§3.4 有向图的强连通性

一、定义

设$G=(V,E)$是一个有向图。我们可以把 V 划分成等价类$V_i(1 \le i \le r)$如下：顶点 v 和 w 是等价的当且仅当存在一条从 v 到 w 的有向路径(简称路径) 和一条从 w 到 v 的路径。设$E_i(1 \le i \le r)$是连接V_i中顶点对的边的集合。图$G_i=(V_i,E_i)$称为 G 的强连通分支。一个图称为是强连通的，是指它只包含一个强连通分支。

强连通分支的深度优先搜索生成树的根称为该强连通分支的根。

二、算法的理论依据

引理 3.4：设$G_i=(V_i,E_i)$是有向图 G 的一个强连通分支，设$S=(V,T)$是 G 的深度优先搜索生成森林。那么，G_i的所有顶点和$E_i \cap T$中的边构成一棵树。

证明：设 v 和 w 是V_i中的顶点(假设每个顶点以深度优先搜索序号命名),见图 3.7。不失一般性，假设$v<w$。由于 v 和 w 在同一强连通分支中，G_i中存在从 v 到 w 的路径 P。设 x 是 P 上标号最小的顶点(可能$x=v$)，当 P 到达 x 的某个后代以后，它不可能离开 x 的后代构成的子树，这因为能离开那棵子树的边是到达一个标号比 x 小的顶点的横跨边或回边(因为 x 的后代是从 x 开始连续地标号的，离开该子树的横跨边或回边必然到达一个标号比 x 小的顶点)。因此 w 是 x 的一个后代。由于顶点按先序遍历的顺序标号，标号在 x 和 w 之间的所有顶点都是 x 的后代。因为$x \le v<w$，故 v 是 x 的后代。

我们已证G_i中任何两个顶点有一个共同的祖先在G_i中。设 r 是G_i中的顶点的公共祖先中标号最小的那个顶点，如果 v 在G_i中，那么在深度优先生成树中，从 r 到 v 的路径上的任何顶点也在G_i中。证毕。

*有向图 G 的强连通分支可以通过深度优先搜索找到各分支的根，然后根据查找各个根的逆顺序找到各个分支。设$r_1,r_2,\cdots,r_k$是各个分支的根，按照深度优先搜索查找这些结点终止的顺序列出(即，对r_i的查找比对r_{i+1}的查找先终止)。那么对每个$i<j$，或者r_i在r_j的左边，或者r_i是r_j的后代。

设G_i是以r_i为根的强连通分支$(1 \le i \le k)$。那么G_1由r_1的所有后代组成，因为不存在$r_j(j>1)$使得r_j是r_1的后代。

引理 3.5：对每个$i(1 \le i \le k)$，G_i由以下顶点组成，它们是r_i的后代，但它们不是$G_1,G_2,\cdots,G_{i-1}$的顶点。

证明：根$r_j(j>i)$不可能是r_i的后代，这因为调用 SEARCH(r_j)在 SEARCH(r_i)之后终止。证毕。

定义 LOWLINK 如下：

LOWLINK[v]=MIN({v}∪{w|存在横跨边或回边从 v 的后代到 w，并且包含 w 的强连通分支的根是 v 的一个祖先})　　(3.3)

引理 3.6：设 G 是一个有向图。顶点 v 是 G 的一个强连通分支的根当且仅当 LOWLINK[v] = v。

证明：仅当) 设 v 是 G 的一个强连通分支的根。由 LOWLINK 的定义，LOWLINK[v] ≤ v。假设

LOWLINK[v] $< v$。那么存在顶点 w 和 r 使得：
1. w 由一条从 v 的后代发出的横跨边或回边到达，
2. r 是包含 w 的强连通分支的根，
3. r 是 v 的祖先，并且$w < v$。

由条件 2，r 是 w 的祖先，因此，$r \leq w$。因而由条件 4，$r < v$，和条件 3 蕴含 r 是 v 的真祖先，但 r 和 v 必然在同一强连通分支中，这因为 G 中存在路径从 r 到 v 和路径从 v 到 w 再到 r。因此，v 不是强连通分支的根，与假设矛盾。从而 LOWLINK[v] = v。

当）设 LOWLINK[v] = v。如果 v 不是包含 v 的那个强连通分支的根，那么 v 的某个真祖先 r 是根。因此存在路径 P 从 v 到 r, 考虑 P 中第一条从 v 的后代到一个不是 v 的后代的顶点 w 的边。这条边或者是到 v 的一个真祖先的回边，或者是到一个标号比 v 小的顶点的横跨边。在两种情况下，都有$w < v$。

剩下的工作要证 r 和 w 在 G 的同一强连通分支中。因为 r 是 v 的祖先，因而一定存在从 r 到 v 的路径。而路径 P 从 v 到 w 到 r。从而 r 和 w 在同一强连通分支中。故 LOWLINK[v] $\leq w < v$，矛盾。 证毕。

三、算法

输入：一个有向图$G = (V, E)$;
输出：G 的所有强连通分支的表；
方法;

```
   BEGIN
      COUNT := 1;
      FOR all v in V DO mark v “new”;
      Initialize STACK to empty;
WHILE there exists a vertex v marked “new” DO
   SEARCHC(v);
   END;
   PROCEDURE SEARCHC (v);
   BEGIN
  mark v “old”;
  DFNUMBER[v] := COUNT;
  COUNT := COUNT+1;
  LOWLINK[v] := DFNUMBER[v];
  push v on STACK;
   FOR each vertex w on L[v] DO
      IF w is marked “new” THEN
        BEGIN
          SEARCHC (w);
          LOWLINK[v] := MIN(LOWLINK[v], LOWLINK[w]);
      END
      ELSE
    IF DFNUMBER[w]<DFNUMBER[v] and w is on STACK THEN
      LOWLINK[v] := MIN(DFNUMBER[w],LOWLINK[v]);
   IF LOWLINK[v] = DFNUMBER[v] THEN
```

```
    BEGIN
      REPEAT
    pop x from top of STACK;
    print x;
  UNTIL x = v;
  print "end of strongly connected component";
    END
  END;
```

§ 3.5 找图中最短圈

*用宽度优先搜索的方法

```
  PROCEDURE BFS1 (i : integer; adjlist : 图的邻接表; VAR Mincyclelength : integer);
  VAR
    queue, FATHER, BRANCH : ARRAY [1‥max] OF integer;
    visit : ARRAY [1‥max] OF boolean;
    j, hp, tp, x, x', y, y', xm, ym, branch, cyclelength : integer;
    p : 顶点指针; new : boolean;
  BEGIN
    FOR j := 1 TO max DO visit[j] := false;
    mincyclelength := ∞; new := false;
    hp := 1; tp := 1;
    visit[i] := true; queue[hp] := i;   /*访问顶点 i*/
p := adjlist[i]; branch:= 1;
    WHILE p≠NIL DO   /*访问根结点 i 的所有相邻点*/
BEGIN
      x := p^.vertex; FATHER[x] := i;
      BRANCH[x] := branch; branch := branch+1;
/*每个儿子一个分支*/
      visit[x] := true;
      tp := tp+1; queue[tp] := x; p := p^.link;
    END;
    hp := hp+1;
    REPEAT      /*访问 i 的所有儿子的后代*/
      p := adjlist[queue[hp]];
      WHILE p≠NIL DO
      BEGIN
        x := p^.vertex;
        IF NOT visit[x] THEN
        BEGIN
          FATHER[x] := queue[hp];
          BRANCH[x] := BRANCH[FATHER[x]];
          visit[x] := true; tp := tp+1; queue[tp] := x;
        END
```

```
            ELSE IF BRANCH[x]≠BRANCH[queue[hp]] THEN
BEGIN
                y := queue[hp];   x' := x;   y' := y;
                cyclelength := 1;
                WHILE x ≠ i  DO
                BEGIN
                    cyclelength := cyclelength+1;
                    x := FATHER[x];
                END;
                WHILE y  ≠ i  DO
                BEGIN
                    cyclelength := cyclelength+1;
                    y := FATHER[y];
                END;
            END; /* WHILE */
            IF cyclelength < mincyclelength THEN
            BEGIN
                y_m := y';   x_m := x';  new := true;
                mincyclelength := cyclelength;
            END;
            p := p^.link;
        END;
        IF new THEN
        BEGIN   /*找到包含 i 的最短圈，打印该圈*/
            WRITE("(x_m, y_m)"); new := false;
            WHILE (x_m ≠ i) OR (y_m ≠ i) DO
            BEGIN
                IF  x_m ≠ i  THEN BEGIN WRITE("(x_m, FATHER[x_m])");
                    x_m := FATHER[x_m]   END;
                IF  y_m ≠ i  THEN BEGIN WRITE("(y_m, FATHER[y_m])");
                    y_m := FATHER[y_m]  END;
            END;
        END;
        hp := hp+1;
UNTIL hp > tp;
    END;
主程序：
    BEGIN
        minlength :=  ∞;
        FOR i := 1 TO |V| DO
        BEGIN
            BFS1(i, adjlist, mincyclelength);
            IF mincyclelength < minlength THEN
                minlength := mincyclelength;
```

```
    END;
    WRITE("The length of minimum cycle is", minlength);
END;
```

§3.6 贪心算法与连线问题

问题：假设要建造一个连接若干城镇的铁路网络。已知城镇v_i和v_j之间直通线路的造价为c_{ij}，试设计一个总造价最小的铁路网络。这个问题名为连线问题。

把每个城镇看作是具有权$w(v_i v_j) = c_{ij}$的赋权图 G 的顶点，问题转化为：在赋权图 G 中，找出具有最小权的连通生成子图。由于权是造价，当然是非负的，所以最小权生成子图是 G 的一棵生成树 T。赋权图的最小权生成树称为最优树。

例 3.5：最小生成树的例子：(见图 3.8)

°Kruskal 算法：

选择连杆e_1，使得$w(e_1)$尽可能小。

若已选定边$e_1, e_2, \cdots, e_i$，则从$E\backslash\{e_1, e_2, \cdots, e_i\}$中选择$e_{i+1}$，使

(i)$G[\{e_1, e_2, \cdots, e_{i+1}\}]$为无圈图；

(ii)$w(e_{i+1})$是满足(i)的尽可能小的权。

当第 2 步不能继续执行时，则停止。

*以例 3.5 的图为例，讲解算法。

*Kruskal 算法本质上是贪心算法，贪心算法不一定总能求出最优解，必须证明算法的正确性。

°算法正确性证明：

定理 3.7：由 Kruskal 算法构作的任何生成树$T^* = G[\{e_1, e_2, \cdots, e_{\nu-1}\}]$都是最优树。

证：用反证法。对 G 的任何异于T^*的生成树 T，用 f(T)记使e_i不在 T 中的最小 i 值。现在假设T^*不是最优树，T 是一棵使 f(T)尽可能大的最优树。

假设$f(T) = k$；也就是说，$e_1, e_2, \cdots, e_{k-1}$同时在 T 和$T^*$中，但$e_k$不在 T 中。则$T + e_k$包含唯一的圈 C。设$e_k'$是 C 的一条边，它在 T 中而不在$T^*$中，$e_k'$不是$T + e_k$的割边，因此，$T' = (T + e_k) - e_k'$是具有

$\nu - 1$条边的连通图，所以它是 G 的另一棵生成树。显然

$$w(T') = w(T) + w(e_k) - w(e_k') \tag{3.4}$$

在 Kruskal 算法中选出的边e_k，是使$G[\{e_1, e_2, \cdots, e_k\}]$为无圈图的权最小的边。由于$G[\{e_1, e_2, \cdots, e_{k-1}, e_k'\}]$是 T 的子图，它也是无圈的。于是得到：

$$w(e_k') \geq w(e_k) \tag{3.5}$$

结合(3.4)式和(3.5)式，有

$$w(T') \leq w(T)$$

所以 T'也是一棵最优树，然而

$$f(T') > k = f(T),$$

与 T 的选法矛盾。因此，$T = T^*$，从而T^*确实是一棵最优树。证毕。