有向图的强连通分量及应用

吴金全

（武汉市武钢三中，武汉 430080）

摘 要：有向图的强连通分量应用非常广泛，比如有向图的强连通分量数量巨大的时候，为了更加高效必须要用缩点法。深度优先遍历是求有向图的强连通分量的一个有效方法，根据实现方式的不同，总体上，求有向图的强连通分量有三种算法，分别是 Kosaraju 算法，Gabow 算法和 Tarjan 算法。三种算法的时间复杂度均为 O(n+e)(n 为顶点数，e 为边数)。

关键字：DAG;Tarjan;Kosaraju;Gabow 时间复杂度

中图分类号：TP311.12 **文献标识码**：A DOI: 10.3969/j.issn.1003-6970.2014.03.022

本文著录格式：[1] 吴金全 . 有向图的强连通分量及应用 [J]. 软件 ,2014,35(3)：72-75

The SCC of Digraph and Application

WU Jin-quan

（*WuGangSanZhong of Wuhan City,Wuhan* 430080,*China* ）

【Abstract】The application of SCC is normal.For example,when the number of SCC is giant,in order to improve the efficiency ,it must use the way of reducing the vertexs.The DFS is an effective way.According to the way of accomplishing,in general,there are three algorithms what can find the SCC of digraph including Kosaraju,Tarjan,Gabow.The Time Complexity of each is O(n+e).

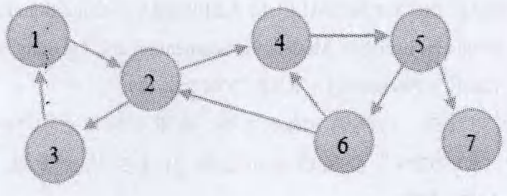
【**Key words**】DAG;Tarjan;Kosaraju;Gabow Time Complexity

0 引言

在对无向图进行遍历时，对于连通图，仅需一次调用搜索过程 (DFS 或 BFS)。换言之，即从图中任一顶点出发，便可历访图中各个顶点。对非连通图，则需多次调用搜索过程。而每次调用得到的顶点访问序列恰为其各个连通分量中的顶点集 [1]。

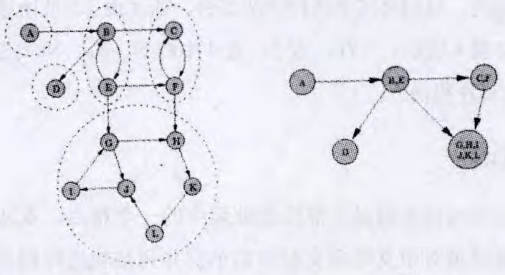
在有向图 G 中，如果两个顶点 Vi，Vj 间有一条从 Vi 到 Vj 的有向路径，同时还有一条从 Vj 到 Vi 的有向路径，则称两个顶点强连通 (strongly connected)[4]。如果有向图 G 的每两个顶点都强连通，称 G 是一个强连通图。有向图的极大强连通子图，称为强连通分量 SCC(Strongly Connected Components)[2]。如果图中一个顶点没有和其它顶点强连通，这个顶点自己也是一个强连通分量。

下面有向图中，子图 {3,6,5,4,2,1} 为一个强连通分量，因为这个连通分量内顶点两两可达。{7} 也是个强连通分量，只包含一个顶点。



直接根据定义，有向图的强连通分量就不会像无向图的连通分量那样简单，那么如何求呢？

对于下面 (a) 图我们借助 DFS。如果从 I 开始 DFS，将得到只包含 {G,H,I,J,K,L} 的一棵 DFS 树，然后从 C 出发，得到 {C,F}，再从 D 出发得到 {D}，依次类推，每次得到一个 SCC。如下面 (b) 图所示。



(a) (b)

遗憾的是，并不是每个顺序都是“好用”的。如果一开始就不幸选择从 A 开始进行遍历，这颗 DFS 树将包含整个图。也就是说，这次不明智的 DFS 把所有 SCC 混在了一起，什么也没得到。很明显，我们希望按 SCC 图拓扑顺序的逆序进行遍历。这样才能每次 DFS 只得到一个 SCC，而不会把两个或多个 SCC 混在一起。Kosaraju 算法便是基于这个思想。

深度优先遍历是求有向图的强连通分量的一个有效方法，根据实现方式的不同，总体上，求有向图的强连通分量有三种算法，分别是 Kosaraju 算法、Gabow 算法和 Tarjan 算法，时间复杂度均为 O(n+e)。其中 Kosaraju 算法要对原图和逆图都进行一次 DFS，另外两种算法只要 DFS 一次，Gabow 算法是 Tarjan 算法的改进。

作者简介：吴金全（1978-），男，中级，计算机数据结构和算法

下面分别详细介绍常用的两种算法 Tarjan 和 Kosaraju。

1 Kosaraju 算法

Kosaraju 算法基于以下思想：强连通分量一定是某种 DFS 形成的 DFS 树森林[5]。

这个算法可以说是最容易理解，最通用的算法，其比较关键的部分是同时应用了原图 G 和反图 GT[3]。步骤 1：先对原图 G 进行深搜形成森林（树），步骤 2：任选一棵树对其进行深搜（注意这次深搜结点 A 能往子结点 B 走的要求是边 A->B 存在于反图 GT），能遍历到的顶点就是一个强连通分量。余下部分和原来的森林一起组成一个新的森林，继续步骤 2 直到没有顶点为止。

具体求解步骤如下：

(1) 在有向图中，从某个顶点出发进行深度优先遍历，并按其所有邻接点的访问都完成（即出栈）的顺序将顶点排列起来。

代码为：

```
void dfs(int x)
{
flag[x]=1;
int e=p[x];
while(e>0&&eflag[e]==0)//eflag[e]==0 表示 e 这条边是原图 G 的边
{
int k=b[e];
if(!flag[k])dfs(k);// 当 x 的邻接点 k 没有访问过，就递归搜索 k。
e=next[e];//x 的下一条边。
    }
stack1[++top]=x;// 当 x 的所有邻接点都访问完后，将 x 存入栈 stack1 中。
}
```

(2) 在该有向图中，从最后完成访问的顶点出发，沿着以该顶点为头的弧作逆向的深度优先遍历，若此次遍历不能访问到有向图中所有顶点，则从余下的顶点中最后完成访问的那个顶点出发，继续作逆向的深度优先遍历，依次类推，直至有向图中所有顶点都被访问到为止。

代码为：

```
void re_dfs(int x)
{
 flag[x]=1;
 attr[x]=ans;//x 的所属的连通分量的编号。
 int e=pp[x];
 while(e>0&&eflag[e]==1)//eflag[e]==1 表示 e 这条边是反图 GT 中的边
    {  int k=bb[e];
       if(!flag[k])re_dfs(k);
       e=nextn[e];
    }
}
```

(3) 每一次逆向深度优先遍历所访问到的顶点集便是该有向图的一个强连通分量的顶点集，若仅作一次逆向深度优先遍历就能访问到图的所有顶点，则该有向图是强连通图。

例如对下图 (a) 所示有向图，从顶点 v1 出发作深度优先遍历，在访问顶点 v2 后，顶点 v2 不存在未访问的邻接点从而从系统栈弹出，并进入栈 stack1，如图 (b) 所示。将 v2 从栈顶弹出后，再从顶点 v1 出发，在访问顶点 v3 v4 后，顶点 v4 不存在未访问的邻接点从而从系统栈弹出，并进入栈 stack1，如图 (c) 所示。将 v4 从栈顶弹出后，顶点 v3 不存在未访问的邻接点从而从系统栈弹出，并进入栈 stack1，将 v3 从栈顶弹出后，顶点 v1 不存在未访问的邻接点从而从系统栈弹出，并进入栈 stack1，将 v1 从栈顶弹出，所以，得到出栈的顶点序列为 v2, v4, v3, v1；再从最后一个出栈的顶点 v1 出发作逆向的深度优先遍历（逆着有向边的箭头方向），得到一个顶点集 { v1, v3, v4}，如图 (d) 所示；再从顶点 v2 出发作逆向的深度优先遍历，得到一个顶点集 {v2}，如图 (e) 所示。这就是该有向图的两个强连通分量的顶点集。

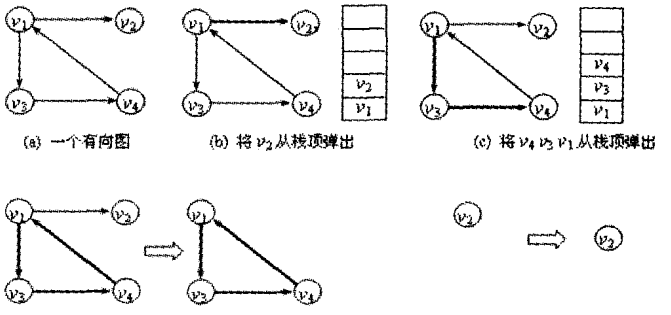


图 6-3 有向图的强连通分量的求解过程

部分程序：

```
void dfs(int x)
{  flag[x]=1;
   int e=p[x];
   while(e>0&&eflag[e]==0)
     {
       int k=b[e];
       if(!flag[k])dfs(k);
       e=next[e];
     }
   stack1[++top]=x;
}
void re_dfs(int x)
{
   flag[x]=1;
   attr[x]=ans;
   int e=pp[x];
   while(e>0&&eflag[e]==1)
     {
       int k=bb[e];
       if(!flag[k])re_dfs(k);
       e=nextn[e];
     }
}
```

```
void traver()
{for(inti=1;i<=n;i++)if(!flag[i])dfs(i);}// 从图中任意一个没有访问过的顶点开始 DFS
void re_traver()
{  memset(flag,0,sizeof(flag));
   while(top>=1)
   {
      int k=stack1[top--];
      if(!flag[k]){ re_dfs(k); ++ans; }
   }
}
```

Kosaraju 是基于对有向图及其逆图两次 DFS 的方法，其时间复杂度也是 O(n+e)。与 Trajan 算法相比，Kosaraju 算法可能会稍微更直观一些。但是 Tarjan 只用对原图进行一次 DFS，不用建立逆图，更简洁。在实际的测试中，Tarjan 算法的运行效率也比 Kosaraju 算法高 30% 左右。Kosaraju 虽然是线性的，但是需要两次 DFS，跟 Tarjan 和 Gabow 相比，这是一个劣势，但是 Kosaraju 算法有个神奇之处在于：计算之后的强分量编号的顺序，刚好是该有向图的一个拓扑排序！因此 Kosaraju 算法同时提供了一个计算有向图拓扑排序的线性算法。这个结果在一些应用中非常重要。

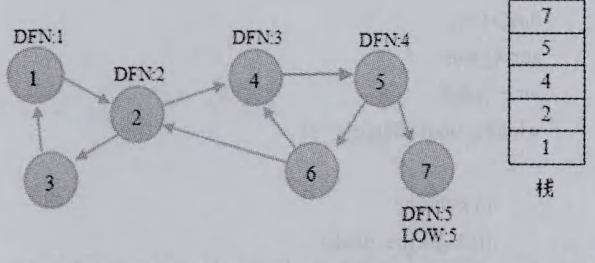
2 Tarjan 算法

Tarjan 由 Robert Tarjan 发明，它的思想是：强连通分量是 DFS 树中的子树（无论你如何进行 DFS）。

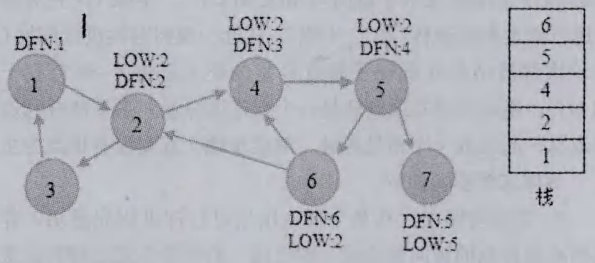
搜索时，任选一结点开始进行深度优先搜索（若深度优先搜索结束后仍有未访问的结点，则再从中任选一点再次进行）。搜索过程中已访问的结点不再访问。搜索树的若干子树构成了图的强连通分量。把当前搜索树中未处理的结点加入一个堆栈，回溯时可以判断栈顶到栈中的结点是否为一个强连通分量。定义 DFN(u) 为结点 u 搜索的次序编号（时间戳），LOW(u) 为 u 或 u 的子树能够追溯到的最早的栈中结点的次序号。由定义可以得出，当 DFN(u)=LOW(u) 时，以 u 为根的搜索子树上所有结点是一个强连通分量[6]。

接下来是对算法流程的演示。

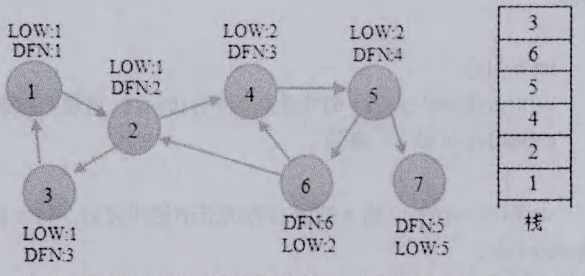
从结点 1 开始 DFS，把遍历到的结点加入栈中。搜索到结点 u=7 时，DFN[7]=LOW[7]，找到了一个强连通分量 {7}。退栈到 u=v 为止，{7} 为一个强连通分量。



返回结点 5，继续搜索 5 的其它邻接点 6，把 6 加入栈，此时，从 6 开始搜索邻接点 4 的时候发现 4 已经访问过并且还在栈中，更新 LOW[6]=DFN[4]=3，接着搜索 6 的邻接点 2 的时候发现 2 已经访问过并且 2 还在栈里面，再次更新 LOW[6]=DFN[2]=2，6 的邻接点访问完毕，5 的邻接点访问完毕，LOW[5]=LOW[6]=2,4 的邻接点访问完毕，LOW[4]=LOW[5]=2,2 的邻接点访问完毕，LOW[2]=2。从 4,5,6 这四个点搜索出发搜索完所有子结点后回来检查发现 LOW[4] 不等于 FN[4],LOW[5] 不等于 DFN[5],LOW[6] 不等于 DFN[6], 所以栈中存储了 1，2,4,5,6 五个结点。



返回结点 2，继续搜索到结点 3，把 3 加入栈，从 3 开始继续搜索到 1，发现 1 已经访问过并且仍然在栈中，此时更新 LOW[3]=DFN[1]=1，返回到 2 结点，更新 LOW[2]=LOW[3]=1，继续返回到结点 1，没有更新 LOW[1]，此时检查发现 LOW[1]=DFN[1]，所以将栈中结点全部出栈得到强连通分量 {3,6,5,4,2,1}。



至此，算法结束。经过该算法，求出了图中全部的两个强连通分量 {3,6,5,4,2,1},{7}。

部分程序：

```
void tarjan(int u)
{  ++index;
   dfn[u]=index;
   low[u]=index;
   stack[++top]=u;
   instack[u]=1;
   flag[u]=1;
   int e=p[u];
   while(e>0)
   {  int v=b[e];
      if(!flag[v])
      {
         tarjan(v);
         low[u]=min(low[u],low[v]);
      }
      else
       if(instack[v])
         low[u]=min(low[u],dfn[v]);
```

```
    e=next[e];
  }
  if(low[u]==dfn[u])
  { int j;
    do
    { j=stack[top--];
      instack[j]=0;
      cout<<j<<" ";
    }while(u!=j);
    cout<<endl;
  }
}
```

可以发现，运行 Tarjan 算法的过程中，每个顶点都被访问了一次，且只进出了一次堆栈，每条边也只被访问了一次，所以该算法的时间复杂度为 O(n+e)。(n 为顶点数，e 为边数)Gabow 算法是 Tarjan 算法的改进。

3 强连通分量的应用

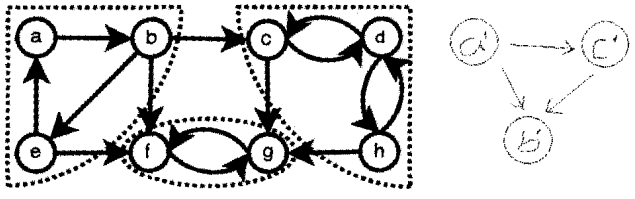
下面介绍一下有关“缩点”的概念：

由于强连通分量的特殊性，在一些实际应用中，会将每个强连通分量看成一个点，然后进行处理。这样做主要是为了降低图的复杂度，特别是在强连通分量规模大、数量多的情况中，利用“缩点”能大幅度降低图的复杂度。缩点在 ACM(大学生程序设计比赛) 和信息学奥林匹克竞赛中都有应用 [7]。

缩点后得到的图，必定是 DAG(有向无环图）。用反证法能够很方便的进行证明：因为若图中含有环路，即意味着至少有两个点彼此可达，那么按照强连通分量的定义，这两个点应该属于一个分量中，因而在缩点发生后，会被一个点所代表。由此推导出矛盾。比如，对下面左图进行缩点处理，最后的结果就是：

设 (a，b，c) -> a'，(f，g) -> b'，(c，d，h) -> c'

因此最后的图就可以表示为下面右图：



例题：

Poj2186 Popular Cows

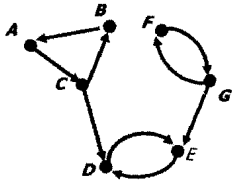
题目大意为：

有一群牛，总数为 N(N<=10000)。

其中奶牛 A 认为奶牛 B 备受注目，而奶牛 B 也可能认为奶牛 C 备受注目。奶牛们的这种“认为”是单向可传递的，就是说若奶牛 A 认为奶牛 B 备受注目，但奶牛 B 不一定会认为奶牛 A 备受注目。而当 A 认为 B 备受注目，且 B 认为 C 备受注目时，A 一定也认为 C 备受注目。

现在给出 M 对这样的“认为 ... 备受注目”的关系对，问有多少只奶牛被除其本身以外的所有奶牛关注。

样例数据



- 本例数据中最受欢迎的牛为
- D牛，E牛

普通解法：

通常，使用图中的弗洛伊德传递闭包方法，处理出所有牛注目的牛，也就求出了哪些牛是受其它所有牛的注目，以此方法可以解决问题。时间复杂度 O(n^3)。

这个题的 N 和 M 都非常大，暴搜肯定 TLE(Time Limit Exceeded)。那么有没有更好的做法呢？

分析：

首先可以想到的是，如果图 G 中包含有环，那么就可以把这个环缩成一个点，因为环中的任意两个点可以到达，环中所有的点具有相同的性质，即它们分别能到达的点集都是相同的，能够到达它们的点集也是相同的。

进一步分析：

那么是否只有环中的点才具有相同的性质呢？进一步的考虑，图中的每一个极大强连通分量中的点都具有相同的性质。所以，如果把图中的所有极大强连通分量求出后，对每个极大强连通分量缩点，就可以把图收缩成一棵有向无环树 DAG。

4 结束语：

只要判断出度为 0 的缩点是否只有 1 个，若 DAG 中有且仅有 1 个这样的缩点，则输出缩点（图 G 的极大强连通分量）内所包含的图 G 的结点个数，问题就解决。

时间复杂度 o（n+e）。

现在，强连通分量算法出场的时候到了。根据定义，一组互相注目的牛就构成了一个强连通分量（一个大牛）。

所以本题的正确解法是，先用 Tarjan 或 Kosaraju 算法求出所有的强连通分量，再求出每个强连通分量的出度，如果出度为 0 的强连通分量个数大于 1，则无解，因为只能有一组牛是最受注目的，不可能有两组牛都是最受注目的；否则，输出出度为 0 的强连通分量所包含的顶点的个数即可。

参考文献

[1] 严蔚敏 [M]《数据结构》清华大学出版社 北京 1992

[2] 刘汝佳 [M]《算法竞赛 训练指南》清华大学出版社 北京 2012

[3]Thomas H.Cormen 等 [M]《算法导论》机械工业出版社 北京 2012

[4] 刘汝佳 黄亮 [M]《算法艺术与信息学竞赛》清华大学出版社 北京 2004

[5] 吴文虎、王建德 [M]《信息学奥林匹克竞赛指导 -- 图论的算法与程序设计》清华大学出版社 1997

[6] 冯林 金博 于瑞云 [M]《图论及应用》哈尔滨工业大学出版社 哈尔滨 2012

[7] 伍鹏，谢凯 . 数据结构教学应注意的几个问题 [J]. 软件，2012,33（5）:123-124