务。

结点v_{end}：表示关键路径上的最后一个任务。

算法 7.7 求解关键路径

Procedure

DfsSweep_CritP($n, adj, color, d, eft, critDep, v_{end}$)

Allocate $color$ array and initialize to $white$.

For each vertex v of G, in some order:

If($color(v) = white$) then

call Dfs_CritP($adj, color, v, d, eft, critDep$)

endif

repeat

Find out the vertex v_{end} such that $eft(v_{end})$ is maximum .

end DfsSweepCritP

procedure Dfs_CritP($adj, color, v, d, eft, critDep$)

int w

int $temadj$

$color(v) \leftarrow gray$

$est \leftarrow 0$； $critDep(v) \leftarrow -1$

//Preorder processing of vertex v//

$temadj \leftarrow adj(v)$

while($temadj \neq nil$) do

$w \leftarrow first(temadj)$

if($color(w) = white$) then

call Dfs_CritP($adj, color, w, d, eft, critDep$)

endif

if($eft(w) \geq est$) then

{ $est \leftarrow eft(w)$, $critDep(v) \leftarrow w$ }

endif

$temadj \leftarrow rest(temadj)$

repeat

$eft(v) \leftarrow est + d(v)$

$color(v) \leftarrow black$

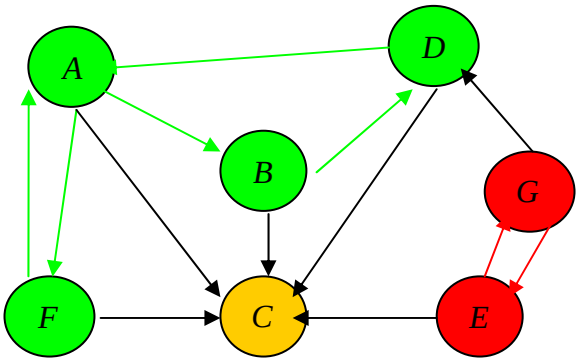
end Dfs_CritP

我们只是论述了有向反圈的一些基本的东西，有向反圈还有许多应用。下一节，我们将了解到，任何有向图G都相伴着一个反圈：G的收缩图（condensation graph），所以反圈的应用问题可延伸到含圈的有向图。

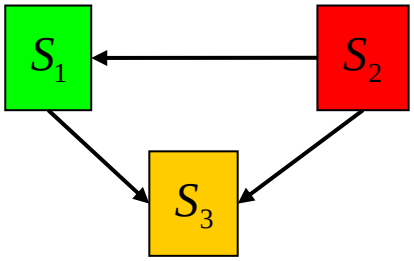
5．有向图的强连通分支

收缩图：设$S_1,S_2,\cdots,S_p$是有向图$G=(V,E)$的强连通分支。G的收缩图$G\downarrow$定义为有向图$G\downarrow=(V\downarrow,E\downarrow)$，其中$V\downarrow$含有$p$个元素$s_1,s_2,\cdots,s_p$，$s_is_j\in E\downarrow$当且仅当$E$中有一条从$S_i$到$S_j$的边。$G\downarrow$相当于对$G$作如下操作得到的图：$G$的每个强连通分支收缩为一点，每两个强连通分支之间的所有边化为所对应两个收缩点之间的一条边。如果不考虑边的方向，有向图与其转置图具有相同的强连通分支，和相同的转置图，即有：$(G^T)\downarrow=(G\downarrow)^T$。

举例



三个强连通分支



收缩图

如果我们能够确定有向图G的所有强连通分支，G的收缩图就很容易得到。我们将要用深度优先检索方法来解决这个问题。

为此，我们来看深度优先检索树与强连通分支之间的关系。

强连通分支的 leader：给定有 p 个强连通分支 $S_1, S_2, \cdots, S_p$ 的有向图 G，在对 G 所作的深度优先周游过程中，S_i（$1 \le i \le p$）中首先被发现的结点称为 S_i 的 leader，记作 v_i。

假如深度优先周游从 v_1（强连通分支 S_1 的 leader）开始，即 v_1 是深度优先检索树的根。则由白道定理，S_1 中所有结点在该深度优先检索树中都是 v_1 的子孙。而且，如果能到达任何强连通分支 S_j（首先被发现的是 v_j），则在 v_j 被发现时应用白道定理，我们看到 S_j 中所有结点都在该树上。随后其他的深度优先检索树也同此理。这证明了下面引理。

引理 7.6 有向图 G 的任何深度优先检索森林的每一棵树由一个或多个强连通分支所组成，（即任何强连通分支的结点都同属于某棵树）。

性质 7.7 leader v_i 是 S_i 中最后完成的结点。（leader 冲锋在前，撤退在后）

那么是否存在一种（结点的）被检索顺序，使得每一棵树恰好组成一个强连通分支？这是我们求解强连通分支的基本思路。从收缩图可以清楚地看到，如果一个强连通分支没有发射出来的边(arrows coming out from it), 则以它的任何结点为起始点的深度优先检索恰好遍历该分支。但是这个事实又有什么实际意义呢？。下面通过考察 leader 的进一步性质，我们就可以利用这个事实得到求解算法。

尽管我们不知道强连通分支或 leaders，我们还是能够得到一些结论。假如从强连通分支 S_i 到 S_j 存在一条边，意味着 G 中有一条从

v_i到v_j的道路。如果v_i比v_j被发现得早，则在包含v_i的深度优先检索树中，v_j是v_i的子孙，此时，$active(v_j) \subset active(v_i)$。反之，如果$v_j$比$v_i$被发现得早，显然，$v_j$不可能是$v_i$的祖先，否则它们将处于同一个强连通分支。

引理 7.8 在深度优先检索中，当 leader v_i被发现时，不存在从v_i到任何灰（gray）结点x的道路

证明：此时，任何灰结点都是v_i的真祖先，且被发现在v_i之前，所以必定在其它的强连通分支中。既然已有从x到v_i的道路，就不能再有从v_i到x的道路。

引理 7.9 如果v是某强连通分支S的 leader，x是另一强连通分支中的结点，且G中存在从v到x的道路，则在v被发现之时，要么x是黑结点，要么存在从v到x的白路（x是白结点），在两种情况下，x都比v完成得早。

证明：v被发现之时，考察任何从v到x的道路，如果该道是白道，所证成立。否则，设z是该道上最后一个非白结点，由引理 7.8，z必是黑结点。如果$z \neq x$，则回顾z被发现时，z到x这段是白道，由白道定理，x是z的子孙，x必在z变黑之前变成黑结点，所以x也是黑结点，这与z是该道上最后一个非白结点相矛盾，所以$z = x$。总之，x是黑结点。

所以，如果从强连通分支S_i到S_j存在一条边，我们排除了v_j的活动时间段整个在v_i的活动时间段后面的可能性。如果不要求v_i是 leader 的话，可找到引理的反例。

求强连通分支的算法

我们将利用上述性质得到求强连通分支的算法。第一个线性复杂度的算法归功于 R.E.Tarjan, 基于深度优先检索。我们要叙述的算法是属于 M. Sharir 的，也基于深度优先检索。它简单精微而颇显雅致。

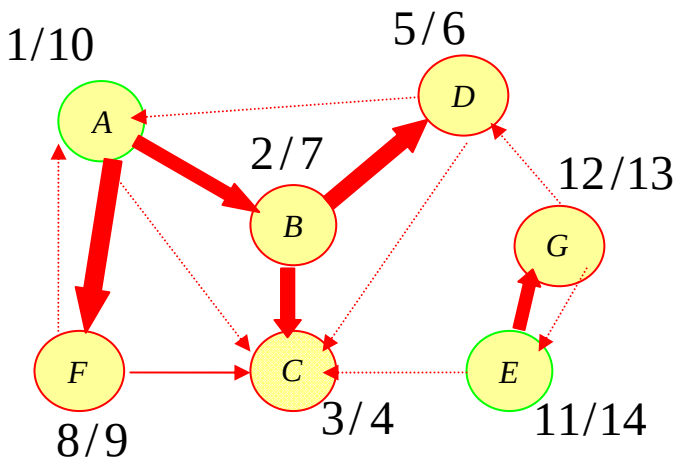
算法包括两大步：

1．对G作深度优先周游，G的所有结点按其完成时间被堆放在一个栈中；

2．对G^T作深度优先周游，只是需要按结点出栈的顺序确定每一次新的检索的起点。每一次检索恰好得到一个G^T的强连通分支。

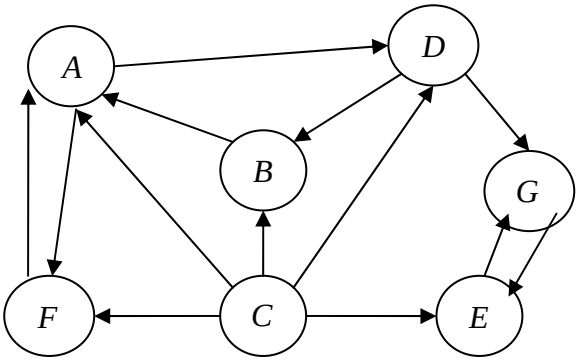
注：算法将启用一个数组$scc(n)$以存放每个结点所在强连通分支的leader。

举例

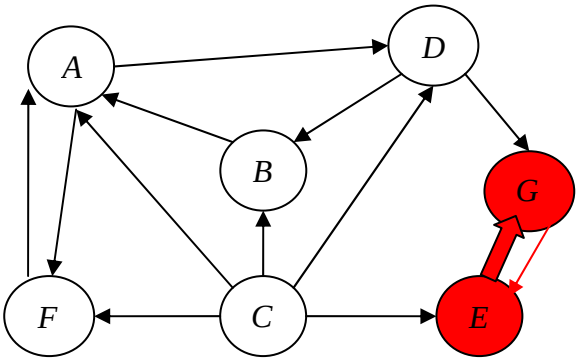


E
G
A
F
B
D
C

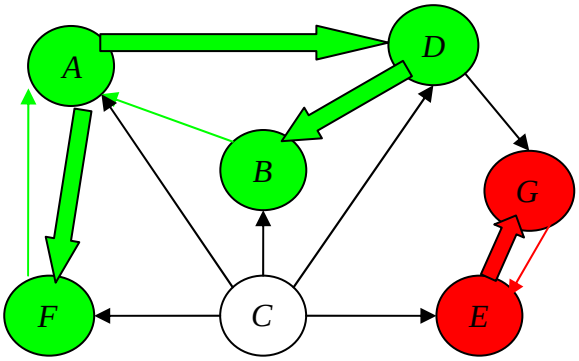
完成对G的深度优先周游后，栈的情形



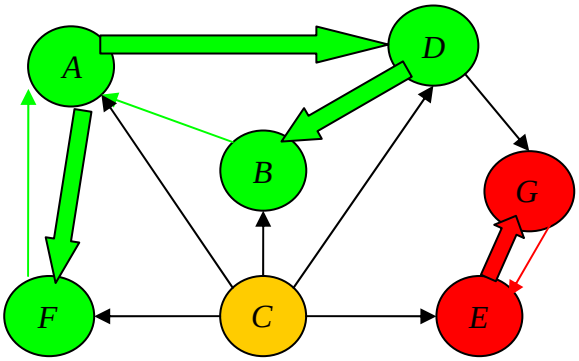
转置图



E 为起点深度优先树



A 为起点深度优先树



C 为起点深度优先树

算法 7.8 求强连通分支

procedure StrongComponents(adj, n, scc)

1. create $finishstack$;
2. perform a depth-first search on G, using the DFS skeleton. At postorder processing for vertex v, insert the statement: $push(finishstack, v)$;
3. Compute G^T, the transpose graph, represented as the array $Transadj$ of adjacency lists.
4. DfsTSweep($transadj, n, finishstack, scc$)

end StrongComponents

procedure DfsTSweep($transadj, n, finishstack, scc$)

//DfsSweep on transpose graph//

Allocate $color$ array and initialize to $white$.

While($finishstack$ is not empty) do

$v \leftarrow top(finishstack)$

$pop(finishstack)$

if($color(v) = white$)then

call DfsT($transadj, color, v, v, scc$) endif

repeat

end DfsTSweep

procedure DfsT($transadj, color, v, leader, scc$)

Use the standard depth-first search skeleton. At preorder processing for Vertex v, insert the statement:

$scc(v) \leftarrow leader$

pass $leader$ and scc into recursive calls

end DfsT

引理 7.10 算法的第二大步中，每当从栈中弹出一个白结点时，该结点必是第一大步中某强连通分支的 leader。

证明：弹出结点的顺序是逆着第一步深度优先周游中结点的完成顺序，即后完成的结点先弹出。而 leader 总是其所在强连通分支中最后完成的结点。所以当非 leader 结点 X 被弹出时，因为其 leader 结点先于它被弹出，故 X 不可能是白结点。

定理 7.11 算法的第二大步中，每个深度优先检索树恰好构成一个强连通分支。

证明：一般说来，每个深度优先检索树由若干个强连通分支构成。我

们需要说明的是，在这种情况下，正好是一个。设v_i是第一步中某S_i的 leader, 假设v_i从栈中被弹出时是白结点，则v_i是一棵深度优先检索树的根，如果v_i在G^T不能到达其它强连通分支，则结论成立毫无问题。假如v_i在G^T能到达某强连通分支S_j(其 leader 是v_j)，则在G中存在从v_j到v_i的路径，由引理 7.9, 在第一步的周游中v_j完成在v_i之后，所以在第二大步的周游中当v_i从栈中被弹出时，v_j已被弹出且S_j中所有结点都已成为黑结点。故从v_i出发的检索不会跑出S_i。

定理 7.12 算法*StrongComponents*正确地求出了G^T的所有强连通分支。

注：算法*StrongComponents*的时间复杂度与空间复杂度都是$\Theta(n+m)$。

6. 无向图上的深度优先检索

无向图上的深度优先检索的基本思路与有向图的情况一样：尽可能“探索”，必要时回溯。所以可沿用有向图上深度优先检索算法的框架，及其一些主要术语：vertex color, discovery times, finishing times, DFS trees. 但无向图上的深度优先检索更为复杂，其原因在于每条边应该只被“探索”或“检查”一次，但在数据结构中每条边有两种表示。

有些无向图的问题每条边被处理两次并无影响，如求连通分支，所以可把无向图当作对称有向图。这节涉及的问题作这种简化行不

通。原则上讲，无向图牵扯到回路的问题就要求每条边只能被处理一次。

对于一个无向图来说，深度优先检索给它的每条边标定了一个方向，这取决于边在哪个端点处首次被遇到，则以该端点为起点定义边的方向。同有向图一样，被发现的边称为树枝，被检查的边称为非树枝边。

当一个结点在其数据结构(邻接表，或邻接矩阵)中遇到已标定方向的一条边（其定向指向该结点）时，不再作任何处理，因为该边已被处理过。DFS 框架可以稍加修改，使其能够辨认这种情形。对于无向图，非树枝边有下列事实：

1.不会出现交叉边。

2.如果vw是向后边，w必是灰结点，且w不是v之父亲。

3.向前边是无向边的第二次出现，故不必处理。因为vw作为向前边被发现时，vw已经作为结点w处的向后边被处理过了。实质上说，没有向前边。

所以，无向图只有树枝和向后边。

上述事实促使我们对深度优先检索框架作下面修改。首先，Dfs 将增添检索起点v的父结点 p 为参数，这样使得在处理边vw时，如果w是非白点，则来作进一步的检测。如果w是灰结点且其不同于v的父结点 p，则边vw第一次被遇到，是向后边；否则，vw已作为向后边或树枝（wv）被处理过了。

算法 7.9 无向图的深度优先检索

procedure Dfs$(n, adj, color, v, p, ans\cdots)$

int w

int $temadj$

int $wAns$

1. $color(v) \leftarrow gray$

2. Preorder processing of vertex v

3. $temadj \leftarrow adj(v)$

4. while($temadj \neq nil$) do

5. $w \leftarrow first(temadj)$

6. if($color(w) = white$) then

7. { Exploratory processing for tree edge vw

8. call Dfs$(n, adj, color, w, v, wAns, \cdots)$

9. Backtrack processing for tree edge vw, using $wAns$ (like inorder)}

10. else if ($color(w) = gray \&\& w \neq p$) then

11. { Checking (i.e. processing) back edge vw}

//else wv was traversed , so ignore vw. //

endif

endif

12. $temadj \leftarrow rest(temadj)$

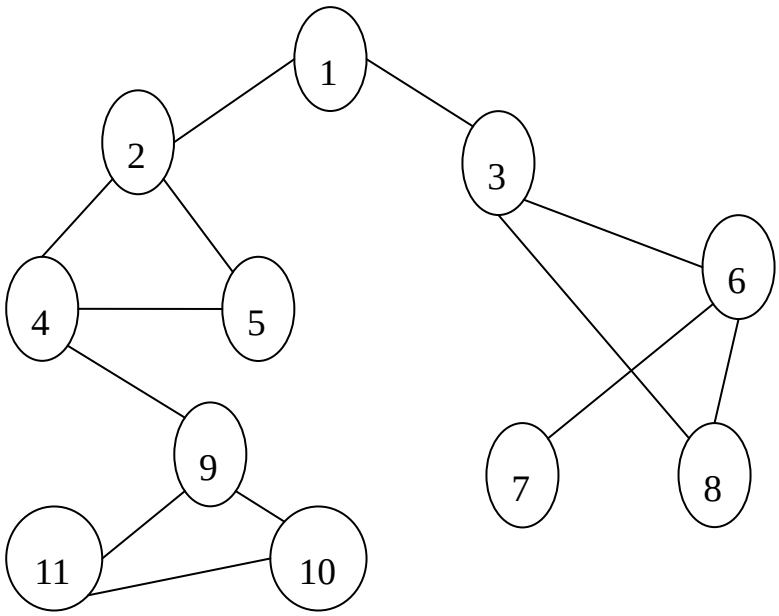
repeat

13. Postorder processing of vertex v, including final computation of ans

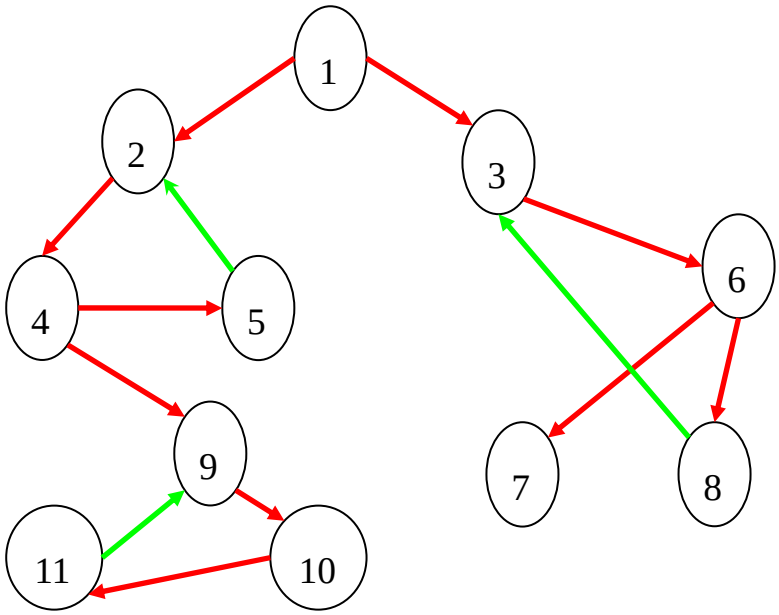
14. $color(v) \leftarrow black$

end Dfs

时间复杂度$\Theta(n+m)$，额外的空间开销$\Theta(n)$。



无向图示例



深度优先检索：红色边是树枝，绿色边是向后边

无向图宽度优先检索

问题可简化为：将无向图表示成对称有向图，采用有向图的深度优先检索。

7．无向图的双连通分支

背景问题举例：

1. 在交通图中，一个站点被毁坏，其它站点能否仍然连通？

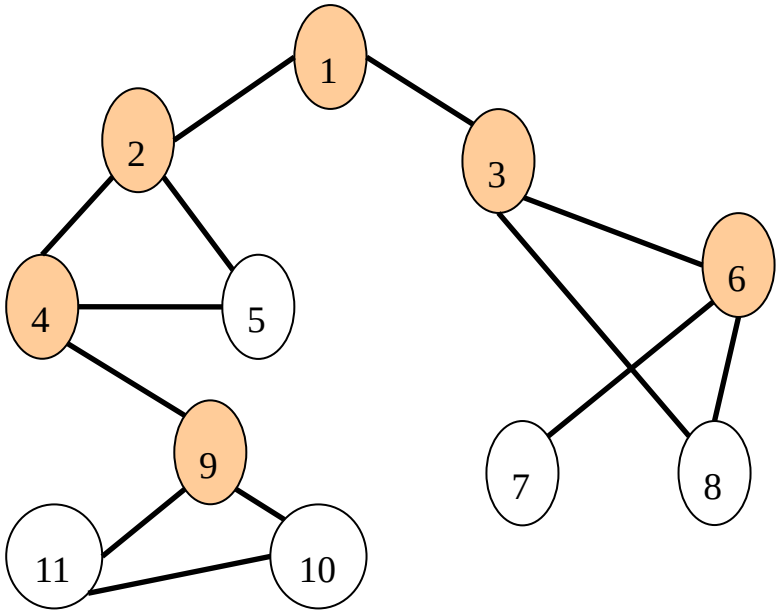
2. 计算机网络中，一台计算机除了故障，其他计算机能否继续保持通讯？

问题：删去连通无向图的一个结点（及其与它关联的边），所留下来的图还连通吗？

这个问题在表示通讯或运输网络的图中，是一个很重要的问题。找出能造成不连通的结点（如果删去它）也是很重要的。本节的目的是介绍回答这些问题的一个有效算法。这个算法是由 R.E.Tarjan 发现的，是佐证深度优先检索强大威力的早期算法之一。

关节点（断点，articulation point，cut point）与双连通分支

无向图G的结点u称为**关节点**（或**断点**），如果存在两个不同结点v和w，使得u在所有v和w之间的道路上。



茶色结点是关节点

显然，删去一个关节点就会导致一个不连通图。所以一个连通图称为是双连通的，如果它不包含任何关节点。双连通分支给出图边集的一个划分。可以在G的边集E上定义一个关系二元～：$e_1 \sim e_2$当且仅当$e_1 = e_2$或e_1和e_2都在某个回路上。 可验证～是等价关系，其每个等价类对应着G的一个双连通分支。

我们将给出的求解双连通分支的算法利用无向深度优先检索算法框架，主要基于深度优先检索树的**树枝必是向后边**的这一特征。在检索进行的过程中，所需要的信息被计算，图的边集及其关联的结点按其所在双连通分支而被划分存储。 那么需要什么样的信息？如何确定一个双连通分支？因为每个回路至少含有一条向后边，所以需要研究向后边与双连通分支之间的关系

双连通分支算法

在深度优先检索过程中，结点可以在三种时刻进行处理：先序（它被发现时），中序（每个子结点回溯到它时），后序（它将要完成时）。双连通分支算法对深度优先检索树中的每个结点在它回溯时进行检测，以断定它是不是关节点。再重申一下，对深度优先检索树（本节简称为“树“）来讲，所有边都是树枝或向后边。

假设检索从结点W回溯到V时，不存在从W为根的子树中的某点到V的某个真祖先的向后边，则V一定在从树的根到W的每条道路上，所以是关节点。W为根的子树及其中的向后边再加上边VW将与图的剩余部分在关节点V分离开来，但它未必是一个双连通分支，它可能由几个连通分支所组成。我们将确保每个双连通分支被恰当地分离，这可以通过每发现一个双连通分支就移走来做到。当W回溯到V时，W为根的子树中含有的其它双连通分支已被剥离，剩下的都在同一个分支中。

定理 7.13 在深度优先检索树中，对任何结点v，

1. 如果v是根结点，则v是关节点当且仅当v有不少于两棵子树；

2. 如果v不是根结点，则v是关节点当且仅当，v不是外结点且v的某棵子树没有与v的真祖先关联的向后边。

对每个结点v，它有被发现的时间$dis\cov erTime(v)$，我们知道，当v是w的真祖先时，$dis\cov erTime(v) < dis\cov erTime(w)$，以$back(v)$定义以$v$为根的子树通过向后边可到达的最早的祖先的被发现时间，则

1 $back(v) = \min\{dis\cov erTime(v),$

$\min\limits_{vw\text{是向后边}} dis\cov erTime(w),$

$\min\limits_{w\text{是}v\text{的儿子}} back(w)\}$

2 对对任何内结点v来说，v是关节点当且仅当存在v的某儿子w，使得$dis\cov erTime(v) \le back(w)$。

算法 7.10 求解无向图的双连通分支

procedure bicomponents(adj, n)

int v, $color(n)$

Initialize $color$ array to $white$ for all vertices

$time \leftarrow 0$

$edgestack \leftarrow create()$

for $v \leftarrow 1$ to n do

if($color(v) = white$) then

call bicompDFS($adj, color, v, -1, back$) endif

repeat

end bicomponents

procedure bicompDFS($adj, color, v, p, back$)

int w

IntList $remAdj$

1. $color(v) \leftarrow gray$

2a. $time++$; $discoverTime(v) \leftarrow time$

2b. $back \leftarrow discoverTime(v)$

3. $remAdj \leftarrow adj(v)$

4. while($remAdj \neq nil$) do

5. $w \leftarrow first(remAdj)$

6. if($color(w) = white$) then

{

$push(edgestack, vw)$

call bicompDFS($adj, color, w, v, wback$)

//backtrack processing of tree edge vw//

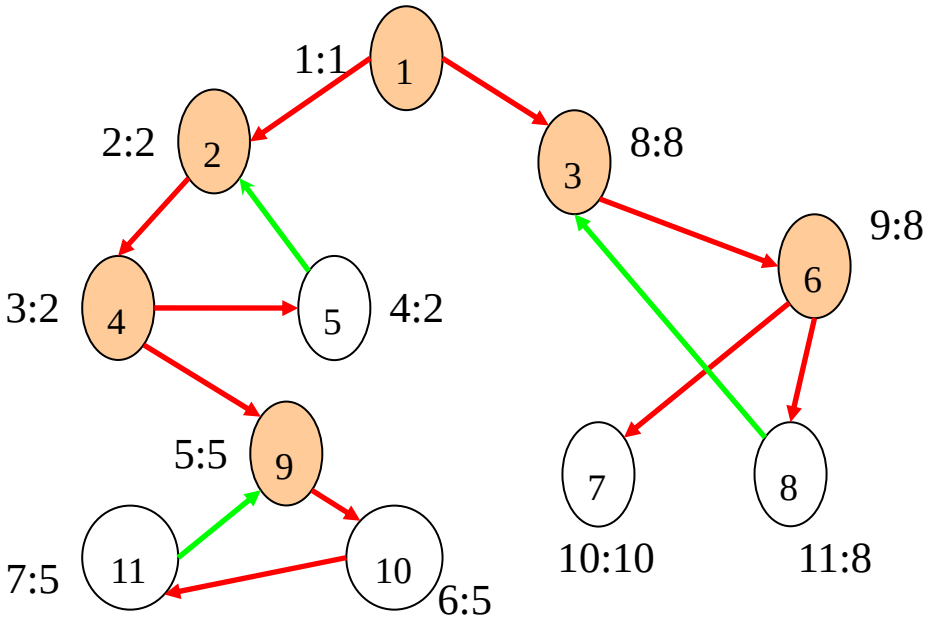
if($wback \geq discoverTime(v)$) then

{

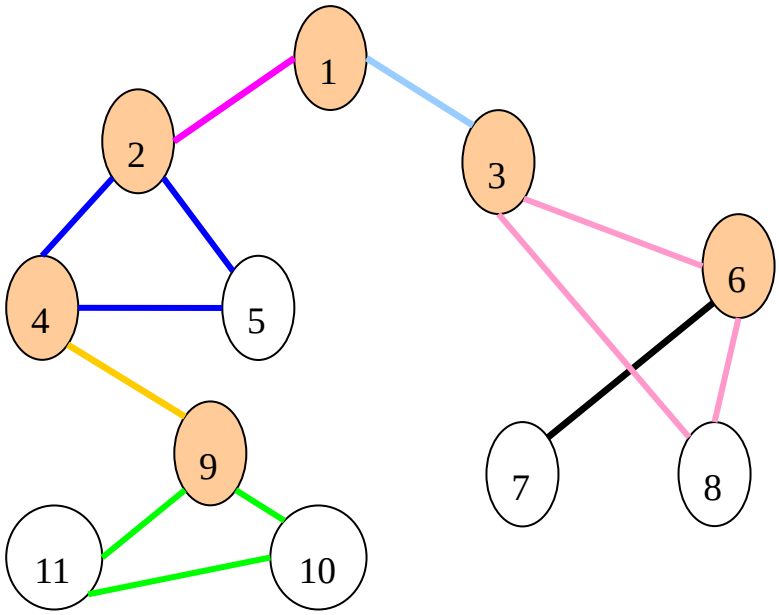
Initialize for new bicomponent

Pop and output $edgestack$ down through vw

```
            }
        else back ← min(back, wback)
        endif
      }
    else
     {
        if(color(w) = gray && w ≠ p) then
          //process back edge  vw//
         {
           push(edgestack, vw)
           back ← min{back, discoverTime(w)}
         }
        //else wv was traversed , so ignore  vw//
        endif
      }
    endif
    remAdj ← rest(remAdj)
repeat
  color(v) ← black
end bicompDFS
```



discoverTime : back 示意图



双连通分支

作业

1.请编制算法，从图G的邻接表得到它的转置图G^t的邻接表，要求时间复杂度为$O(n+m)$。假设结点用n个数字编号，每个链表中，链结点的顺序由所表示的外邻点的编号决定（从小到大）。

2.请编制算法来判定有向图G是否反圈，时间复杂度限定为$\Theta(n+m)$，假定G表示为邻接表。

3.一个有向反圈G称为一个格，如果有一个结点s可以到达所有其他结点，另有一个结点t能被其他结点所到达。假设有向图G用邻接表表示，请编制算法来判定图G是否格，要求时间复杂度为$\Theta(n+m)$。

4.请编制算法判定无向图G是否二部图。图G用邻接表表示， 要求算法复杂度为$O(n+m)$。

5.称s是有向图G的超汇点，如果对G的任何其他结点v来说，都有vs是G的边，而sv不是G的边。请给出一个算法判定有向图G是否有超汇点，在有向图G用邻接矩阵表示的情况下，要求算法最坏时间复杂度为$O(n)$。

6.$F(i,j)$（$i,j\geq 0$， $i+j\geq 1$）满足下列递归式

$$\begin{cases} F(i,j)=pF(i-1,j)+qF(i,j-1), \\ F(i,0)=a_i,\ F(0,j)=b_j, \\ i\geq 1,\ j\geq 1, \end{cases} ,$$

请设计算法求解$F(n,n)$，要求算法最坏时间复杂度为$O(n^2)$。并请尝试给出$F(i,j)$的一个表达式。

7.有n台相互联网的计算机，分别标记为$c_1,c_2,\cdots,c_n$，某种病毒在时

刻x感染了其中的一台计算机c_a，问在时刻y（$y>x$）计算机c_b有没有感染这种病毒。已知数据为m个三元组(c_i,c_j,t_k)，(c_i,c_j,t_k)表示计算机c_i与计算机c_j在时刻t_k进行过数据交换；如果计算机c_i（计算机c_j）在时刻t_k前（包括在时刻t_k）感染这种病毒，那么在时刻t_k计算机c_j（计算机c_i）也会感染这种病毒。假设任两个计算机至多进行过一次数据交换，m个三元组(c_i,c_j,t_k)分别在数据交换发生的时刻t_k报告给你，即m个三元组是以时刻的非递减序列被提供的。请编制算法解决该问题。要求算法时间复杂度为$O(n+m)$。