强连通分支

Gabow算法

求解有向图强连通分量的Gabow算法 [编辑]

Gabow算法与Tarjan算法的核心思想实质上是相通的,就是利用强连通分量必定是DFS的一棵子树 这个重要性质,通过找出这个子树的根来求解强分量.具体到实现是利用一个栈S来保存DFS遇到的 所有树边的另一端顶点,在找出强分量子树的根之后,弹出S中的顶点一一进行编号. 二者不同的是,Tarjan算法通过一个low数组来维护各个顶点能到达的最小前序编号,而Gabow算法 通过维护另一个栈来取代low数组,将前序编号值更大的顶点都弹出,然后通过栈顶的那个顶点来判 断是否找到强分量子树的根

```
int Gabow(Graph G) {
  // 初始DFS用到的全局变量
  S = StackInit(G->V);  // S用来保存所有结点
  P = StackInit(G->V);  // P用来维护路劲
  int v;
  for (v = 0; v < G->V; ++v)
    pre[v] = G->sc[v] = -1;
  cnt = id = 0;
  // DFS
  for (v = 0; v < G->V; ++v)
    if (pre[v] == -1)
      GabowDFS(G, v);
  // 释放栈空间
  StackDestroy(S);
  StackDestroy(P);
  return id;  // 返回id的值,这恰好是强连通分量的个数
}

void GabowDFS(Graph G, int w) {
  Link t;
  int v;
  pre[w] = cnt++;  // 对前序编号编号
  StackPush(S, w);  // 讲路径上遇到的树边顶点入栈
  StackPush(P, w);
  for (t = G->adj[w]; t; t = t->next) {
    if (pre[v = t->v] == -1)                  // 如果当前顶点以前未遇到,则对其进行DFS
      GabowDFS(G, v);
    else if (G->sc[v] == - 1)                 // 否则如果当前顶点不属于强分量
      while (pre[StackTop(P)] > pre[v])  // 就将路径栈P中大于当前顶点pre值的顶点都弹出
        StackPop(P);
  }
  if (StackTop(P) == w) {  // 如果P栈顶元素等于w,则找到强分量的根,就是w
    StackPop(P);
    do {
      v = StackPop(S);  // 把S中的顶点弹出编号
      G->sc[v] = id;
    } while (v != w);
    ++id;
  }
}
```

图论及图论算法 [编辑]

图 - 有向图 - 无向图 - 连通图 - 强连通图 - 完全图 - 稀疏图 - 零图 - 树 - 网络

基本遍历算法：宽度优先搜索 - 深度优先搜索 - A* - 并查集求连通分支 - Flood Fill

最短路：Dijkstra - Bellman-Ford（SPFA） - Floyd-Warshall - Johnson算法

最小生成树：Prim - Kruskal

强连通分支：Kosaraju - Gabow - Tarjan

网络流：增广路法（Ford-Fulkerson，Edmonds-Karp，Dinic） - 预流推进 - Relabel-to-front

图匹配 - 二分图匹配：匈牙利算法 - Kuhn-Munkres - Edmonds' Blossom-Contraction

1个分类: 图论

导航

- 首页
- 社区主页
- 当前事件
- 最近更改
- 随机页面
- 使用帮助
- NOCOW地图
- 新手试练场

搜索

进入 搜索

工具箱

- 链入页面
- 链出更改
- 特殊页面
- 可打印版
- 永久链接

Kosaraju算法

这个算法的较为普遍的版本是:

```
1.对原图进行DFS并将出栈顺序进行逆序，得到的顺序就是拓扑顺序
2.将原图每条边进行反向
3.按照1中生成顺序再进行DFS染色染成同色的是一个强连通块
合理性：如果是强连通子图，那么反向没有任何影响，依然是强连通子图。
        但如果是单向的边连通，反向后再按原序就无法访问了（因此反向处理是对非强连通块的过滤）
```

下面的C++代码是对Kosaraju算法的简单演示

```
/*Kosaraju算法的简单演示，使用邻接矩阵储存图 */
/*
        title:Kosaraju's Algorithm
        author:JiangZX
        date:2011/8/18
*/
#include <iostream>
using namespace std;
const int MAXV = 1024;
int g[MAXV][MAXV], dfn[MAXV], num[MAXV], n, m, scc, cnt;
void dfs(int k)
{
        num[k] = 1;
        for(int i=1; i<=n; i++)
                if(g[k][i] && !num[i])
                        dfs(i);
        dfn[++cnt] = k;                                         //记录第cnt个出栈的顶点为k
}
void ndfs(int k)
{
        num[k] = scc;                                           //本次DFS染色的点，都属于同一个scc，
用num数组做记录
        for(int i=1; i<=n; i++)
                if(g[i][k] && !num[i])                          //注意我们访问的原矩阵的对称矩阵
                        ndfs(i);
}
void kosaraju()
{
        int i, j;
        for(i=1; i<=n; i++)                                     //DFS求得拓扑排序
                if(!num[i])
                        dfs(i);
        memset(num, 0, sizeof num);
        /*     我们本需对原图的边反向，但由于我们使用邻接矩阵储存图，所以反向的图的邻接矩阵
         即原图邻接矩阵的对角线对称矩阵，所以我们什么都不用做，只需访问对称矩阵即可*/
        for(i=n; i>=1; i--)
                if(!num[dfn[i]]){                               //按照拓扑序进行第二次DFS
                        scc++;
                        ndfs(dfn[i]);
                }
        cout<<"Found: "<<scc<<endl;
}
int main()
{
        int i, j;
        cin>>n>>m;
        for(i=1; i<=m; i++){
                int x, y, z;
                cin>>x>>y>>z;
                g[x][y] = z;
        }
        kosaraju();
        return 0;
}
```

```
pascal版本
                                ——市守沐Samuel
```

```
{
  Kosaraju Algorithm
     By Samuel
    2009.11.04
}
program Kosaraju;

var
   link,link2:array[0..110,0..110]of longint;
   a:array[0..110]of longint;
   p:longint;
   rec:array[0..110]of boolean;
   col:array[0..110]of longint;
   color,n,i,j,x:longint;

function max(a,b:longint):longint;
begin
  if a>b then exit(a) else exit(b);
end;
procedure sou(x:longint);
var
   i:longint;
begin
  rec[x]:=true;
  for i:=1 to link[x,0] do
    if not rec[link[x,i]] then sou(link[x,i]);
  inc(p);
  a[p]:=x;
end;

procedure sou2(x:longint);
var
   i:longint;
begin
  col[x]:=color;
  for i:=1 to link2[x,0] do
    if col[link2[x,i]]=0 then sou2(link2[x,i]);
end;

BEGIN
  //assign(input,inf);reset(input);
  //assign(output,ouf);rewrite(output);
  readln(n);
  for i:=1 to n do
  begin
    read(x);
    while x<>0 do
    begin
      inc(link[i,0]);
      link[i,link[i,0]]:=x;
      inc(link2[x,0]);
      link2[x,link2[x,0]]:=i;  { 2.将原图每条边进行反向}
      read(x);
    end;
    readln;
  end;
  fillchar(rec,sizeof(rec),0);
  for i:=1 to n do                { 1.对原图进行DFS↓}
    if not rec[i] then sou(i);
  color:=0;
  for i:=p downto 1 do          {并将出栈顺序a[i]进行逆序，得到的顺序就是拓扑顺序}
    if col[a[i]]=0 then
    begin
      inc(color);
      sou2(a[i]);               {3.按照1中生成顺序再进行DFS染色染成同色的是一个强连通块}
    end;
  //close(input);close(output);
END.
```

图论及图论算法 [编辑]

图 - 有向图 - 无向图 - 连通图 - 强连通图 - 完全图 - 稀疏图 - 零图 - 树 - 网络

基本遍历算法：宽度优先搜索 - 深度优先搜索 - A* - 并查集求连通分支 - Flood Fill

最短路：Dijkstra - Bellman-Ford（SPFA） - Floyd-Warshall - Johnson算法

最小生成树：Prim - Kruskal

强连通分支：Kosaraju - Gabow - Tarjan

网络流：增广路法（Ford-Fulkerson，Edmonds-Karp，Dinic） - 预流推进 - Relabel-to-front

图匹配 - 二分图匹配：匈牙利算法 - Kuhn-Munkres - Edmonds' Blossom-Contraction

1个分类: 图论

Tarjan算法

感谢Faint.Wisdom讲解求最近公共祖先(LCA)的Tarjan算法！

求最近公共祖先(LCA)的Tarjan算法 [编辑]

```
    首先，Tarjan算法是一种离线算法，也就是说，它要首先读入所有的询问（求一次LCA叫做一次询问），然后并不一定按
照原来的顺序处理这些询问。而打乱这个顺序正是这个算法的巧妙之处。看完下文，你便会发现，如果偏要按原来的顺序
处理询问，Tarjan算法将无法进行。
```

```
    Tarjan算法是利用并查集来实现的。它按DFS的顺序遍历整棵树。对于每个结点x，它进行以下几步操作：
    * 计算当前结点的层号lv[x]，并在并查集中建立仅包含x结点的集合，即root[x]:=x。
    * 依次处理与该结点关联的询问。
    * 递归处理x的所有孩子。
    * root[x]:=root[father[x]]（对于根结点来说，它的父结点可以任选一个，反正这是最后一步操作了）。
```

现在我们来观察正在处理与x结点关联的询问时并查集的情况。由于一个结点处理完毕后，它就被归到其父结点所在的集合，所以在已经处理过的结点中（包括 x本身），x结点本身构成了与x的LCA是x的集合，x结点的父结点及以x的所有已处理的兄弟结点为根的子树构成了与x的LCA是father[x]的集合，x结点的父结点的父结点及以x的父结点的所有已处理的兄弟结点为根的子树构成了与x的LCA是father[father[x]]的集合……（上面这几句话如果看着别扭，就分析一下句子成分，也可参照右面的图）假设有一个询问(x,y)（y是已处理的结点），在并查集中查到y所属集合的根是z，那么z 就是x和y的LCA，x到y的路径长度就是lv[x]+lv[y]-lv[z]*2。累加所有经过的路径长度就得到答案。 现在还有一个问题：上面提到的询问(x,y)中，y是已处理过的结点。那么，如果y尚未处理怎么办？其实很简单，只要在询问列表中加入两个询问(x, y)、(y,x)，那么就可以保证这两个询问有且仅有一个被处理了（暂时无法处理的那个就pass掉）。而形如(x,x)的询问则根本不必存储。 如果在并查集的实现中使用路径压缩等优化措施，一次查询的复杂度将可以认为是常数级的，整个算法也就是线性的了。

http://purety.jp/akisame/oi/TJU/

求有向图的强连通分支(SCC)的Tarjan算法 [编辑]

求有向图的强连通分支的Tarjan算法是以其发明者Robert Tarjan命名的。Robert Tarjan还发明了求双连通分量（割点、桥）的Tarjan算法，以及求最近公共祖先的Tarjan算法。

Tarjan算法是通过对原图进行一次DFS实现的。下面给出该算法的PASCAL语言模板：

```pascal
procedure dfs(s:int);
var ne:int;
begin
  view[s]:=1;                    //view[i]表示点i的访问状态.未访问,正访问,已访问的点,值分别为0,1,2
  inc(top); stack[top]:=s;       //当前点入栈
  inc(time); rea[s]:=time; low[s]:=time;  //记录访问该点的真实时间rea和最早时间low

  ne:=head[s];
  while ne<>0 do begin
    if view[e[ne]]=0 then dfs(e[ne]);      //如果扩展出的点未被访问,继续扩展
    if view[e[ne]]<2 then low[s]:=min(low[s],low[e[ne]]);
    //如果扩展出的不是已访问的点,更新访问源点s的最早时间.容易理解,如果一个点能到达之前访问过的点,那么路径中存在一
个环使它能更早被访问
    ne:=next[ne];
  end;

  if rea[s]=low[s] then begin              //如果s的最早访问时间等于其实际访问时间,则可把其视作回路的"始
点"
    inc(tot);                              //连通块编号
```

```
    while stack[top+1]<>s do begin        //将由s直接或间接扩展出的点标记为同一连通块,标记访问后出栈
      lab[stack[top]]:=tot;               //lab[i]表示点i所属的连通块
      view[stack[top]]:=2;
      dec(top);
    end;
  end;
end;
```

图是用邻接表存储的，e[i]表示第i条边指向的点。

算法运行过程中，每个顶点和每条边都被访问了一次，所以该算法的时间复杂度为O(V+E)。

下面是求强连通分量的Tarjan算法的C++实现

```
#define  M 5010              //题目中可能的最大点数
int STACK[M],top=0;          //Tarjan 算法中的栈
bool InStack[M];             //检查是否在栈中
int DFN[M];                  //深度优先搜索访问次序
int Low[M];                  //能追溯到的最早的次序
int ComponentNumber=0;        //有向图强连通分量个数
int Index=0;                 //索引号
vector <int> Edge[M];        //邻接表表示
vector <int> Component[M];    //获得强连通分量结果
int InComponent[M];    //记录每个点在第几号强连通分量里
int ComponentDegree[M];     //记录每个强连通分量的度
void Tarjan(int i)
{
    int j;
    DFN[i]=Low[i]=Index++;
    InStack[i]=true;
    STACK[++top]=i;
    for (int e=0;e<Edge[i].size();e++)
    {
        j=Edge[i][e];
        if (DFN[j]==-1)
        {
            Tarjan(j);
            Low[i]=min(Low[i],Low[j]);
        }
        else if (InStack[j])
            Low[i]=min(Low[i],DFN[j]);
    }
    if (DFN[i]==Low[i])
    {
        ComponentNumber++;
        do
        {
            j=STACK[top--];
            InStack[j]=false;
            Component[ComponentNumber].push_back(j);
   InComponent[j]=ComponentNumber;
        }
        while (j!=i);
    }
}

void solve(int N)     //N是此图中点的个数，注意是0-indexed!
{
    memset(STACK,-1,sizeof(STACK));
    memset(InStack,0,sizeof(InStack));
    memset(DFN,-1,sizeof(DFN));
    memset(Low,-1,sizeof(Low));

    for(int i=0;i<N;i++)
        if(DFN[i]==-1)
            Tarjan(i);
}
```

关于Tarjan算法的更为详细的讲解，可以在这里找到。

Tarjan的C++代码（STL）：

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <cstdlib>
#include <algorithm>
#include <list>
#include <stack>

using namespace std;
```

```
const int kMaxN = 3001;

class Graph {
 public:
  Graph(int vertex_count = 0) {
    vertex_count_ = vertex_count;
    memset(degree_, 0, sizeof(degree_));
  }
  void insert_edge(int v, int w) {
    graph_[v].push_back(w);
    degree_[w]++;
  }
  void TarjanInit() {
    tarjan_count = 0;
    memset(tarjan_dfn, 0, sizeof(tarjan_dfn));
    memset(tarjan_low, 0, sizeof(tarjan_low));
    memset(tarjan_instack, false, sizeof(tarjan_instack));
    for (int i = 1; i <= vertex_count_; i++) {
      tarjan_set[i] = i;
    }
  }
  void Tarjan(int v) {                  // Need TarjanInit()
    tarjan_count++;
    tarjan_dfn[v] = tarjan_count;
    tarjan_low[v] = tarjan_count;
    tarjan_stack.push(v);
    tarjan_instack[v] = true;

    for (list<int>::const_iterator i = graph_[v].begin(); i != graph_[v].end(); ++i) {
      if (!tarjan_dfn[*i]) {
        Tarjan(*i);
        tarjan_low[v] = min(tarjan_low[v], tarjan_low[*i]);
      } else if (tarjan_instack[*i]) {
        tarjan_low[v] = min(tarjan_low[v], tarjan_dfn[*i]);
      }
    }

    if (tarjan_dfn[v] == tarjan_low[v]) {
      while (tarjan_stack.top() != v) {
        tarjan_instack[tarjan_stack.top()] = false;
        tarjan_set[tarjan_stack.top()] = v;
        tarjan_stack.pop();
      }
      tarjan_instack[tarjan_stack.top()] = false;
      tarjan_set[tarjan_stack.top()] = v;
      tarjan_stack.pop();
    }
  }
  static bool compare(const int &a, const int &b) {
    return a < b;
  }
  void unique() {
    for (int v = 0; v < vertex_count_; v++) {
      graph_[v].sort(compare);
      graph_[v].unique();
    }
  }
  void BuildDAG(Graph &new_graph) {
    for (int v = 1; v <= vertex_count_; v++) {
      for (list<int>::const_iterator i = graph_[v].begin(); i != graph_[v].end(); ++i) {
        if (tarjan_set[v] == tarjan_set[*i]) {
          continue;
        } else {
          new_graph.insert_edge(tarjan_set[v], tarjan_set[*i]);
        }
      }
    }
    new_graph.unique();
  }
  int view_degree(int v) {
    return degree_[v];
  }

  void show(int v) {
    cout << "Vertex " << v << " : ";
    for (list<int>::const_iterator i = graph_[v].begin(); i != graph_[v].end(); ++i) {
      cout << *i << " ";
    }
    cout << endl;
  }
  void show_all() {
    for (int i = 1; i <= vertex_count_; i++) {
      show(i);
    }
  }

  int tarjan_count;
  int tarjan_set[kMaxN];
```

```
  int tarjan_dfn[kMaxN];
  int tarjan_low[kMaxN];
  bool tarjan_instack[kMaxN];
  stack<int> tarjan_stack;
 private:
  int vertex_count_;
  int degree_[kMaxN];
  list<int> graph_[kMaxN];
};

int n, p, r;
int buy[kMaxN];

int main() {
  ios::sync_with_stdio(false);
  cin >> n;
  Graph graph(n);
  Graph graph_dag(n);
  cin >> r;
  for (int i = 1; i <= r; i++) {
    int x, y;
    cin >> x >> y;
    graph.insert_edge(x, y);
  }

  graph.TarjanInit();
  for (int i = 1; i <= n; i++) {
    if (!graph.tarjan_dfn[i]) {
      graph.Tarjan(i);
    }
  }
  graph.BuildDAG(graph_dag);

  graph_dag.show_all();

  return 0;
}
```