

订阅DeepL Pro以编辑此演示文稿。  
访问[www.DeepL.com/pro](https://www.deepl.com/pro?cta=edit-document)，了解更多信息。

# 决定定向无环图

如何决定一个给定的有向图是否是DAG？下面是DFS的变体（有计时），给出了一个算法。具体来说，当算法检查一条边(*vi, v j*)时：如果*v*已经*j*被探索过了*，而且*它的后位数还没有被设置，那么算法就会报告说*G*不是DAG。

函数DFS (*G* = (*V, E*))

*clock*=1。

*visited*[*i*] = 0*, pre*[*i*] = -1*, post*[*i*] = - 1,for ≤ 1*i* ≤ |*V* |; for *i* = →1 |*V* |

如果（*visited*[*i*] = 0）探索（*G, vj* ）。

end if; end for;

结束算法。

函数探索 (*G* = (*V, E*)*, vi*∈*V* )

*visited*[*i*] = 1; *pre*[*i*] = *clock*; *clock* + +;

对于任何边（*vi，vj* ）∈*E*

如果（*visited*[ *j*] = 0）：探索（*G，vj*）。

else if (*post*[ *j*] = -1): report "*G* is not a DAG"; end for;

*post*[*i*] = *clock*; *clock* + +;

结束算法。

现在我们来证明这个算法是正确的。我们首先证明，如果*G*不是一个DAG，那么该算法总是会在某个时间给出该报告。假设*G*包含一个循环*C*，因为它不是一个DAG。让*v Cj*是*C*中第一个被探索的顶点，让(*vi, vj* )*E*是*C*中的一条边，由于*v*可以*j*到达*vi*，在探索*v的*过程中，*j*会有一个时间，*vi*会被探索到。考虑在探索*v的*过程中检查边（*vi，vj*）的时间*i*：此时*visited*[*j*]已被设置为1，但其后数还没有被设置，因为现在算法还在探索*vj*的过程中。因此，算法会报告说*G*不是一个DAG。

∈

∈

然后我们证明，如果算法给出该报告，那么*G*确实不是一个DAG。考虑到算法正在探索*vi*，检查边（*vi，vj*），发现*visited*[*j*] = *和*1*post*[*j*] =，*post*[*j*]还没有被设置，1.这意味着算法是在探索*vj*的范围内。所以我们有*v*可以*j*到达*vi*，因为现在我们正在探索*vi*。此外，存在边（*vi，vj*）。综合起来，*G*包含循环。

-

请注意，这个算法本质上是确定是否存在边（*vi，vj* ）*E*，使得区间[*pre*[*i*]*，post*[*i]*]在区间[*pre*[ *j*]*，post*[ *j*]]之内。(这样的边在教科书[DPV]第95页中被称为*后边*）。

∈

# 寻找一个DAG的线性化

DFS-with-timing也可以用来寻找一个DAG的线性化：我们只需在给定的DAG *G*上运行DFS-with-timing并得到postlist（按post值递减的顶点列表）；postlist将是*G*的线性化。

为什么呢？首先，观察一下，DAG *G*的每个连接部分都正好包含一个顶点，也就是说，DAG *G*中的每个顶点都形成了自己的连接部分。(你能用图1发现这一点吗？)这是因为，如果一个连接部分至少包含两个顶点*u*和*v*，那么*u*可以到达*v*，*v*也可以到达*u*，所以一定存在一个循环。因此，任何DAG *GM*的元图*G*也是它自己，也就是说，*G* = *GM*。

现在让我们在DAG的背景下解释一下我们在A11讲座中得到的关于一般有向图的结论。考虑A11讲座中1的主张：*让Ci和Cj是有向图G*=（*V，E*）*的两个连接部分，即Ci和Cj是其核心对应的元图GM*=（*VM，EM*）*中的两个顶点。如果我们有*(*C,iCj*)*EM，那么我们必须有maxiu*∈*C post*[*u*] *> v*∈*Cmaxj post*[*v*]*。*由于在DAG中，组件*Ci*和*Cj*将退化为两个顶点，例如*vi*和*vj*，其元图*GM*与*G*相同，我们可以将DAG的这一要求转化为：如果在DAG中我们有(*vi, v j*)*E*，那么我们必须有*post*[*i*] *> post*[ *j*]。根据线性化的定义和postlist的定义，这就立即得到了所需的结论：postlist是*G*的线性化。

∈

∈

*v*2

*v*3



*v*4

*v*5



*vv*1 1

(1*,*12 )

*v*6

*v*2 (2*,*9 )

*v*5

(3*,*4 )

*v*3

*v*4

(5*,*8 )

*v*

(10*,*11 )

6

(6*,*7 )

图1：在DAG *G*上运行DFS的例子（带计时）。每个顶点的[*pre, post*]间隔被标记在每个顶点旁边。这次运行的*后列表*是（*v*1*, v*5*, v*2*, v*4*, v, v*63），它是*G*的线性化。

# 排队

一个*队列*数据结构支持以下四种操作。

1. empty (*Q*)：决定队列*Q*是否为空。
2. insert (*Q*, *x*): 向*Q*添加元素*x*。
3. find-earliest (*Q)*: 返回*Q*中最早加入的元素。
4. delete-earliest (*Q)*: 删除*Q*中最早加入的元素。

为了实现上述操作，我们可以使用一个（动态）数组*S*来存储所有的元素，并使用两个指针，*头部*和*尾部*，其中*头部指针*总是指向*S*中的第一个可用空间，而*尾部*指针总是指向*S*中最早添加的元素，当我们向*S*中添加一个元素时，我们可以直接将其添加到*头部*指向的地方，而当我们删除最早添加的元素时，我们可以直接删除*尾部*指向的那个。

尾巴

头

10

5

8

4

尾巴

插入（Q，2）。

头

删除-最早的 (Q)

尾巴 头

2

10

5

8

2

10

5

8

4

图2：队列的一个例子。

函数empty(*Q*)

如果*头*=*尾*：返回真；否则：返回假。

结束功能。

函数 insert(*Q*, *x*) *S*[*head*] = *x*; head = head + 1;

结束功能。

函数 find-earliest(*Q*) 返回 *S*[*tail*]。

结束功能。

函数 delete-earliest(*Q*) tail = tail + 1;

结束功能。

注意，队列数据结构表现出先入先出的特性（而堆栈是先入后出）。

# 优先队列

在一个优先级队列中，每个元素都与一个*优先级*（也称为键）相关。换句话说，优先级队列中的每个元素都是一对（*键，值*），其中*键*表示其优先级，*而值*存储实际数据。一个*优先级队列*数据结构支持以下操作。

1. empty (*PQ*)：决定优先级队列*PQ*是否为空。
2. insert (*PQ*, *x*): 向*PQ*添加元素*x*。
3. find-min (*PQ*)：返回*PQ*中键值最小的元素（即优先级最高）。
4. delete-min (*PQ*)：删除*PQ*中键值最小的元素（即优先级最高）。
5. decrease-key (*PQ*, pointer-to-an-element, new-key)：将指定元素的键设置为给定的新键。

请注意，队列可以被看作是优先权的一个特例，对它来说，优先权是指一个元素被添加到队列中的时间。

优先级队列有许多不同的实现方式（查看维基百科）。这里我们介绍其中的一个，即*二进制堆*。要做到这一点，首先让我们正式介绍一下*堆*。

*堆*是一个（有根的）树状数据结构，满足*堆*的*属性*。如果一个堆满足*min-heap属性*，那么它就是min*-he*ap：对于（有根）树*T*中的任何一条边（*u，v*），*u*的键都小于*v*的键；如果一个堆满足*max-heap属性，那么*它就是max*-heap*：对于（有根）树*T*中的任何一条边（*u，v*），*u*的键都大于*v*的。

10



5

6

9

8

7

11

12

15

22

图3：堆的一个例子。一个元素的关键是在顶点的旁边

*二进制堆*是一个堆，其树是*完整的二进制树*。一棵*完整的二叉树*是一棵二叉树（即每个顶点至多有2个孩子），并且在树的每一层，可能除了最后一层，都是完全填充的，最后一层的所有顶点都是从左到右放置的。

22



5

10

9

12

14

11

15 13 20

图4：二进制堆的一个例子。

由于二元堆*T*是如此有规律，我们可以使用数组*S*来存储其元素（而不是使用邻接列表）。*T*的根（即第0层）被放在*S*[1]中（我们假设*S*的索引从1开始），第1层的第一个元素被放在*S*[2]中，以此类推。一般来说，*T*的第*i*层中的第*j*个元素将被放在*S*[2*i*+*j* -1]中。我们也可以很容易地访问一个元素的父和子。

1. *S*[*k*]的父元素是*S*[*k/*]2（*k*/2的底层）。
2. *S*[*k*]的左子是*S*[2*k*]；*S*[*k*]的右子是*S*[2*k*+1]。

我们现在介绍两个用于实现二元堆的常用程序。这些程序适用于一个顶点违反堆属性的情况，它们可以调整堆，使其满足堆属性。

当一个顶点的键比它的父顶点小时，*bubble-up*函数适用。 函数bubble-up (*S*, *k*)

*p* = *k*/2。

如果（*S*[*k*]*.key < S*[*p*]*.key*）；交换*S*[*p*]和*S*[*k*]。

bubble-up (*S*, *p*); end if;

结束功能。

222222



5

10

9

12

14

11



5

10

9

8

14

11



5

8

9

10

14

11

15 8 20

15 12 20

15 12 20

图5：说明起泡程序。

当一个顶点的键值大于它的子节点时，*筛下*函数适用。 函数 筛下 (*S*, *k*)

*c* = argmin*t*∈{2*k,*2*k*+1} *S*[*t*]*. key*是*S*[*k*]中键值较小的孩子的索引。

如果（*S*[*k*]*.key>S*[*c*]*.key*）；交换*S*[*c*]和*S*[*k*]。

sift-down (*S*, *c)*; end if;

结束功能。

222222



14

9

10

12

17

16



9

14

10

12

17

16



9

12

10

14

17

16

151820

15 18 20

15 18 20

图6：说明筛下程序。