

# 算法设计与分析作业四

作者：吴润泽      学号：181860109

**Email:** [181860109@smail.nju.edu.cn](mailto:181860109@smail.nju.edu.cn)

2020 年 4 月 16 日

## 目录

<b>Chapter 4</b>	<b>3</b>
problem 4.2 . . . . .	3
problem 4.5 . . . . .	4
problem 4.7 . . . . .	5
problem 4.8 . . . . .	5
problem 4.9 . . . . .	5
problem 4.12 . . . . .	6
problem 4.13 . . . . .	7
problem 4.14 . . . . .	9
problem 4.16 . . . . .	11
problem 4.17 . . . . .	12
problem 4.18 . . . . .	13
problem 4.20 . . . . .	15
problem 4.22 . . . . .	16
problem 4.23 . . . . .	16
 <b>Chapter 5</b>	 <b>19</b>
problem 5.1 . . . . .	19
problem 5.2 . . . . .	19
problem 5.4 . . . . .	20
problem 5.8 . . . . .	21
problem 5.9 . . . . .	25
problem 5.10 . . . . .	26

## Chapter 4

### problem 4.2

(1)

$\Rightarrow$  如果  $w$  是  $v$  在 DFS 树中的后继结点, 那么  $actice(w) \subseteq active(v)$ :  
 当  $w \neq v$  时, 因为  $w$  是  $v$  的后继结点, 所以  $v.discover < w.discover$ , 并且  $v.finish > w.finish$ 。所以  $actice(w) \subset active(v)$ 。当  $w = v$  时显然成立。  
 $\Leftarrow$  如果  $actice(w) \subseteq active(v)$ , 那么  $w$  是  $v$  在 DFS 树中的后继结点:  
 当  $w \neq v$  时, 因为  $active(w) \subseteq active(v)$ , 即  $v.discover < w.discover$ , 并且  $v.finish > w.finish$ , 即在遍历  $v$  的过程中将  $w$  遍历, 即  $w$  是  $v$  的后继结点。

(2)

由 (1) 可知,  $w$  不是  $v$  的后继结点  $\Leftrightarrow actice(w) \not\subseteq active(v)$ 。 $v$  不是  $w$  的后继结点  $\Leftrightarrow active(v) \not\subseteq active(w)$  得证。

(3)

①  $\Rightarrow$  如果  $vw$  是 CE, 那么  $v$  和  $w$  没有祖先和后继关系, 由 (2) 可知  $active(w)$  和  $active(v)$  互不包含。同时 CE 说明在  $v$  指向  $w$  时,  $w$  已经是黑色结点,  $w$  已经遍历结束, 所以  $active(w)$  在  $active(v)$  之前。

$\Leftarrow$   $active(w)$  在  $active(v)$  之前,  $w$  先完成整个遍历过程, 后才遍历到  $v$ 。且二者没有祖先后继关系, 那么边  $vw$  即为 CE。

②  $\Rightarrow vw$  是 DE, 即  $v$  指向  $w$  时  $w$  为黑色, 并且  $active(w) \subset active(v)$ , 若不存在第三个结点  $x$ , 满足  $x$  是  $v$  的后继,  $w$  是  $x$  的后继, 则  $v$  遍历到  $w$  时  $w$  一定为白色, 边为 TE。即一定有  $active(w) \subset active(x) \subset active(v)$ 。

$\Leftarrow$  如果存在结点  $x$ , 满足  $active(w) \subset active(x) \subset active(v)$ , 由 (1) 可知,  $x$  是  $v$  的后继,  $w$  是  $x$  的后继, 且在遍历时  $v$  先走到  $x$ , 然后  $x$  走到  $w$ , 即  $v$  是  $w$  的祖先结点, 因此  $vw$  是 DE。

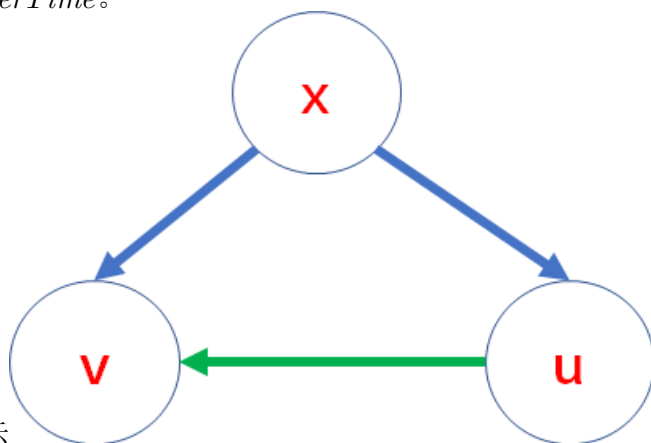
③  $\Rightarrow vw$  是 TE, 即  $v$  指向  $w$  时  $w$  为白色,  $w$  是  $v$  的后继, 由 (1) 可知,  $active(w) \subset active(v)$ 。若存在  $x$ , 满足  $active(w) \subset active(x) \subset active(v)$ , 则在遍历时  $v$  先走到  $x$ , 然后  $x$  走到  $w$ , 那么  $v$  是  $w$  的祖先而非父结点, 与  $vw$  是 TE 矛盾。

$\Leftarrow$  同理可得  $w$  是  $v$  的后继, 且  $v$  直接指向  $w$ , 则  $w$  是白色, 即  $vw$  是 TE。

④  $vw$  是 BE  $\Leftrightarrow v$  是  $w$  的后继  $\Leftrightarrow active(v) \subset active(w)$  得证。

### problem 4.5

1. 不可能是 TE。如果是 TE, 则有  $active(v) \subset active(u)$ , 即  $v.finishTime > u.discoverTime$ , 故不成立。
2. 不可能是 BE。如果是 BE, 则有  $active(u) \subset active(v)$ , 即  $v.finishTime > u.discoverTime$ , 故不成立。
3. 不可能是 DE。如果是 DE, 同样的  $v$  是  $u$  的后继结点, 满足  $active(v) \subset active(u)$ , 同 1. 不成立。
4. 可能是 CE。 $x$  结点先遍历  $v$ , 然后从  $v$  返回  $x$ ,  $x$  遍历  $u$ , 易知  $v.finishTime < u.discoverTime$ 。



如图所示

**problem 4.7**

在第一次 DFS 中，将结点压栈，同一强连通片的源头结点是最后一个压栈的。(引理 4.4) 若  $l$  是某个强连通片首结点， $x$  是另一个强连通片中的结点，并且存在  $l$  通向  $x$  的路径，则  $x$  比  $l$  先结束遍历，即  $x$  先进栈  $l$  后进栈。满足这些性质，才能保证第二次 DFS 中，按正确的顺序取出每个 SCC 的首结点。

无论是 DFS 还是 BFS 在一次遍历中都可以访问一个或者多个强连通片的所有结点。

如果第一次 DFS 换为 BFS，由于 BFS 中按层序遍历，同一连通片中出度不为 0 的点可能先入栈。在第二次 DFS 的时候，不能有正确的访问顺序。

如果第二次 DFS 换为 BFS，由于出栈的访问首结点顺序正确，BFS 同样可以正确地划分强连通片。

因此第一次必须为 DFS，第二次 DFS 和 BFS 都可以。

**problem 4.8**

**充要条件:** 对于无向连通图的 DFS 生成树的根结点  $v$ ， $v$  是割点，当且仅当  $v$  有两个及两个以上的子树。

**证明:**  $\Rightarrow$  如果  $v$  是割点，假设  $v$  只有一个子树，易知子树是连通的，将  $v$  删除，剩下的部分为  $v$  的子树仍然连通，这与  $v$  是割点相矛盾。因此  $v$  有两个及两个以上的子树。

$\Leftarrow$  如果  $v$  有两个及两个以上的子树，因为图本身连通，则子树之间相连必然通过  $v$ ，即  $v$  是割点。

**problem 4.9**

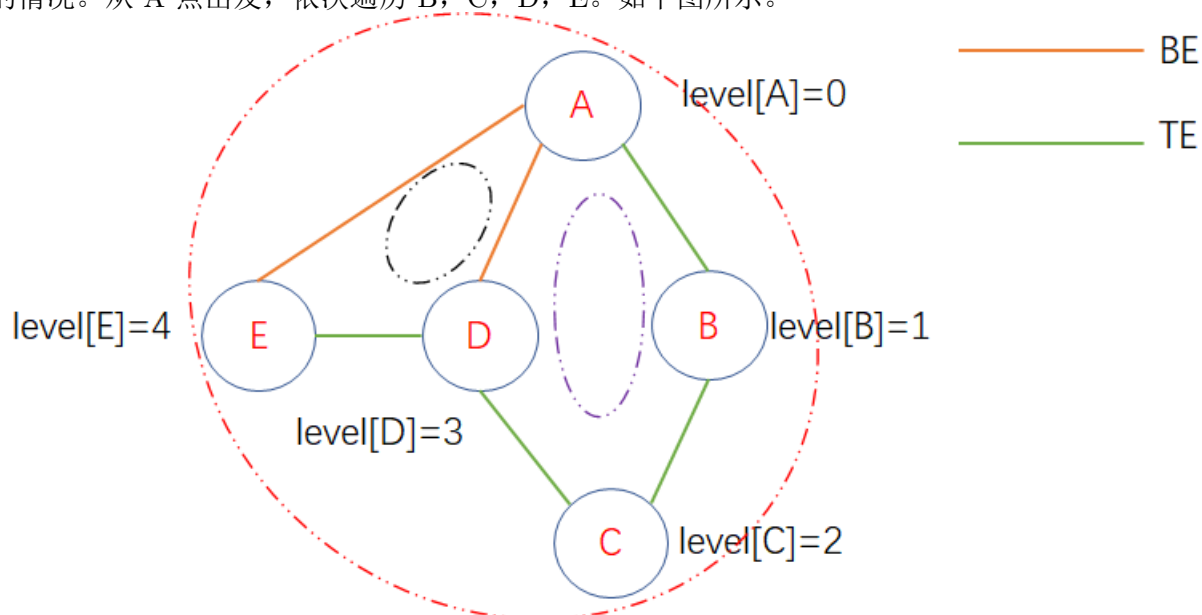
正确

**证明:** 当从 TE  $vw$  回退时，如果以  $w$  为根的子树存在 BE 指向  $v$  的祖先，则  $v$  的祖先的 `discoverTime` 会被赋值给以  $w$  为根的子树中某个结点的 `back` 值，并最终会传递到  $w.back$ ，即必有  $w.back < v.discoverTime$ ;

否则，没有存在 BE 指向  $v$  的祖先，如果子树不存在 BE，则子树所有点 `back` 值均为初始值，即  $w.back > v.discoverTime$ ，如果子树中存在 BE，则子树中的所有结点的 `back` 值也将大于等于  $v.discoverTime$ ，因为 BE 只能指向  $v$  或者子树内部结点，故 `back` 值最小也不会低于  $v.discoverTime$ ，仍然满足  $w.back > v.discoverTime$ ，即仍能正确判断是否为割点。

## problem 4.12

算法中，默认环为多条 TE 和 BE 构成，但存在两条 BE 和 TE 构成环的情况。从 A 点出发，依次遍历 B, C, D, E。如下图所示。



图中的 BE 为 AE 和 AD，根据题目给定算法，计算出外圈大环的大小为 5 和右侧环大小为 4，最终结果为 4。但图中最小环大小应为 3，为左侧小环。因此算法不正确。

**problem 4.13**

**算法设计** 如果有向图没有环，则至少有一个点的入度为 0。

1. 有向图中每个点的入度至少为 1，则有图的边数不小于图的点数，则无向图中必定有环（存在 BE）。
2. 利用 DFS 找到一条 BE（找到一个环），设边为  $uv$ ， $u$  为后继， $v$  为祖先。（如果没有找到 BE，说明存在一棵 DFS 生成树无环，则不能构造成功）
3. 以  $v$  为起点进行 DFS，将每条 TE  $xy$ ，标记方向为  $x \rightarrow y$ ，最后将  $uv$  边标记方向为  $u \rightarrow v$ ，即满足了该环中每个点的入度都大于 0。
4. 找到图中仍为白色的点，进行 2、3 步，若没有算法结束。
5. 由于只进行了两次深度优先遍历，较易得算法时间复杂度为线性。

具体算法实现见 [AddDirection](#) 算法

---

**算法 1** *AddDirection* 算法

---

**1.Function** *FindLoop* (*u*)\\找到一个环路，并返回 BE 边祖先结点

2.    *u.color* := *GRAY*, *res* := *None*
  3.    **foreach** *neighbor v of u* **do**
  4.       **if** *v.color* = *WHITE* **and** *v.vis* = *False* **then**
  5.            *FindLoop*(*v*)
  6.       **elif** *uv* = *BE* **then**
  7.            *res* := *u*, **break**
  8.    **return** *res*
- 

**1.Function** *AddDir* (*u*)\\给环路加方向

2.    *u.vis* = *True*\\标记其已经加边避免重复
  3.    **foreach** *neighbor v of u* **do**
  4.       **if** *v.color* = *WHITE* **and** *v.vis* = *False* **then**
  5.            change *uv dircetion* to  $u \rightarrow v$
  6.            *AddDir*(*v*)
  7.       **elif** *uv* = *BE* **then**
  8.            change *uv dircetion* to  $u \rightarrow v$
- 

**1.Function** *Main – AddDirection* (*V*, *E*)\\wrapper 部分

2.    **foreach** *point v in V* **do**
  3.        *v.color* := *WHITE*, *v.vis* := *False*
  4.    **foreach** *point v in V* **do**
  5.        **if** *v.color* = *WHITE* **and** *v.vis* = *False* **then**
  6.            *res* := *FindLoop*(*v*)
  7.            **if** *res* = *None* **then return** *False*
  8.            **else** *AddDir*(*res*)
  9.    **return** *True*
-



**problem 4.14**

**证明** 有向无环图中必有入度为 0 的点。

设图有  $N$  个结点，假设每个点的入度均不为 0，必有  $A_2 \rightarrow A_1, A_3 \rightarrow A_2, \dots, A_n \rightarrow A_{n-1}$ ，而对于  $A_n$  入度不为 0，故存在某个结点指向它，则出现环路，产生矛盾，得证。

**算法设计**

0. 因为  $G$  是有向无环图，则  $G$  一定存在拓扑排序（引理 4.2）。

1. 入度最小的点（即拓扑排序的队首） $a$ ，如果图中存在哈密顿路径，则  $a$  的入度一定为 0。假设  $a$  的入度不为 0，结点  $x$  指向  $a$ 。由于  $a$  是拓扑排序队首，故一定存在  $a$  到达  $x$  的路径（因为有图连通）设为  $a \rightarrow \dots \rightarrow x$ ，同时  $x$  指向  $a$ ，构成环路，与题干相矛盾。故  $a$  的入度一定为 0。

2. 显然，从  $a$  点出发，如果存在哈密顿路径，则一次 DFS 遍历所有结点。

3. 首先找到图  $G$  的入度为 0 的点  $a$ ，如果不存在或存在多个则不可能存在哈密顿通路。

4. 从  $a$  开始 DFS，每当递归回溯时（边为  $uv$ ），检查哈密顿路径  $path$  长度是否为结点个数。

5. 如果等于结点个数，说明存在路径  $a \rightarrow \dots u \rightarrow v \rightarrow \dots n$ ，算法结束。

6. 否则，将  $v.vis$  置为假，将  $v$  从  $path$  中剔除（哈密顿通路上  $u$  不直接到达  $v$ ），从  $u$  的下一子结点继续 DFS。

7. 找最小入度结点为  $O(m+n)$ ，DFS 过程显然是  $O(m+n)$ ，因此时间复杂度是线性的。

具体算法实现见 [FindHamilton 算法](#)

---

**算法 2** *FindHamilton* 算法

---

1. **Function** *FindHamilton* ( $u, path, n$ )\\找到一个环路，并返回 BE 边祖先结点

2.      $u.vis := True, path.push(u)$

3.     **foreach** neighbor  $v$  of  $u$  **do**

4.         **if**  $v.vis = False$  **then**

5.              $FindHamilton(v, path, n)$

6.             **if**  $len(path) == n$  **then return** *True*

7.              $v.vis := False, path.pop()$ \\没有找到，故将 vis 置为假，将 v 从 path 剔除

8.     **return** *False*\\u 所有子结点都没有找到哈密顿通路

---

1. **Function** *Main – FindHamilton* ( $V, E$ )\\wrapper 部分

2.     **foreach** point  $u$  in  $V$  **do**

3.          $u.vis := False, u.cnt := 0$ \\入度初始化为 0

4.         **foreach** neighbor  $v$  of  $u$  **do**  $u.cnt++$ \\计算各点的入度

5.      $tar := None, cnt := 0$ \\初始化要寻找的点

6.     **foreach** point  $u$  in  $V$  **do**\\找到唯一一个入度为 0 的点

7.         **if**  $tar \neq None$  **and**  $cnt = 0$  **and**  $u.cnt = 0$

8.             **then return** *False*\\存在不止一个入度为 0 的点，不可能存在哈密顿通路

9.         **if**  $cnt \geq u.cnt$  **then**  $tar := u, cnt := u.cnt$

10.     **if**  $tar = None$  **or**  $cnt \neq 0$  **then return** *False*\\不存在入度为 0 的点

11.     **return** *FindHamilton(tar)*

---

## problem 4.16

由problem 4.14证明可知，有向无环图必有入度为 0 的点。

## 算法设计

1. 有向无环图可以出现一个或者多个入度为 0 的结点，对这些结点的拓扑先后顺序没有要求。

2. 入度为 0 的结点的出边删去，必会出现入度为 0 的点（仍为无环图）。

3. 重复 1, 2 步，直到算法结束。可知时间复杂度为线性。

由开始的证明可知，图中存在回路，会出现没有入度为 0 的点的情况，无法确定拓扑顺序。

## 算法实现:

## 算法 3 ZeroTopoLogical 算法

---

1. **Function** *InitQueue* ( $G(V, E), queue$ )\\找到一个环路，并返回 BE 边祖先结点

2.     **foreach** *point*  $u$  *in*  $V$  **do**

3.         **if**  $InEdge[u] = 0$  **then**

4.              $queue.push(u)$

5.     **return**  $len(queue) > 0$ \\没有找到入度为 0 的点，返回 False

---

1. **Function** *Main – ZeroTopoLogical* ( $G(V, E)$ )\\找到一个环路，并返回 BE 边祖先结点

2.      $queue.init(), topo := list(), count := 0$

3.     /\*queue 存放入度为 0 的结点的队列, topo 存放拓扑排序结果, count 为排好序的结点数\*/

4.     **if** *InitQueue*( $G, queue$ ) = False **then return** False

5.     **while**  $queue.empty() \neq False$  **do**

6.          $u = queue.pop(), topo[count++] = u$ \\确定  $u$  为当前的逻辑起点

7.         **foreach** *neighbor*  $v$  *of*  $u$  **do**

8.              $InEdge[v] --$

9.             **if**  $InEdge[v] = 0$  **then**  $queue.push(v)$

11.     **return**  $count = len(V)$ \\所有结点应确定拓扑排序，否则存在环路

---

## problem 4.17

(1)

易知只需要以顶点  $s$  为起点进行一次 DFS，判断是否遍历所有点即可。代价为  $O(m+n)$ 。

---

```

1. Function OneToAll ( $G(V, E), s$ )
2.   DFS( $s$ )
3.   foreach point  $u$  in  $V$  do
4.     if  $u.color = WHITE$  then return False
5.   return True

```

---

(2)

利用课本上的划分强连通片的算法，将有向图  $G$  改造为  $G$  的收缩图  $G\downarrow$ ，各点之间的方向转换为各连通片之间的方向，易知  $G\downarrow$  为有向无环图。

题目可以转换为一个连通片可以到达其他所有连通片。由于  $G\downarrow$  为有向无环图，必有入度为 0 的连通片。故如果图  $G$  存在到达所有顶点的点，其必在该连通片中。再利用 (1) 算法判断该连通片能否到达所有连通片即可。

强连通片划分与 DFS 判断是否可达算法时间复杂度均为  $O(m+n)$ ，因此时间复杂度为  $O(m+n)$ ，符合题意。

具体算法实现见 [SccOneToAll 算法](#)

---

算法 4 SccOneToAll 算法

---

```

1. Function SccOneToAll ( $G(V, E)$ )
2.   construct  $G\downarrow$  using Scc( $G$ )
3.    $point := None, CountZero := 0$  \\ 记录入度为 0 的连通片个数
4.   foreach point  $u$  in  $V\downarrow$  do
5.     if  $InEdge[u] = 0$  then
6.        $point := u, CountZero := CountZero + 1$ 
7.   if  $point = None$  or  $CountZero > 1$  then
8.     return False \\ 没有或多个入度为 0 的点，返回 False
9.   return OneToAll( $G\downarrow, point$ )

```

---

## problem 4.18

(1)

同样利用课本上的划分强连通片的算法，将有向图  $G$  改造为  $G$  的收缩图  $G_{\downarrow}$ 。易知同一个连通片中的影响力值相同。易得影响力最低的连通片，一定是出度为 0 的连通片（一定存在，可能只有一个连通片），影响力是连通片中结点数减 1。因此，找到所有出度为 0 的连通片，并找出结点数最小的即可。

SCC 的划分和寻找出度为 0 的连通片并统计的时间复杂度为  $O(m+n)$

算法实现:

算法 5 MinImpact 算法

---

```

1. Function MinImpact ( $G(V, E)$ )
2.   construct  $G_{\downarrow}$  using  $Scc(G)$ 
3.    $MinPoint := None, MinCount := \infty$  \\ 记录连通片的最小个数
4.   foreach point  $u$  in  $V_{\downarrow}$  do
5.     if  $OutEdge[u] = 0$  and  $MinImpact < number(u)$  then
6.        $MinPoint := u, MinCount := number(u)$ 
7.   return ( $MinCount - 1, MinPoint$ ) \\ 结点数减 1 为影响力, 以及对应的连通片

```

---

(2)

同样得到有向图  $G$  的收缩图  $G_{\downarrow}$ ，易知影响力最大的连通片，一定是入度为 0 的连通片（压缩图无环）。

则对所有的入度为 0 的连通片进行 DFS 遍历，遍历到的连通片即为其可达的，DFS 结束后计算遍历到连通片的结点数。取所有 DFS 得到结点数最多的即为所求。

由于每次 DFS 后，都要查找所有访问过的连通片，因此时间复杂度为  $O(n(m+n))$ 。

具体算法实现见 [MaxImpact 算法](#)

---

**算法 6** MaxImpact 算法
 

---

```

1. Function MaxImpact ( $G(V, E)$ )
2.   construct  $G \downarrow$  using  $Scc(G)$ 
3.    $MaxPoint := None, CurCount := 0, MaxCount := 0$ 
4.   /*记录最大的 DFS 生成树的根以及结点个数*/
5.   foreach point  $u$  in  $V \downarrow$  do
6.     if  $InEdge[u] = 0$  then
7.        $DFS(G \downarrow, u), CurCount := 0$  \\更新当前 DFS 树的结点个数
8.       foreach point  $x$  in  $V \downarrow$  do
9.         if  $x.vis = True$  then
10.           $CurCount := CurCount + number(x)$ 
11.           $x.vis := False$  \\将访问置为 False, 避免下次 DFS 错误计算
12.        if  $MaxCount < CurCount$  then
13.           $MaxPoint := u, MaxCount := CurCount$ 
14.   return ( $MaxCount - 1, MaxPoint$ ) \\返回最大影响力, 以及对应的连通片

```

---

## problem 4.20

与problem 4.16 类似，每学期可以修所有入度为 0 的结点，将其加入暂存队列 *temqueue* 中。0. 在安排课程数小于总课程数前，重复下列步骤。

1. 将 *temqueue* 的副本复制给 *queue*，并将 *temqueue* 清空（避免每次都查找所有点来初始化队列）。
2. 将 *queue* 中的结点弹出，并将其指出的边删去，安排课程数加 1。
3. 假设本题有解即为有向无环图，一定再次出现入度为 0 的结点，将其加入暂存队列 *temqueue* 中。
4. 当 *queue* 中结点全部弹出，学期数加 1，即 *queue* 中的课程均被安排。可得算法时间复杂度是线性  $O(m + n)$ 。

算法实现:

算法 7 *ClassTopoLogical* 算法

- 
1. **Function** *Main – ClassTopoLogical* ( $G(V, E)$ )\\找到一个环路，并返回 BE 边祖先结点
  2.     *queue.init()*, *temqueue.init()*, *count* := 0, *TermCount* := 0
  3.     /\**temqueue* 存放入度为 0 的结点的队列, *count* 为排好序的结点数, *TermCount* 为学期数\*/
  4.         **if** *InitQueue*( $G$ , *temqueue*) = *False*\\初始化 *temqueue* 队列
  5.             **then return** *False*\\与problem 4.16 *InitQueue* 函数相同。
  6.     **while** *count* < *len*( $V$ ) **do**\\安排的课程数还少于总课程
  7.         *queue* := *temqueue.copy()*, *temqueue.clear()*\\得到上一次记录的入度为 0 的结点
  8.         **while** *queue.empty()* != *False* **do**
  9.             *u* = *queue.pop()*
  10.             *count* := *count* + 1\\已经确定的课程安排数加 1
  11.             **foreach** *neighbor v of u* **do**
  12.                 *InEdge*[*v*] – –
  13.                 **if** *InEdge*[*v*] = 0 **then** *temqueue.push*(*v*)
  14.         *TermCount* := *TermCount* + 1\\安排的学期数加 1
  15.     **return** *TermCount*\\最少的学期数
-

## problem 4.22

(1)

与problem 4.16 相同,  $i$  恨  $j$  记为  $i$  指向  $j$ 。找到合理的拓扑排序即可。如果存在环路则不存在拓扑排序。仍为重复找入度为 0 的顶点  $u$ , 将  $u$  放入拓扑序列中 (因为没有人恨他), 并删去  $u$  指向的边。如果在所有点安排拓扑排序前, 没有找到入度为 0 的顶点, 说明存在环路, 不存在符合条件的排队。

具体算法实现与ZeroTopoLogical 算法相同。

(2)

与problem 4.20 相同。首先将所有入度为 0 的点的边删除后, 这些结点的行数记为  $row$ 。再将删除过程中发现的入度为 0 的点加入队列,  $row$  加 1, 表明开始下一一次的排列。重复上述过程, 直到所有小孩被安排顺序。

如果没有发现入度为 0 的点且没有小孩全部安排, 说明存在环路, 不存在符合条件的安排。

具体算法实现与ClassTopoLogClassical 算法相同。

## problem 4.23

(1) 存在,  $x_1 = TRUE, x_2 = TRUE, x_3 = FALSE, x_4 = TRUE$  符合。

(2)  $(x_3 \vee x_4) \wedge (x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2) \wedge (x_1 \vee \overline{x_2})$ 。

(3)

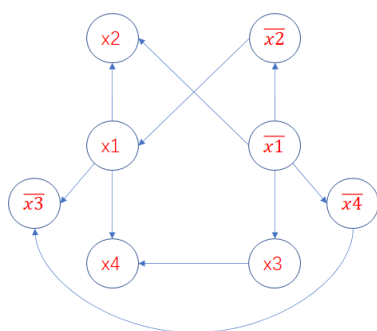


图 1: 题目所给的实例

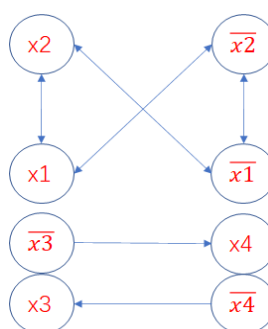


图 2: (2) 中所给实例



(4)

**证明** 如果连通片中存在同时包含  $x$  和  $\bar{x}$  的强连通片,  $(x \rightarrow \bar{x}), (x \leftarrow \bar{x})$  两条边存在, 且由构造实例图过程表明原 2-SAT 有子句  $(x \wedge \bar{x}) \equiv FALSE$ , 该子句恒为假, 故实例 I 不存在使所有子句都满足的赋值。

(5)

**证明** 子句  $(\alpha \vee \beta) \equiv (\bar{\alpha} \rightarrow \beta) \wedge (\bar{\beta} \rightarrow \alpha)$ , 因此若要使得  $(\alpha \vee \beta)$  成立, 上述蕴涵关系必须成立。在 2-SAT 实例图中的一条边  $a \rightarrow b$  则说明了必含有子句  $(\bar{a} \vee b)$ 。故要想使得原式成立  $a \rightarrow b$  必须为 TRUE。

因此可以抽象地看作, 如果选取点  $a$  ( $a$  为 TRUE), 且存在  $a \rightarrow b$ , 则  $b$  点必须选择。或者两者均不选择。如果一系列结点在同一个强连通片中, 意味着彼此可达, 则要么都选择, 要么都不能选择。如果连通片中不存在一个文字及该文字的取反, 说明这种选择是一定存在的。

**证明选择的可行性**

如果存在  $a \rightarrow b$ , 则  $\bar{b} \rightarrow \bar{a}$  一定存在, 即原图是存在对称关系的。因此可以将原图化为强连通片压缩图  $G_{\downarrow}$  (必为 DAG), 原图中的对称关系也存在于  $G_{\downarrow}$  中。即  $a$  所在连通片  $G_1$  的其他文字的取反在  $\bar{a}$  的连通片  $\overline{G_1}$  中。

对立连通片之间的对称传递: 如果  $G_1 \rightarrow G_2$ , 那么  $\overline{G_2} \rightarrow \overline{G_1}$ 。

同时对于 DAG 中存在边相连 (祖先后继关系) 的情况, 为满足原式逻辑成立, 故其后继也应为真, 故需要进行拓扑排序。

因此, 当  $x$  所在的强连通分量  $G_1$  的拓扑序在  $\bar{x}$  所在的强连通分量的拓扑序之后则  $G_1$  为真,  $\overline{G_1}$  为假, 即按照反向拓扑的顺序进行 (出度最小, 对其它连通片影响最小)。重复选择直至所有连通片都被标记即可得到一组解。

(6)

利用课本上的划分强连通片的算法, 将有向图  $G$  改造为  $G$  的收缩图  $G_{\downarrow}$ , 之后进行逆向拓扑排序, 即在构造图时将边的方向置反即可, 出度为 0 转化为入度为 0。反复选取拓扑序在前的连通片并将其对立的连通片置为假即可。时间复杂度为线性  $O(m + n)$ 。

**具体算法实现2-SAT 算法**

---

 算法 8 2-SAT 算法
 

---

 1. **Function** 2 – SAT ( $G(V, E)$ )

 2.      $queue.init(), temqueue.init(), count := 0$ 

 3.     construct  $G \downarrow$  using  $Scc(G)$ 

 4.      $queue.init(), temqueue.init(), count := 0$ 

5.     /\*temqueue 存放入度为 0 的结点的队列, count 为被选择的连通片个数\*/

 6.     **foreach** part  $X$  of  $V \downarrow$  **do**

 7.         **foreach** point  $y$  of  $X \downarrow$  **do**

 8.             **if**  $\bar{y} \in X$  **then return** FALSE

 9.         **if** InitQueue( $G, temqueue$ ) = False \\ 初始化 temqueue 队列

 10.             **then return** False \\ 与 problem 4.16 InitQueue 函数相同。

 11.     **while**  $count < len(V \downarrow)$  **do** \\ 被选择的连通片个数

 12.          $queue := temqueue.copy(), temqueue.clear()$  \\ 得到上一次记录的入度为 0 的结点

 13.         **while**  $queue.empty() \neq \text{False}$  **do**

 14.              $u = queue.pop()$ 

 15.             **foreach** point  $x$  of  $u$  **do**

 16.                  $x.value = \text{TRUE}$  \\ 将  $u$  中所有点标记为 TRUE

 17.              $count := count + 2$  \\ 已经确定连通片个数，对立连通片也被确认

 18.             **foreach** neighbor  $v$  of  $u$  **do**

 19.                  $InEdge[v] --$ 

 20.                 **if**  $InEdge[v] = 0$  **then**  $temqueue.push(v)$ 

 21.     **return** True \\ 能够正常选择
 

---

## Chapter 5

### problem 5.1

#### 不存在 BE

假设存在边  $uv$  是 BE，即当遍历结点  $u$  时，发现其黑色邻居  $v$ ，且  $v$  是  $u$  在 BFS 树中的祖先结点。由于为无向图， $v$  也直接指向  $u$ ，故当队列中  $v$  弹出时，发现  $u$  为白色，则将  $u$  加入队列，即  $vu$  是 TE。则在  $u$  出队时发现的  $uv$  边应是 TE，不应二次遍历，故无向图 BFS 不存在 BE。

#### 不存在 DE

假设存在边  $uv$  是 DE，在结点  $u$  刚出队列，即将处理它的所有邻居的时刻，结点  $v$  的情况：

1. 结点  $v$  不可能是白色，否则  $uv$  只能是 TE。
2. 结点  $v$  不可能是灰色，因为  $u$  刚出队列，如果  $v$  为灰色，说明  $v$  已经在队列中，这和  $u$  是  $v$  在遍历树中的祖先结点矛盾。
3. 结点  $v$  不可能是黑色，在  $u$  刚被发现的时刻，如果  $v$  已经结束遍历，这同样与  $u$  是  $v$  在遍历树中的祖先结点矛盾。

### problem 5.2

可以使用深度优先遍历判断。无向图 DFS 在遍历过程中如果  $uv$  为 TE 则直接给  $v$  染为与  $u$  相反颜色即可，如果  $uv$  为 BE 说明存在环路，则判断  $uv$  两点颜色是否相同，若不同继续递归遍历，若相同则不是二分图。

#### 算法 9 BinaryDFS 算法

1. **Function** *BinaryDFS* ( $u$ )

3.   **foreach** *neighbor*  $v$  of  $u$  **do**

4.       **if**  $v.color = WHITE$  **and**  $v.vis = False$  **then**

5.            $v.parcolor : r = !u.parcolor$  \\\给  $v$  添加与  $u$  相反的颜色

6.           **if** *BinaryDFS*( $v$ ) = *FALSE* **then return** *FALSE*

7.       **elif**  $uv = BE$  **then**

8.           **if**  $u.parcolor = v.parcolor$  **then return** *FALSE* \\\如果  $uv$  颜色相同说明不是二分图

9.   **return** *True* \\\可以正确染色

### 深度优先遍历

**优点：**能够找到所有的可行解，相比于 BFS 内存开销较小。

**缺点：**因为会发生回溯在深度较大的情况下效率不高。递归栈的空间与深度成正比存在爆栈的风险。

### 广度优先遍历

**优点：**保证了每个结点只访问一遍，且以最短路径访问，效率高。

**缺点：**但是需要较大的内存来维护队列。

### problem 5.4

#### (1)

对于一个结点，其高度为左右子树中较高子树的高度加 1。因此递归计算左右子树高度，当结点为空时高度记为 0 即可。每个结点访问一次，时间复杂度为  $O(n)$ 。

#### 算法实现：

---

```
1. Function MaxLen (root)  
2.   if root = NULL then return 0  
3.   left := MaxLen(root → left)  
4.   right := MaxLen(root → right)  
5.   return max(left, right) + 1
```

---

#### (2)

树的最短路径的最大的边一定通过根结点，从根出发到达左子树最长边和到达右子树最长边之和即为左右子树高度之和。时间复杂度同样为  $O(n)$ 。

#### 算法实现：

---

```
1. Function MaxWidth (root)  
2.   if root = NULL then return 0  
3.   left := MaxLen(root → left)  
4.   right := MaxLen(root → right)  
5.   return (left + right)
```

---

## problem 5.8

(1)

## 1. 有向图

**1.1 DFS 证明:** 有向图中, 有环路当且仅当存在 BE

在有向图中如果存在边  $uv$  是 BE, 说明  $v$  是  $u$  的祖先节点, 且有一条路径  $v \rightarrow x \rightarrow \cdots \rightarrow u$ , 即  $u$  和  $v$  存在一条环路。

如果有环, 假设图中存在环路  $u \rightarrow x \rightarrow \cdots v \rightarrow u$ , 任取环路中一点  $y$  为起点进行 DFS, 且存在边  $wy$ , 则在遍历中, 当遍历至  $w$  时, 发现有一条指向  $y$  的边, 同时  $y$  是其祖先结点, 故  $wy$  为 BE。

**算法实现:****算法 10** DirDFSLoop 算法

---

1. **Function** *DFSLoop* ( $u$ )\\找到 BE, 返回 True 存在环路
2.  $u.color := GRAY$ 3. **foreach** neighbor  $v$  of  $u$  **do**4. **if**  $v.color = WHITE$  **then**5. **if** *DFSLoop*( $v$ ) **then return** *True*6. **elif**  $uv = BE$  **then**7. **return** *True*8. **return** *False*


---

1. **Function** *DirDFSLoop* ( $V, E$ )\\wrapper 部分
2. **foreach** point  $v$  in  $V$  **do**3.  $v.color := WHITE$ \\初始化所有点为未访问4. **foreach** point  $v$  in  $V$  **do**5. **if**  $v.color = WHITE$  **then**6. **if** *DFSLoop*( $v$ ) **then return** *True*\\存在环路, 无需继续遍历7. **return** *False*\\不存在返回 *False*

**1.2 BFS** 对于有向图，没有环，当且仅当存在拓扑排序。由problem 4.14证明可知，有向无环图必有入度为 0 的点。每次寻找入度为 0 的点将其加入队列，并按照 BFS 的方式进行遍历，即队列每弹出一个结点，便将其出边删除，如果为无环图，该过程必产生入度为 0 的点，同样将其加入队列。最后判断放入队列的结点数的个数，如果等于总结点数，说明无环，否则有环。

算法实现:

算法 11 *DirBFSLoop* 算法

---

```

1. Function BFSLoop (u)\\计算该次 BFS 可被安排的结点数
2.   queue.push(u), count := 0
3.   /*queue 存放入度为 0 的结点的队列, count 为排好序的结点数*/
4.   while queue.empty() != False do
5.       u = queue.pop(), count := count + 1\\确定 u 为当前的逻辑起点
6.       foreach neighbor v of u do
7.           InEdge[v] --
8.           if InEdge[v] == 0 then queue.push(v)
9.   return count\\返回排好序的结点数

```

---

```

1. Function DirBFSLoop (G(V, E))\\wrapper 部分
2.   count := 0\\count 为排好序的结点数
3.   foreach point u in V do
4.       if InEdge[u] = 0 then
5.           count := count + BFSLoop(u)
6.   return count = len(V) > 0\\所有结点应确定拓扑排序，否则存在环路

```

---

## 2. 无向图

**2.1 DFS** 对于无向图，DFS 中可能出现 BE 和 TE，由1.1DFS证明可知，如果有 BE 则有环路。

算法实现:

---

算法 12 NoDirDFSLoop 算法

---

1. **Function** *NoDFSLoop* (*u*)\\找到 BE, 返回 True 存在环路

2.     *u.color* := *GRAY*

3.     **foreach** *neighbor v of u* **do**

4.         **if** *v.color* = *WHITE* **then**

5.             **if** *NoDFSLoop*(*v*) **then return** *True*

6.         **else return** *True*\\不是 TE, 则有环路

7.     **return** *False*

---

1. **Function** *NoDirDFSLoop* (*V, E*)\\wrapper 部分

2.     **foreach** *point v in V* **do**

3.         *v.color* := *WHITE*\\初始化所有点为未访问

4.     **foreach** *point v in V* **do**

5.         **if** *v.color* = *WHITE* **then**

6.             **if** *NoDFSLoop*(*v*) **then return** *True*\\存在环路, 无需继续遍历

7.     **return** *False*\\不存在返回 False

---

**2.2 BFS** 对于无向图，BFS 中可能存在 CE 和 TE，如果存在  $uv$  为 CE，即当遍历  $u$  时刻发现  $v$  是灰色，即两者皆在队列中，那么  $u$  和  $v$  必有一点  $x$ （已经出队）可达，则三点构成环路。

算法实现:

算法 13 *NoDirBFSLoop* 算法

---

1. **Function** *NoBFSLoop* ( $u$ )\\找到一个环路，并返回 BE 边祖先结点

```

2.   queue.push(u), u.color := GRAY
3.   /*queue 存放入度为 0 的结点的队列*/
4.   while queue.empty() != False do
5.        $u = \text{queue.pop()}$ \\确定  $u$  为当前的逻辑起点
6.       foreach neighbor v of u do
7.           if  $v.\text{color} = \text{WHITE}$  then queue.push(v)
8.           else return True\\存在其他边，则为环路
9.   return False\\当前 BFS 生成树不存在环路

```

---

1. **Function** *NoDirBFSLoop* ( $G(V, E)$ )\\wrapper 部分

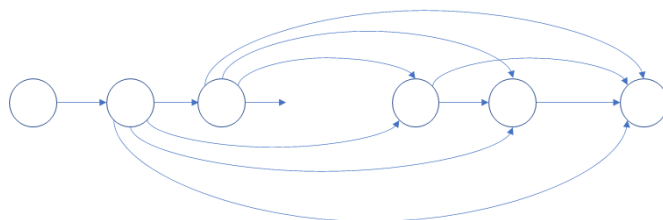
```

2.   foreach point v in V do
3.        $v.\text{color} := \text{WHITE}$ \\初始化所有点为未访问
4.   foreach point u in V do
5.       if  $u.\text{color} = \text{WHITE}$  then
6.           if NoBFSLoop(u) then return True\\存在环路，无需继续遍历
7.   return False\\不存在返回 False

```

---

(2)



构造如上图所示的有向图，除了出发结点外，每个结点与它的后继之间均存在 DE，即总的边数为  $\Omega(n^2)$ 。因为  $O(n)$  时间复杂度无法遍历所有边，故不能判断是否成环。



**problem 5.9****(1)**

因为图边权均为 1，且图是连通的，则任意顶点出发进行一次 DFS，除了 TE 别的边都舍弃，得到的 DFS 生出树即为最小生成树。

**(2)**

首先进行一次 DFS 生成，在生成树中选取了  $n-1$  条边，还有额外的 11 条边。DFS 生成树建立线性时间复杂度  $O(m+n)$ 。为了满足边权最小，将这 11 条边逐一加入生成树。每加一条边则必然产生一个环，在这个环中找出边权最大的将其删除即可。每插入一次，找出环中最大边权并删除是线性时间  $O(m+n)$ 。综上时间复杂度为  $O(m+n)$ 。

**(3)**

先只考虑边权为 1 的边，进行相同规则的 DFS，记录所加的边权为 1 的边数。如果所有边权为 1 所在的点均已被遍历，所加边数仍小于  $n-1$ 。若产生了  $k$  个连通片，从那些边权为 2 的边进行选择，每选一条使得两个连通片连通，选取  $k-1$  条后，算法终止。

## problem 5.10

(1)

经由 BFS 遍历的生成树，从  $s$  到达每个顶点的 TE 即为最短路径。同时 BE 对图中的  $s$  和  $u$  来讲对最短路径没有贡献。而对于 CE 边来讲，如果存在边  $uv$  是 CE，在有向图中  $v.dis \leq u.dis + 1$ ，则  $s$  从  $u$  到达  $v$  的距离是  $u.dis + 1$ ，而从  $s$  不经过  $u$  直接经过若干 TE 到达距离为  $v.vis$ ，因此当  $v.dis = u.dis + 1$  时， $s$  经由  $u$  到达  $v$  也是最短路径。从  $s$  为起点进行 BFS，每当遇到 TE 或者 CE 的边 (两点的  $dis$  相差不为 1) 则将该边加入新图中。

---

1. **Function**  $BFS(u)$

```

2.    $temG.init(), queue.push(u), u.color := GRAY, u.dis := 0$ 
3.   /* $temG$  为新产生的图， $queue$  存放入度为 0 的结点的队列*/
4.   while  $queue.empty() \neq True$  do
5.        $u = queue.pop()$  \\ 确定  $u$  为当前的逻辑起点
6.       foreach  $neighbor\ v\ of\ u$  do
7.           if  $v.color = WHITE$  then
8.                $v.vis := u.dis + 1, v.color := GRAY, queue.push(v)$ 
9.                $temG.Add(u, v)$  \\ 将 TE 边  $uv$  加入图中
10.          elif  $v.color = GRAY$  and  $v.dis = u.dis + 1$  then
11.               $temG.Add(u, v)$  \\ 将特定的 CE 边  $uv$  加入图中
12.               $u.color := BLACK$ 
13.   return  $temG$ 

```

---

1. **Function**  $BFS - Wrapper(G(V, E), s)$  \\ wrapper 部分

```

2.   foreach  $point\ v\ in\ V$  do
3.        $v.color := WHITE, v.vis := \infty$  \\ 初始化所有点为未访问
4.    $G' := BFS(u)$ 

```

---

(2)

由第一问的分析可知， $s$  到达  $u$  的最短路径的个数，即为在  $G'$  中  $u$  的父结点的个数，则从  $s$  开始 BFS 遍历逐层累加即可。

具体算法实现见 [GetPathNum 算法](#)。

---

算法 13 GetPathNum 算法

---

```
1. Function GetPathNum ( $G'(V', E'), s$ )  
2.   foreach point  $v$  in  $V'$  do  
3.      $v.color := WHITE, c[v] := 0$  \ 初始化所有点为未访问  
4.    $queue.push(s), c[s] := 1$  \ 从  $s$  开始 BFS,  $s$  到达  $s$  本身有一条最短路径  
5.   while  $queue.empty() \neq True$  do  
6.      $u = queue.pop()$  \ 确定  $u$  为当前的逻辑起点  
7.     foreach neighbor  $v$  of  $u$  do  
8.        $c[v] := c[v] + c[u]$  \  $v$  的父结点数加  $u$  的父结点数目相同  
9.       if  $v.color = WHITE$  then  
10.         $v.color := GRAY, queue.push(v)$   
11.       $u.color := BLACK$   
12.   return  $c$ 
```

---