北京大学暑期课《ICPC竞赛训练》

课程网页：http://acm.pku.edu.cn/summerschool/pku_acm_train.htm

郭 炜

微博：http://weibo.com/guoweiofpku

微信公众号

学会程序和算法，走遍天下都不怕!

讲义照片均为郭炜拍摄

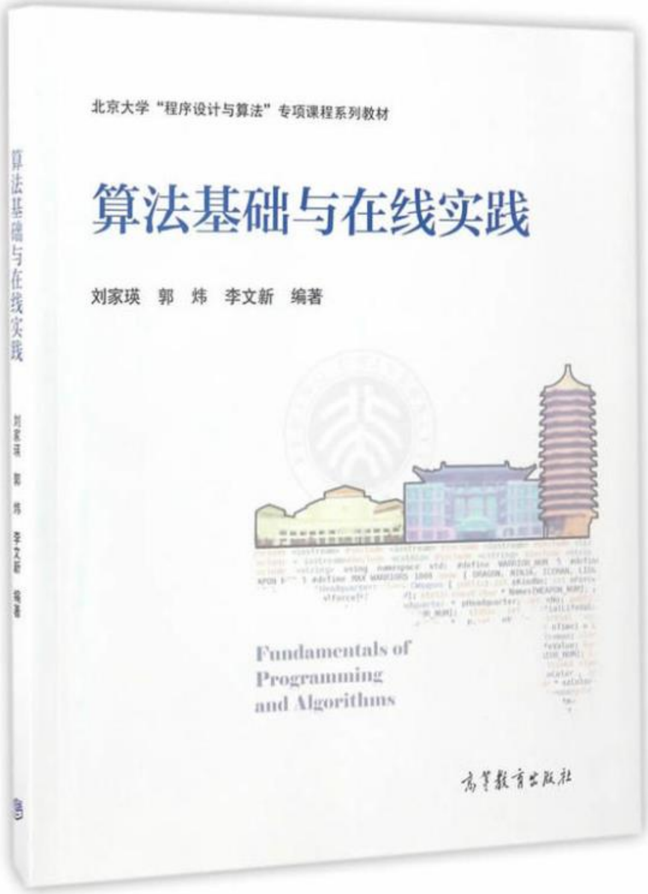
配套教材：

高等教育出版社

《算法基础与在线实践》

刘家瑛 郭炜 李文新 编著

本讲义中所有例题，根据题目名称在
http://openjudge.cn
“百练”组进行搜索即可提交



图论基础

拓扑排序

韩国济州岛火山口

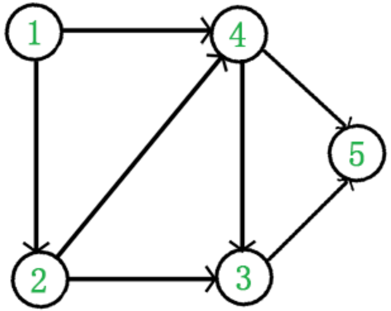
拓扑排序的概念

- 拓扑排序（Topological Sorting）：在有向无环图（DAG, Directed Acyclic Graph）中求一个顶点的序列，使其满足以下条件:

1）每个顶点出现且只出现一次

2）若存在一条从顶点 A 到顶点 B 的路径，那么在序列中顶点 A 出现在顶点 B 的前面

1,2,4,3,5



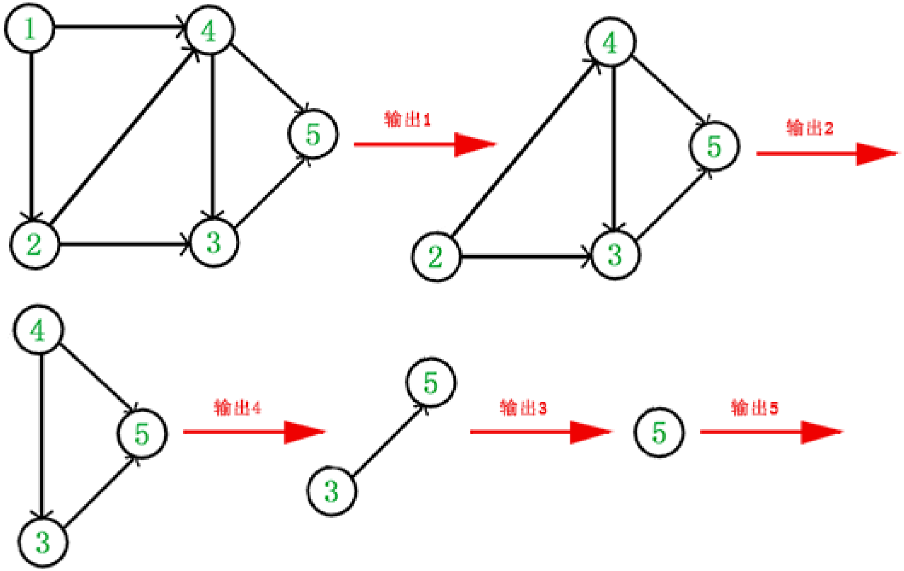
拓扑排序算法

1. 从图中任选一个没有前驱（入度为0）的顶点 x 输出

2. 从图中删除 x 和所有以它为起点的边

重复 1 和 2 直到图为空或当前图中不存在无前驱的顶点为止(后一种情况说明图中有环，无法拓扑排序)

具体实现：用队列存放入度变为0的点



例题：Genealogical tree

给一个有向无环图图，输出任一拓扑排序

样例输入

```
5          #5个点
0          #1号点没出边
4 5 1 0 #2号点有边连到 4,5, 1
1 0
5 3 0
3 0
```

样例输出

```
2 4 5 3 1
```

```python
import queue
n = int(input())
G = [[] for i in range(n+1)]
inDegree = [0] * (n+1) #G是邻接表，inDegree[i]是i的入度
for i in range(1,n+1):
    lst = list(map(int,input().split()))
    G[i] = lst[:-1]
q = queue.Queue()
for i in range(1,n+1):
    for v in G[i]:
        inDegree[v] += 1
for i in range(1,n+1):
    if inDegree[i] == 0:
        q.put(i)
seq = []
```

```
while not q.empty():
    k = q.get()
    seq.append(k)
    for v in G[k]:
        inDegree[v] -= 1   #删除边(k,v)后将v入度减1
        if inDegree[v] == 0:
            q.put(v)
if len(seq) != n:   #如果拓扑序列长度少于点数，则说明有环
    print("error")
else:
    for x in seq:
        print(x,end = " ")
    print("")
```

最小生成树

内蒙古阿斯哈图石林

图的生成树

- 在一个连通图G中，如果取它的全部顶点和一部分边构成一个子图G′，即：

  $$V(G')=V(G);E(G')\in E(G)$$

  若边集E(G′)中的边既将图中的所有顶点连通又不形成回路，则称子图G′是原图G的一棵生成树。

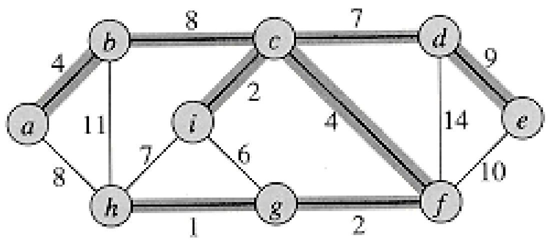
- 一棵含有n个点的生成树，必含有n-1条边。

最小生成树

- ✓ 对于一个连通带权图，每棵树的权（即树中所有边的权值总和）也可能不同
- ✓ 具有权最小的生成树称为最小生成树。

最小生成树

- 生成树
  - 无向连通图的边的集合
  - 无回路
  - 连接所有的点
- 最小
  - 所有边的权值之和最小

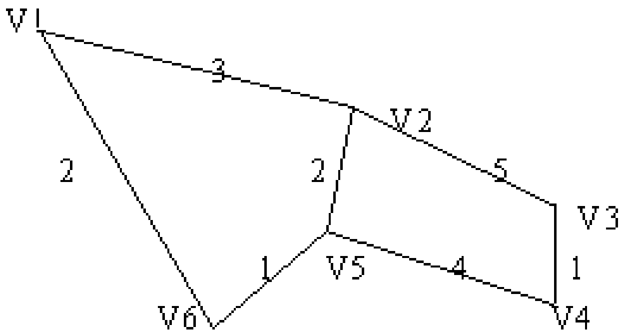


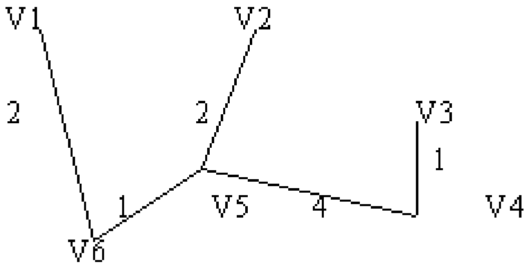
Prim算法

- 假设G=(V,E)是有n个顶点的带权连通图，T=(U,TE)是G的最小生成树，U,TE初值均为空集。

- 从V中任取一个顶点将它并入U中

- 每次从一个端点已在T中，另一个端点仍在T外的所有边中，找一条权值最小的，并把该边及端点并入T。做n-1次，T中就有n个点，n-1条边，T就是最小生成树





Prim算法实现

- 图节点数目为N,正在构造的生成树为T,
- 维护Dist数组,Dist[i]表示Vi到T的“距离”，即Vi和T中所有的点的连边的最小权值
- 开始所有Dist[i] = 无穷大, T 为空集

1) 若|T| = N，最小生成树完成。否则取Dist[i]最小的不在T中的点Vi, 将其加入T
2) 更新所有与Vi有边相连且不在T中的点Vj的Dist值：
   Dist[j] = min(Dist[j],W(Vi,Vj))
3) 转到1)

✓ 如果用邻接矩阵存放图，而且选取最短边的时候遍历所有点进行选取，则总时间复杂度为$O(V^2)$, V 为顶点个数

Prim算法加快选边速度

每次如何从连接T中和T外顶点的所有边中，找到一条最短的

- 1) 如果用邻接矩阵存放图，而且选取最短边的时候遍历所有点进行选取，则总时间复杂度为$O(V^2)$, V 为顶点个数

- 2)用邻接表存放图,并使用堆来选取最短边，则总时间复杂度为$O(ElogV)$

- 不加堆优化的Prim 算法适用于密集图，加堆优化的适用于稀疏图

POJ 1258 Agri-Net 最小生成树模版题

输入图的邻接矩阵，求最小生成树的总权值(多组数据)

输入样例:

4

0 4 9 21

4 0 8 17

9 8 0 16

21 17 16 0

输出样例:

28

Prim + 堆 完成POJ1258 Agri-Net

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>
using namespace std;
const int  INFINITE = 1 << 30;
struct Edge
{
        int v; //边端点，另一端点已知
        int w; //边权值，也用来表示v到在建最小生成树的距离
        Edge(int v_ = 0, int w_ = INFINITE):v(v_),w(w_) { }
        bool operator <(const Edge & e) const
        {
                return w > e.w;  //在队列里，边权值越小越优先
        }
};
vector< vector <Edge> > G(110); //图的邻接表
```

```
int HeapPrim(const vector<vector<Edge> > & G,  int n)
//G是邻接表,n是顶点数目，返回值是最小生成树权值和
{
        int i,j,k;
        Edge xDist(0,0);
        priority_queue<Edge> pq; //存放顶点及其到在建生成树的距离
        vector<int> vDist(n); //各顶点到已经建好的那部分树的距离
        vector<int> vUsed(n);//标记顶点是否已经被加入最小生成树
        int nDoneNum = 0; //已经被加入最小生成树的顶点数目
        for( i = 0;i < n;i ++ ) {
                vUsed[i] = 0;
                vDist[i] = INFINITE;
        }
        nDoneNum = 0;
        int nTotalW = 0; //最小生成树总权值
        pq.push(Edge(0,0)); //开始只有顶点0，它到最小生成树距离0
```

```
    while( nDoneNum < n && !pq.empty() ) {
            do {//每次从队列里面拿离在建生成树最近的点
                    xDist = pq.top();        pq.pop();
            } while( vUsed[xDist.v] == 1 && ! pq.empty());
            if( vUsed[xDist.v] == 0 ) {
                  nTotalW += xDist.w; vUsed[xDist.v] = 1;
                  nDoneNum ++;
                  for( i = 0;i < G[xDist.v].size();i ++ ) {
                    //更新新加入点的邻点
                    int k = G[xDist.v][i].v;
                    if( vUsed[k] == 0) {
                            int w = G[xDist.v][i].w ;
                            if( vDist[k] > w ) {
                                    vDist[k] = w; pq.push(Edge(k,w));
                            }
                    }
                  }
            }
    }
    if( nDoneNum < n )      return -1; //图不连通
    return nTotalW;
}
```

```
int main()
{
        int N;
        while(cin >> N) {
                for( int i = 0;i < N; ++i)
                        G[i].clear();
                for( int i = 0; i < N; ++i)
                        for( int j = 0; j < N; ++j) {
                                int w;
                                cin  >> w;
                                G[i].push_back(Edge(j,w));
                        }
                cout << HeapPrim(G,N) << endl;
        }
}
```

考察了所有的边，且考察一条边时 可能执行 pq.push(Edge(k,w)) 故复杂度O(ELogV)

Kruskal算法

- 假设G=(V,E)是一个具有n个顶点的连通网，T=(U,TE)是G的最小生成树，U=V,TE初值为空。

- 将图G中的边按权值从小到大依次选取，若选取的边使生成树不形成回路，则把它并入TE中，若形成回路则将其舍弃，直到TE中包含N-1条边为止，此时T为最小生成树。

关键问题

- ✓ **如何判断欲加入的一条边是否与生成树中边构成回路。**
- ✓ 将各顶点划分为所属集合的方法来解决，每个集合的表示一个无回路的子集。开始时边集为空，N个顶点分属N个集合，每个集合只有一个顶点，表示顶点之间互不连通。
- ✓ 当从边集中按顺序选取一条边时，若它的两个端点分属于不同的集合，则表明此边连通了两个不同的部分，因每个部分连通无回路，故连通后仍不会产生回路，此边保留，同时把相应两个集合合并
- ✓ 要用并查集

Kruskal算法完成POJ1258 Agri-Net

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
struct Edge
{
        int s,e,w; //起点，终点，权值
        Edge(int ss,int ee,int ww):s(ss),e(ee),w(ww) { }
        Edge() { }
        bool operator < (const Edge & e1) const {
                return w < e1.w;
        }
};
vector <Edge> edges;
vector <int> parent;
```

```
int GetRoot(int a)
{
        if( parent[a] == a)
                return a;
        parent[a] = GetRoot(parent[a]);
        return parent[a];
}
void Merge(int a,int b)
{
        int p1 = GetRoot(a);
        int p2 = GetRoot(b);
        if( p1 == p2)
                return;
        parent[p2] = p1;
}
```

```
int main()   {
        int N;
        while(cin >> N) {
                parent.clear();          edges.clear();
                for( int i = 0;i < N; ++i)   parent.push_back(i);
                for( int i = 0; i < N; ++i)
                        for( int j = 0; j < N; ++j) { int w;
                                cin  >> w;
                                edges.push_back(Edge(i,j,w));
                        }
                sort(edges.begin(),edges.end()); //排序复杂度O(ElogE)
                int done =  0;  int totalLen = 0;
                for( int i = 0;i < edges.size(); ++i) {
                        if( GetRoot(edges[i].s) != GetRoot(edges[i].e)) {
                                Merge(edges[i].s,edges[i].e);
                                ++done;         totalLen += edges[i].w;
                        }
                        if( done == N - 1) break;
                }
                cout << totalLen << endl;
        }
}
```

算法：Kruskal 和 Prim 的比较

- Kruskal:将所有边从小到大加入，在此过程中判断是否构成回路
  - 使用数据结构：并查集
  - 时间复杂度：O(ElogE)
  - 适用于稀疏图

- Prim:从任一节点出发，不断扩展
  - 使用数据结构：堆
  - 时间复杂度：O(ElogV) 或 O(VlogV+E)(斐波那契堆)
  - 适用于密集图
  - 若不用堆则时间复杂度为$O(V^2)$

例题： POJ 2349 Arctic Network

- 某地区共有n座村庄，每座村庄的坐标用一对整数(x, y)表示，现在要在村庄之间建立通讯网络。
- 通讯工具有两种，分别是需要铺设的普通线路和无线通讯的卫星设备。
- 只能给k个村庄配备卫星设备，拥有卫星设备的村庄互相间直接通讯。
- 铺设了线路的村庄之间也可以通讯。但是由于技术原因，两个村庄之间线路长度最多不能超过 d, 否则就会由于信号衰减导致通讯不可靠。要想增大 d 值，则会导致要投入更多的设备（成本）

例题： POJ 2349 Arctic Network

- 已知所有村庄的坐标 (x , y)， 卫星设备的数量 k 。
- 问：如何分配卫星设备，才能使各个村庄之间能直接或间接的通讯，并且 d 的值最小？ 求出 d 的最小值。
- 数据规模： 0 <= k <= n<= 500

思 路

- 把整个问题看做一个完全图，村庄就是点，图上两点之间的边的权值，就是两个村庄的直线距离。如果没有卫星设备，就只能铺电缆建一棵最小生成树

- 有了卫星设备，就可以去掉一些最小生成树上的边。k个卫星设备，就可以去掉k-1条边。

- 自然去掉最长的k-1条边

- d就是最小生成树上的第k长边

为什么d不可能比最小生成树第k长边更小？

建一棵生成树 T(未必是非最小生成树），去掉其最长的k-1条边，也是使用卫星设备的可行办法。T的第k长边为何一定不会比最小生成树的第k长边更短?

- 定理：若最小生成树的边按权值从大到小排序为

  $a_1,a_2,a_3.... a_{n-1}$

  某生成树的边按权值从大到小排序为

  $b_1,b_2,b_3.... b_{n-1}$

  则对任意 i， $a_i <= b_i$

- 推论：一个图的两棵不同最小生成树，边的权值序列排序后结果相同

● 定理证明:

将生成树T1中的一条边x去掉后，T1中的顶点被分成不连通的两部分G1和G2 （G1和G2都是点集，不考虑边）。在生成树T2中，必然有且只有一条边y，连接G1和G2。则称x,y是T1和T2中互为对应的两条边。两棵生成树的边必然一一对应。

最小生成树记位T1, 另一生成树记为 T2，令i为使得 ai > bi 成立的最小的 i。

ai和{ b_i， b_{i+1} .... b_{n-1}}中的任意边x，都不可能是对应关系（否则用x替换ai，可以得到比T1更小的生成树）。ai对应边位于{b_1,b_2...b_{i-1}}，则必有某个a_k(k<i)，其对应边y位于{b_i,b_{i+1}...b_{n-1}}。由于a_k > y，则用y替换a_k，则得到比T1更小的生成树，矛盾。

所以不存在一个i，使得a_i>b_i

2011 ACM/ICPC亚洲区预选赛北京赛站
Problem A. Qin Shi Huang’s National Road System

一个无向完全图，边有正权值，点也有正权值。可以选择一条边，将其边权值变为0。要求选定这条边(假定为e0)并将其权值变为0后，满足以下条件：A/B最大。其中A是e0连接的两个点的点权值和，B是修改后的图的最小生成树的边权值和。

解题思路：先求一棵最小生成树，求的过程中，每加入一个点，就记录已经在树上的所有点到该点的路径（树上的路径）上的最长边的权值。然后枚举权值要变成0的边uv，如果uv不是树边，则用它替换uv路径上的最大权值边，o(1)时间即得新最小生成树的边权值和；uv是树边，新最小生成树的边权值和即为原最小生成树的边权值和减去边uv的权值。

红色部分的做法:

- prim算法中，已经加入生成树的点集合为W
- 往W新增点s时，设 u 属于W,且 s是被连接到W中的v点的，
- 则

  Max_val[v][s] = 边(v,s)的权
  Max_val[u][s] = Max(Max_val[v][s], Max_val[u][v])

- 用时O(V^2)。

最短路问题

最短路
Dijkstra 算法

美国加州太浩湖

基本思想

- 解决无负权边的带权有向图或无向图的单源最短路问题
- 贪心思想，若离源点s前k-1近的点已经被确定，构成点集P，那么从s到离s第k近的点t的最短路径，$\{s,p_1,p_2\ldots p_i,t\}$满足$s,p_1,p_2\ldots p_i \in P$。
- 否则假设$pi \notin P$，则因为边权非负，pi到t的路径≥0，则$d[pi] \le d[t]$,pi才是第k近。将pi看作t，重复上面过程，最终一定会有找不到pi的情况
- $d[i]=\min(d[p_i]+cost(p_i,i)), i \notin P, p_i \in P$

  $d[t]=\min(d[i])\ , i \notin P$

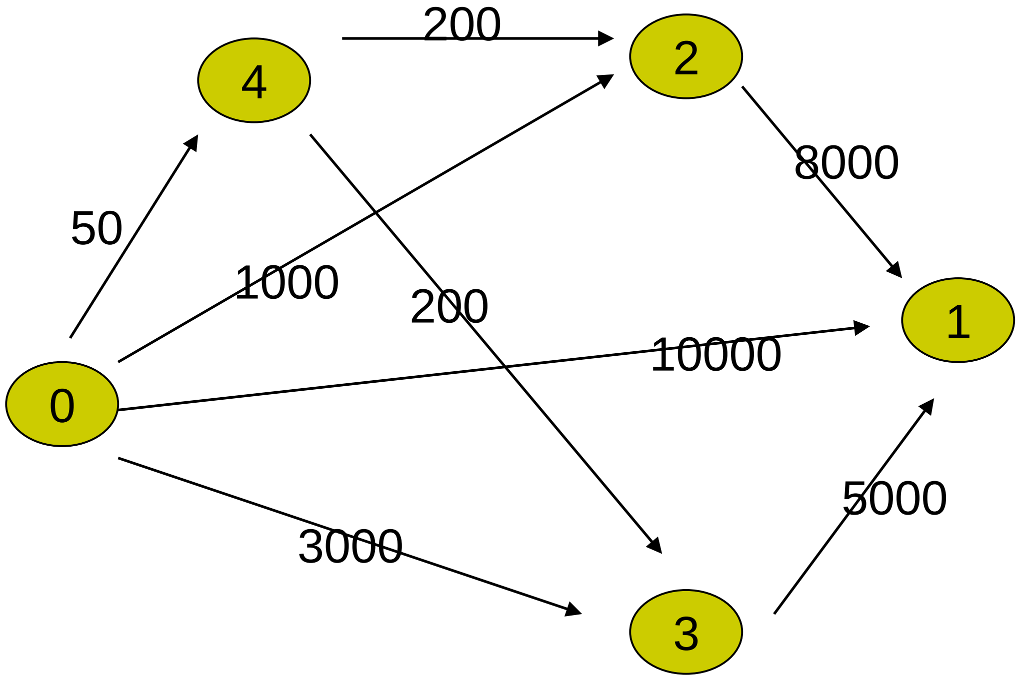
Dijkstra's Algorithm

- d[i]表示i点到起点s的距离
- 初始令d[s]=0，d[i]=+∞，P=∅
- 找到点i∉P，且d[i]最小
- 把i添入P，对于任意j∉P，若d[i]+cost(i,j)<d[j]，则更新d[j]=d[i]+cost(i,j)。

Dijkstra's Algorithm

- 用邻接表，不优化，时间复杂度$O(V^2+E)$
- Dijkstra+堆的时间复杂度 $o(ElgV)$
- 用斐波那契堆可以做到$O(VlogV+E)$

- 若要输出路径，则设置prev数组记录每个节点的前趋点，在d[i]更新时更新prev[i]



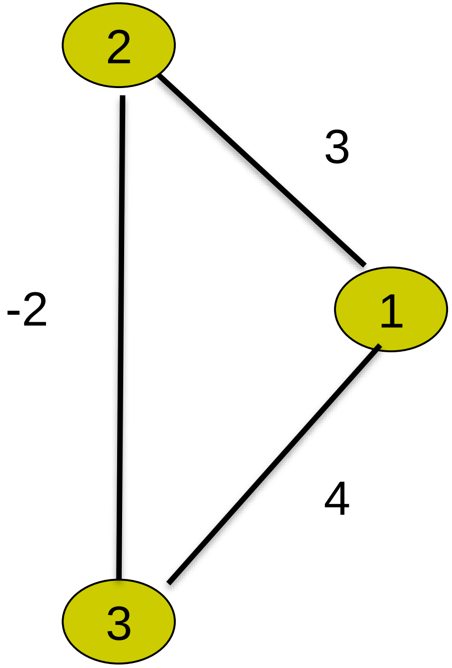
v	Dist[v]
0	0
1	10000 8250 5250
2	1000 250
3	3000 250
4	50

Dijkstra's Algorithm

Dijkstra算法也适用于无向图。但不适用于有负权边的图。

d[1,2] = 2

但用Dijkstra算法求得 d[1,2] = 3



Dijkstra算法实现

- 已经求出到V0点的最短路的点的集合为T
- 维护Dist数组，Dist[i]表示目前Vi到V0的“距离”
- 开始Dist[0] = 0, 其他Dist[i] = 无穷大, T为空集

- 1) 若|T| = N，算法完成，Dist数组就是解。否则取Dist[i]最小的不在T中的点Vi, 将其加入T，Dist[i]就是Vi到V0的最短路长度。
- 2) 更新所有与Vi有边相连且不在T中的点Vj的Dist值：
-     Dist[j] = min(Dist[j],Dist[i]+W(Vi,Vj))
- 3) 转到1）

POJ3159 Candies

有N个孩子（N<=3000)分糖果。
有M个关系(M<=150,000)。每个关系形如：

A B C (A,B,C是孩子编号)

表示A比B少的糖果数目，不能超过C

求第N个学生最多比第1个学生能多分几个糖果

POJ3159 Candies

思路：30000点，150000边的稀疏图求单源最短路

读入 “A B C”，就添加A->B的有向边，权值为C

然后求1到N的最短路

用prioirty_queue实现 dijkstra + 堆的 POJ 3159 Candies

```
#include <cstdio>
#include <iostream>
#include <vector>
#include <queue>
#include <cstring>
using namespace std;
struct CNode  {
        int k; //有向边的终点
        int w; //边权值，或当前k到源点的距离
};
bool operator < ( const CNode & d1, const CNode & d2 )
{  return d1.w > d2.w; } //priority_queue总是将最大的元素出列
priority_queue<CNode> pq;
bool bUsed[30010]={0}; // bUsed[i]为true表示源到i的最短路已经求出

vector<vector<CNode> > v; //v是整个图的邻接表
const unsigned int INFINITE = 100000000;
```

```
int main()
{
        int N,M,a,b,c;
        int i,j,k;
        CNode p;
        scanf("%d%d", & N, & M );
        v.clear();
        v.resize(N+1);
        memset( bUsed,0,sizeof(bUsed));
        for( i = 1;i <= M; i ++ ) {
                scanf("%d%d%d", & a, & b, & c);
                p.k = b;
                p.w = c;
                v[a].push_back( p);
        }
        p.k = 1; //源点是1号点
        p.w = 0; //1号点到自己的距离是0
        pq.push (p);
```

```
    while( !pq.empty ()) {
            p = pq.top ();
            pq.pop();
            if( bUsed[p.k])   //已经求出了最短路
                    continue;
            bUsed[p.k] = true;
            if( p.k == N ) //因只要求1-N的最短路，所以要break
                    break;
            for( i = 0, j = v[p.k].size(); i < j; i ++ ) {
                    CNode q; q.k = v[p.k][i].k;
                    if( bUsed[q.k] )  continue;
                    q.w = p.w + v[p.k][i].w ;
                    pq.push (q); //队列里面已经有q.k点也没关系
            }
    }
    printf("%d", p.w ) ;
    return 0;
}
```

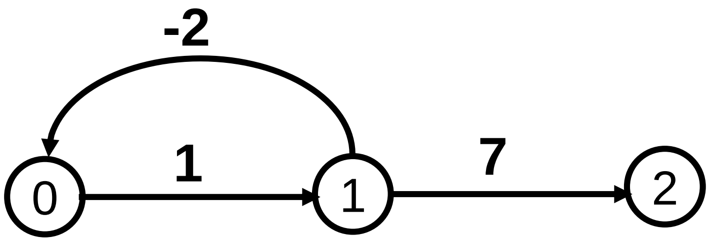
最短路
Bellman-Ford算法

美国黄石公园大棱镜温泉

Bellman-Ford算法

- 解决含负权边的带权有向图的单源最短路径问题
- 不能处理带负权边的无向图(因可以来回走一条负权边)
- 限制条件：

  要求图中不能包含权值总和为负值回路(负权值回路)，如下图所示。



(c)

Bellman-Ford算法思想

- 构造一个最短路径长度数组序列 $dist^1[u]$, $dist^2[u]$, …, $dist^{n-1}[u]$ (u = 0,1…n-1,n为点数)
  - $dist^1[u]$为从源点v到终点u的只经过一条边的最短路径长度，并有$dist^1[u]$ =Edge[v][u];
  - $dist^2[u]$为从源点v**最多经过两条边**到达终点u的最短路径长度;
  - $dist^3[u]$为从源点v出发**最多经过不构成负权值回路的三条边**到达终点u的最短路径长度;
  - ……
  - $dist^{n-1}[u]$为从源点v出发**最多经过不构成负权值回路的n-1条边**到达终点u的最短路径长度;
- 算法的最终目的是计算出$dist^{n-1}[u]$，为源点v到顶点u的最短路径长度。

$dist^{k}[u]$的计算

- **设已经求出 $dist^{k-1}[u]$，$u = 0, 1, \ldots, n-1$，即从源点v经过最多不构成负权值回路的$k-1$条边到达终点u的最短路径的长度**

递推公式(求顶点u到源点v的最短路径)：

$dist^{1}[u]$ = Edge[v][u]（若v->u无边，则 $dist^{1}[u]$ 为无穷大)

$dist^{k}[u]$ = min{ $dist^{k-1}[u]$, min{ $dist^{k-1}[j]$ + Edge[j][u] } }, j=0,1,…,n-1,j≠u

注意：具体实现时，若 $dist^{k-1}[j]$ 等于无穷大(即目前还没有发现从v到j的路)，则 $dist^{k-1}[j]$ + Edge[j][u] 也应是无穷大

Dijkstra算法与Bellman-Ford算法的区别

- Dijkstra算法在求解过程中，源点到集合S内各顶点的最短路径一旦求出，则之后不变了，修改的仅仅是源点到S外各顶点的最短路径长度。

- Bellman-Ford算法在求解过程中，每次循环都要修改所有顶点的dist[ ]，也就是说源点到各顶点最短路径长度一直要到算法结束才确定下来。

负权回路的判断

如果存在从源点可达的负权值回路，则最短路径不存在，因为可以重复走这个回路，使得路径长度无穷小。

思路：在求出$dist^{n-1}$[ ]之后，再对每条边<u,k>判断一下：加入这条边是否会使得顶点k的最短路径值再缩短，即判断：

$$dist[u]+w(u,k)<dist[k]$$

是否成立，如果成立，则说明存在从源点可达的负权值回路。

存在负权回路就一定能导致该式成立的证明略

负权回路的判断

证明：

如果成立，则说明找到了一条经过了n条边的从 s 到k的路径，且其比任何少于n条边的从s到k的路径都短。

一共n个顶点，路径却经过了n条边，则必有一个顶点m经过了至少两次。则m是一个回路的起点和终点。走这个回路比不走这个回路路径更短，只能说明这个回路是负权回路。

POJ3259 Wormholes

要求判断任意两点都能仅通过正边就互相可达的有向图(图中有重边）中是否存在负权环

Sample Input
2
3 3 1
1 2 2
1 3 4
2 3 1
3 1 3

3 2 1
1 2 3
2 3 4
3 1 8

Sample Output
NO
YES

2个test case
每个test case 第一行:

N M W (N<=500,M<=2500,W<=200)

N个点
M条双向正权边
W条单向负权边
第一个test case 最后一行
3 1 3
是单向负权边，3->1的边权值是-3

```
#include <iostream>
#include <vector>
using namespace std;
int F,N,M,W;
const int INF = 1 << 30;
struct Edge {
        int s,e,w;
        Edge(int ss,int ee,int ww):s(ss),e(ee),w(ww) { }
        Edge() { }
};
vector<Edge> edges; //所有的边
int dist[1000];
```

```
int Bellman_ford(int v)   {
        for( int i = 1; i <= N; ++i)
                dist[i] = INF;
        dist[v] = 0;
        for( int k = 1; k < N; ++k) { //经过不超过k条边
                for( int i = 0;i < edges.size(); ++i) {
                        int s = edges[i].s;
                        int e = edges[i].e;
                        if(dist[s] != INF &&
                                dist[s] + edges[i].w < dist[e])
                                dist[e] = dist[s] + edges[i].w;
                }
        }
        for( int i = 0;i < edges.size(); ++ i) {
                        int s = edges[i].s;
                        int e = edges[i].e;
                        if(dist[s] != INF &&
                                dist[s] + edges[i].w < dist[e])
                                return true;
        }
        return false;
}
```

```
int main()  {
        cin >> F;
        while( F--) {
                edges.clear();
                cin >> N >> M >> W;
                for( int i = 0; i < M; ++ i) {
                        int s,e,t;
                        cin >> s >> e >> t;
                        edges.push_back(Edge(s,e,t)); //双向边等于两条边
                        edges.push_back(Edge(e,s,t));
                }
                for( int i = 0;i < W; ++i) {
                        int s,e,t;
                        cin >> s >> e >> t;
                        edges.push_back(Edge(s,e,-t));
                }
                if( Bellman_ford(1))//从1可达所有点
                        cout << "YES" <<endl;
                else cout << "NO" <<endl;
        }
}
```

问题

```
for( int k = 1; k < N; ++k) { //经过不超过k条边
    for( int i = 0;i < edges.size(); ++i) {
        int s = edges[i].s;
        int e = edges[i].e;
        if( dist[s] + edges[i].w < dist[e])
            dist[e] = dist[s] + edges[i].w;
    }
}
```

会导致在一次内层循环中，更新了某个 dist[x]后，以后又用dist[x]去更新 dist[y]，这样dist[y]就是经过最多不超过k+1条边的情况了

出现这种情况没有关系，因为整个 for(int k = 1; k < N; ++k) 循环的目的是要确保,对任意点u,如果从源s到u的最短路是经过不超过n-1条边的,则这条最短路不会被忽略。至于计算过程中对某些点 v 计算出了从s->v的经过超过N-1条边的最短路的情况，也不影响结果正确性。若是从s->v的经过超过N-1条边的结果比经过最多N-1条边的结果更小，那一定就有负权回路。有负权回路的情况下，再多做任意多次循环，每次都会发现到有些点的最短路变得更短了。

算法复杂度分析

- 假设图的顶点个数为n，边的个数为e
  - 使用邻接表存储图，复杂度O(n*e)
  - 使用邻接矩阵存储图，复杂度为$O(n^3)$;

Bellman-Ford算法改进

Bellman-Ford算法不一定要循环n-1次，n为顶点个数

只要在某次循环过程中，考虑每条边后，源点到所有顶点的最短路径长度都没有变，那么Bellman-Ford算法就可以提前结束了

例题

- POJ 1860 3259 2240

最短路
SPFA算法

黄山

SPFA算法

- 快速求解含负权边的带权有向图的单源最短路径问题
- 是Bellman-Ford算法的改进版，利用队列动态更新dist[]

SPFA算法

●维护一个队列，里面存放所有需要进行迭代的点。初始时队列中只有一个源点S。用一个布尔数组记录每个点是否处在队列中。

●每次迭代，取出队头的点v，依次枚举从v出发的边v->u，若Dist[v]+len(v->u) 小于Dist[u]，则改进Dist[u]（可同时将u前驱记为v）。此时由于S到u的最短距离变小了，有可能u可以改进其它的点，所以若u不在队列中，就将它放入队尾。这样一直迭代下去直到队列变空，也就是S到所有节点的最短距离都确定下来，结束算法。**若一个点最短路被改进的次数达到n，则有负权环(原因同B-F算法)**。可以用spfa算法判断图有无负权环

●**在平均情况下，SPFA算法的期望时间复杂度为O(E)。**

POJ3259 Wormholes 判断有没有负权环spfa

```
//by guo wei
#include <iostream>
#include <vector>
#include <queue>
#include <cstring>
using namespace std;
int F,N,M,W;
const int INF = 1 << 30;
struct Edge {
        int e,w;
        Edge(int ee,int ww):e(ee),w(ww) { }
        Edge() { }
};
vector<Edge> G[1000]; //整个有向图

int updateTimes[1000]; //最短路的改进次数

int dist[1000]; //dist[i]是源到i的目前最短路长度
```

```
int Spfa(int v) {
        for( int i = 1; i <= N; ++i)
                dist[i] = INF;
        dist[v] = 0;
        queue<int> que;  que.push(v);
        memset(updateTimes ,0,sizeof(updateTimes));
        while( !que.empty()) {
                int s = que.front();
                que.pop();
                for( int i = 0;i < G[s].size(); ++i) {
                        int e = G[s][i].e;
                        if(dist[s] != INF &&
                                dist[e] > dist[s] + G[s][i].w ) {
                                dist[e] = dist[s] + G[s][i].w;
                                que.push(e); //没判队列里是否已经有e,可能会慢一些
                                ++updateTimes[e];
                                if( updateTimes[e] >= N) return true;
                        }
                }
        }
        return false;
}
```

```
int main(){
        cin >> F;
        while( F--) {
                cin >> N >> M >> W;
                for( int i = 1; i <1000; ++i)
                        G[i].clear();
                int s,e,t;
                for( int i = 0; i < M; ++ i) {
                        cin >> s >> e >> t;
                        G[s].push_back(Edge(e,t));
                        G[e].push_back(Edge(s,t));
                }
                for( int i = 0;i < W; ++i) {
                        cin >> s >> e >> t;
                        G[s].push_back(Edge(e,-t));
                }
                if( Spfa(1))
                        cout << "YES" <<endl;
                else    cout << "NO" <<endl;
        }
}
```

例题

POJ 2387
POJ 3256

最短路
弗洛伊德算法

华山

弗洛伊德算法

- 用于求每一对顶点之间的最短路径。有向图，无向图均可，也可以有负权边

弗洛伊德算法

- 用于求每一对顶点之间的最短路径。有向图，无向图均可，也可以有负权边
- 假设求从顶点v_i到v_j的最短路径。如果从v_i到v_j有边，则从v_i到v_j存在一条长度为cost[i,j]的路径，该路径不一定是最短路径，尚需进行n次试探。

弗洛伊德算法

- 用于求每一对顶点之间的最短路径。有向图，无向图均可，也可以有负权边
- 假设求从顶点v_i到v_j的最短路径。如果从v_i到v_j有边，则从v_i到v_j存在一条长度为cost[i,j]的路径，该路径不一定是最短路径，尚需进行n次试探。
- 考虑路径（v_i，v_1，v_j）是否存在（即判别弧（v_i，v_1）和（v_1，v_j）是否存在）。如果存在，则比较cost[i,j]和（v_i，v_1，v_j）的路径长度，取长度较短者为从v_i到v_j的中间顶点的序号不大于1的最短路径，记为新的cost[i,j]。

弗洛伊德算法

- 用于求每一对顶点之间的最短路径。有向图，无向图均可，也可以有负权边
- 假设求从顶点v_i到v_j的最短路径。如果从v_i到v_j有边，则从v_i到v_j存在一条长度为cost[i,j]的路径，该路径不一定是最短路径，尚需进行n次试探。
- 考虑路径（v_i，v_1，v_j）是否存在（即判别弧（v_i，v_1）和（v_1，v_j）是否存在）。如果存在，则比较cost[i,j]和（v_i，v_1，v_j）的路径长度，取长度较短者为从v_i到v_j的中间顶点的序号不大于1的最短路径，记为新的cost[i,j]。
- 假如在路径上再增加一个顶点v_2，如果（v_i，…，v_2）和（v_2，…，v_j）分别是当前找到的中间顶点的序号不大于2的最短路径，那么（v_i，…，v_2，…，v_j）就有可能是从v_i到v_j的中间顶点的序号不大于2的最短路径。将它和已经得到的从v_i到v_j的中间顶点的序号不大于1的最短路径相比较，从中选出中间顶点的序号不大于2的最短路径之后，再增加一个顶点v_3，继续进行试探。依次类推。

弗洛伊德算法

- 在一般情况下，若（v_i，…，v_k）和（v_k，…，v_j）分别是从v_i到v_k和从v_k到v_j的中间顶点的序号不大于k-1的最短路径，则将（v_i，…，v_k，…，v_j）和已经得到的从v_i到v_j且中间顶点的序号不大于k-1的最短路径相比较，其长度较短者便是从v_i到v_j的中间顶点的序号不大于k的最短路径。这样，在经过n次比较后，最后求得的必是从v_i到v_j的最短路径。按此方法，可以同时求得各对顶点间的最短路径。
- 复杂度$O(n^3)$。不能处理带负权回路的图

弗洛伊德算法伪代码

```
for( int i = 1 ;i <= vtxnum; ++i )
   for( int j = 1; j <= vtxnum; ++j)   {
        dist[i][j] = cost[i][j]; // cost是边权值，dist是两点间最短距离
        if( dist[i][j] < INFINITE) //i到j有边
                path[i,j] = [i]+[j]; //path是路径
   }
for( k = 1; k <= vtxnum; ++k) //每次求中间点标号不超过k的i到j最短路
   for( int i = 1; i <= vtxnum; ++i)
        for(int j = 1; j <= vtxnum ; ++j)
                if( dist[i][k] + dist[k][j] < dist[i][j]) {
                        dist[i][j] = dist[i][k]+dist[k][j];
                        path[i,j] = path[i,k]+path[k,j];
                }
```

例题： POJ3660 Cow Contest

N个选手，如果A比B强,B比C强，则A必比C强
告知若干个强弱关系，问有多少人的排名可以确定

- **Sample Input**

5 5

4 3

4 2

3 2

1 2

2 5

5个人，5个胜负关系

4 比3强
4 比2强
3 比2强

.....

- **Sample Output**

2

例题： POJ3660 Cow Contest

如果一个点u, 有x个点能到达此点，从u点出发能到达y个点，若x+y=N-1，则u点的排名是确定的。用floyd算出每两个点之间的距离，最后统计，若dist[a][b] 无穷大且dist[b][a]无穷大，则a和b的排名都不能确定。最后用点个数减去不能确定点的个数即可。

模版例题

POJ1125

连通性问题

有向图强连通分支

瑞士布里茨恩湖

有向图强连通分支的定义

- 在有向图G中，如果任意两个不同的顶点相互可达，则称该有向图是强连通的。有向图G的极大强连通子图称为G的强连通分支。

有向图强连通分支的Tarjan算法

- 用DFS的方法遍历有向图（每次任选没访问过的点作为起点），用dfn[i]表示编号为i的节点在整个DFS过程中的访问序号(也可以叫做开始时间）。在DFS过程中会形成一棵或若干棵搜索树。在一棵搜索树上越先遍历到的节点，显然dfn的值就越小。dfn值越小的节点，就称为越“早”。
- 用low[i]表示从i节点出发DFS过程中i下方节点(开始时间大于dfn[i]，且由i可达的节点）所能到达的最早的，在当前搜索路径上的节点的开始时间。初始时low[i]=dfn[i]

有向图强连通分支的Tarjan算法

- DFS过程中，碰到哪个节点，就将哪个节点入栈。栈中节点只有在其所属的强连通分量已经全部求出时，才会出栈。

- 如果发现某节点u有边连到当前搜索路径上的节点v，则更新u的low 值为min(low[u],dfn[v])，若low[u]被更新为dfn[v],则表明目前发现u可达的最早的节点是v.

有向图强连通分支的Tarjan算法

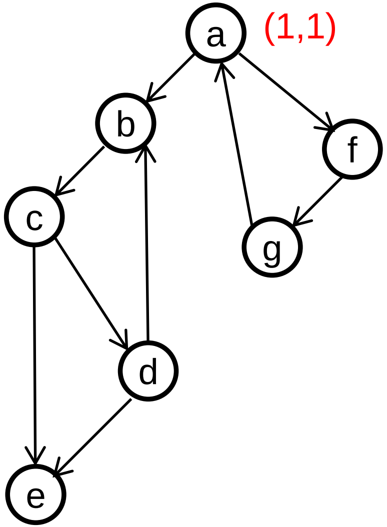
- 对于u的子节点v，从v出发进行的DFS结束回到u时，使得 low[u] = min(low[u],low[v])。因为u可达v,所以v可达的最早的节点，也是u可达的。
- 如果一个节点u，从其出发进行的DFS已经全部完成并回到u，而且此时其low值等于dfn值，则说明u可达的所有节点，都不能到达任何比u早的节点 ---- 那么该节点u就是一个强连通分量在DFS搜索树中的根。
- 此时，显然栈中u上方的节点，都是不能到达比u早的节点的。将栈中节点弹出，一直弹到u(包括u),弹出的节点就构成了一个强连通分量.

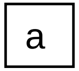
有向图强连通分支的Tarjan算法

```
void Tarjan(u) {
  dfn[u]=low[u]= ++index //index是开始时间
  stack.push(u)
  for each (u, v) in E { // E是边集合
        if (v is not visited) {
                tarjan(v)
                low[u] = min(low[u], low[v])
        }
        else if (v in stack) {
                low[u] = min(low[u], dfn[v])
        }
  }
  if (dfn[u] == low[u]) { //u是一个强连通分量的根
        repeat
                v = stack.pop
                print v
        until (u== v)
  } //退栈，把整个强连通分量都弹出来
} //复杂度是O(E+V)的
```

(dfn,low)

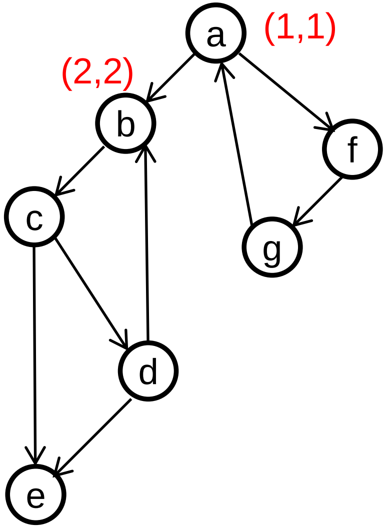
栈

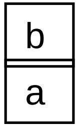




(dfn,low)

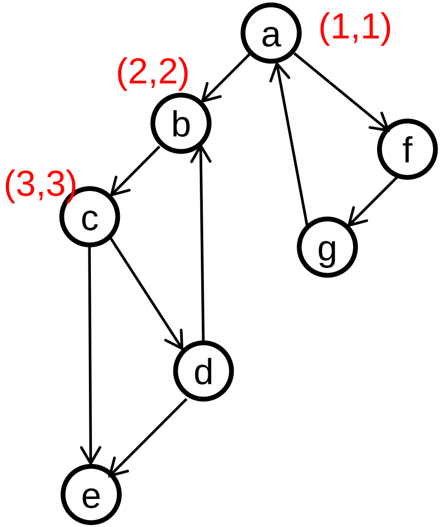
栈

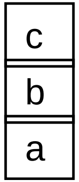




(dfn,low)

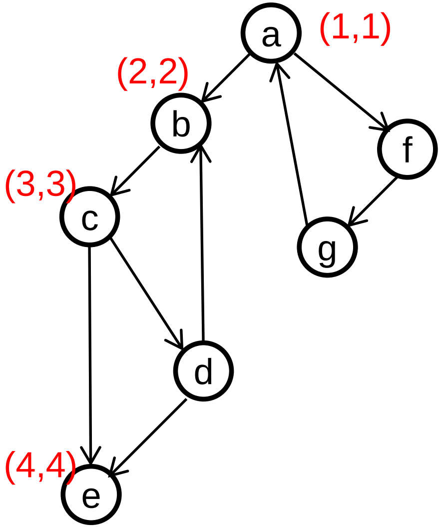
栈

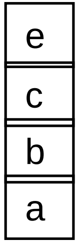




(dfn,low)

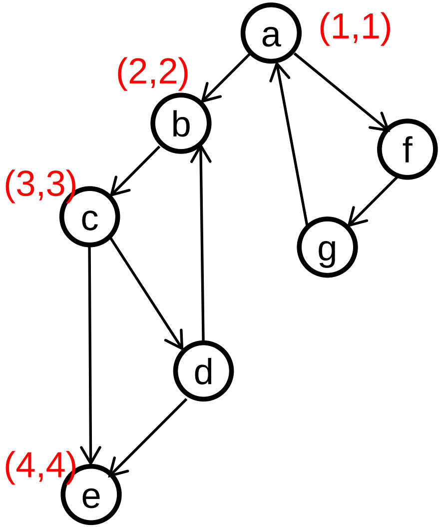
栈

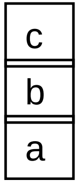




(dfn,low)

栈



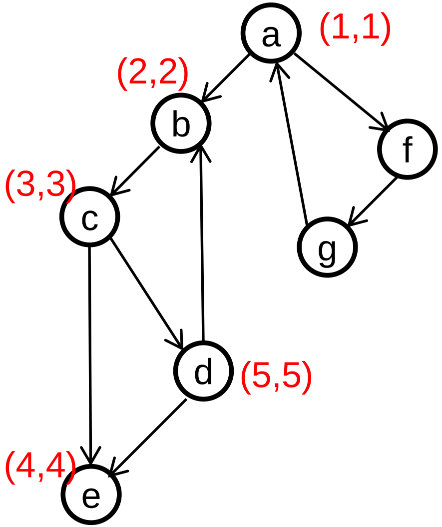


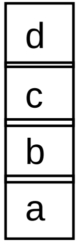
强连通分量:

{e}

(dfn,low)

栈



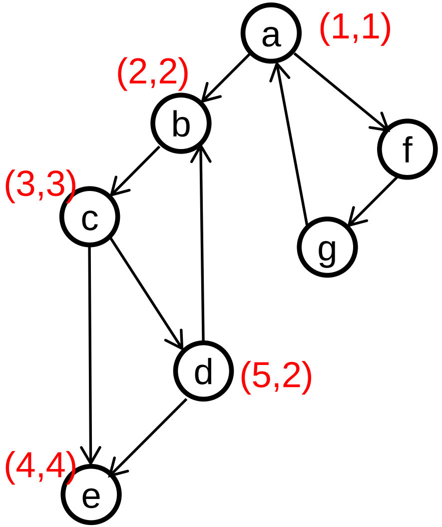


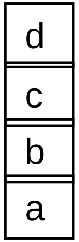
强连通分量：

{e}

(dfn,low)

栈

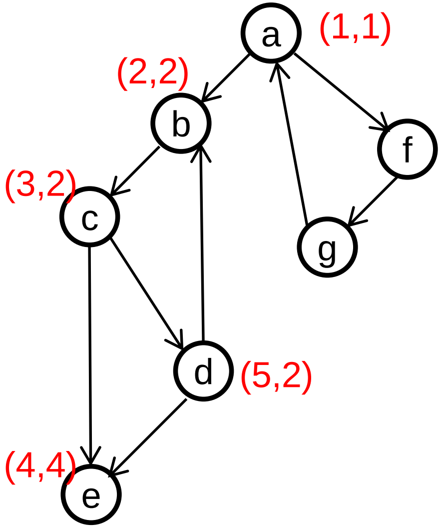




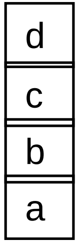
强连通分量:

{e}

(dfn,low)



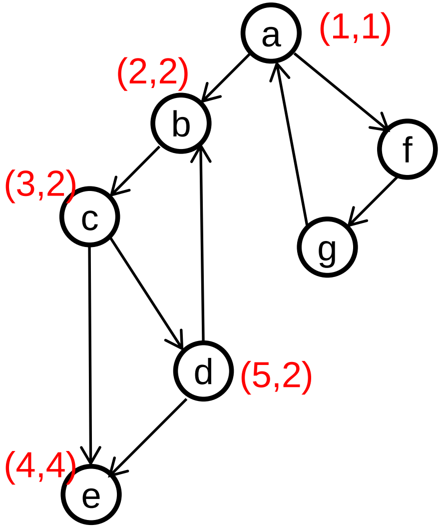
栈



强连通分量：

{e}

(dfn,low)



栈

a

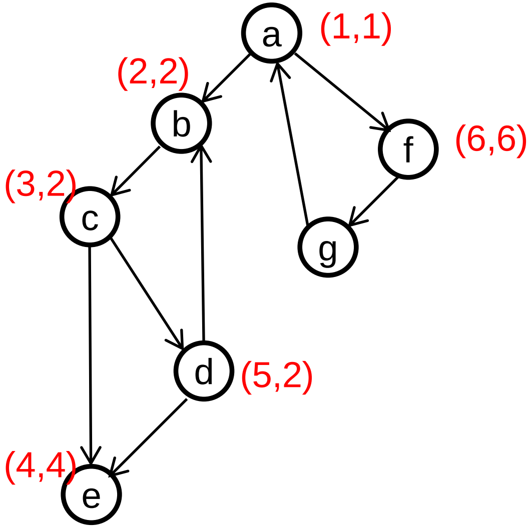
强连通分量:

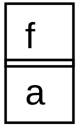
{e}

{b c d}

(dfn,low)

栈



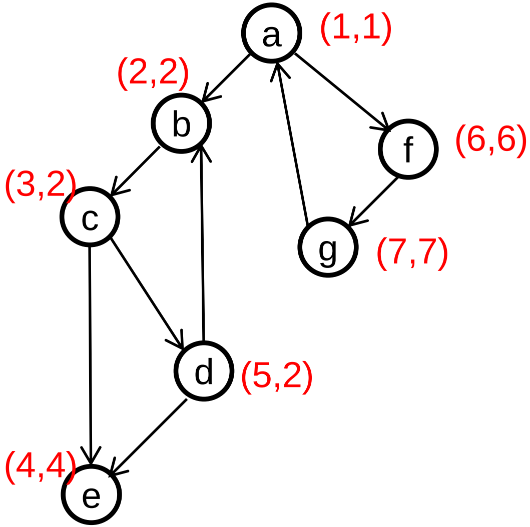


强连通分量:

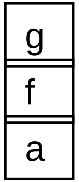
{e}

{b c d}

(dfn,low)



栈



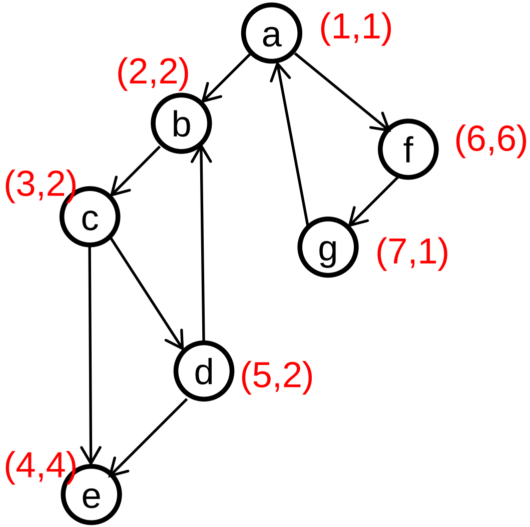
强连通分量：

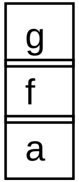
{e}

{b c d}

(dfn,low)

栈

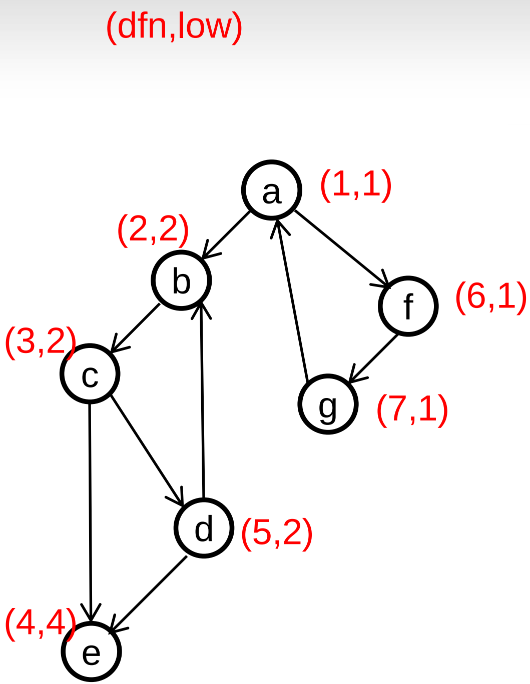




强连通分量:

{e}

{b c d}



栈

强连通分量：

{e}

{b c d}

{a f g}

有向图强连通分支的Tarjan算法

- 为什么从u出发的DFS全部结束回到u时，若dfn[u]=low[u], 此时将栈中u及其上方的节点弹出，就找到了一个强连通分量?

此时所有节点分成以下几类:

1） 还没被访问过的节点 —— 从u不可达
2) 栈中比u早的节点(栈中在u下方) —— 可达u，但从u不可达 因为low[u]=dfn[u]
3) 栈中比u晚的节点(栈中在u上方) —— 和u互相可达
4) 栈中的u —— 和u互相可达
5) 曾经在u之前入栈（访问过），又出了栈的节点 —— 不可达u 否则应该还在栈里，u下面
6) 曾经在u之后入栈（访问过），又出了栈的节点 —— 从u可达，但不可达u

有向图强连通分支的Tarjan算法

证：3）栈中比u晚的节点x(栈中在u上方) 和u互相可达

由u可达显然。

则寻找x所能到达的最早的，栈里面的节点y:

1) y比u早或y就是u: y可达u → x 也可达u
2) y比u晚: 不可能，因为：
   a) low[y] < dfn[y]: 不可能。low[y] < dfn[y] 说明y可达比y更早的节点，则x也可达比y更早的节点。这和y是x可达的最早的节点矛盾。
   b) low[y] = dfn[y]: 也不可能。若为真，则由y出发做DFS回到y时，栈中y及其上方节点应该已经被弹出栈了，y上方的x当然也已经不再栈中，这和x是第3类节点矛盾。

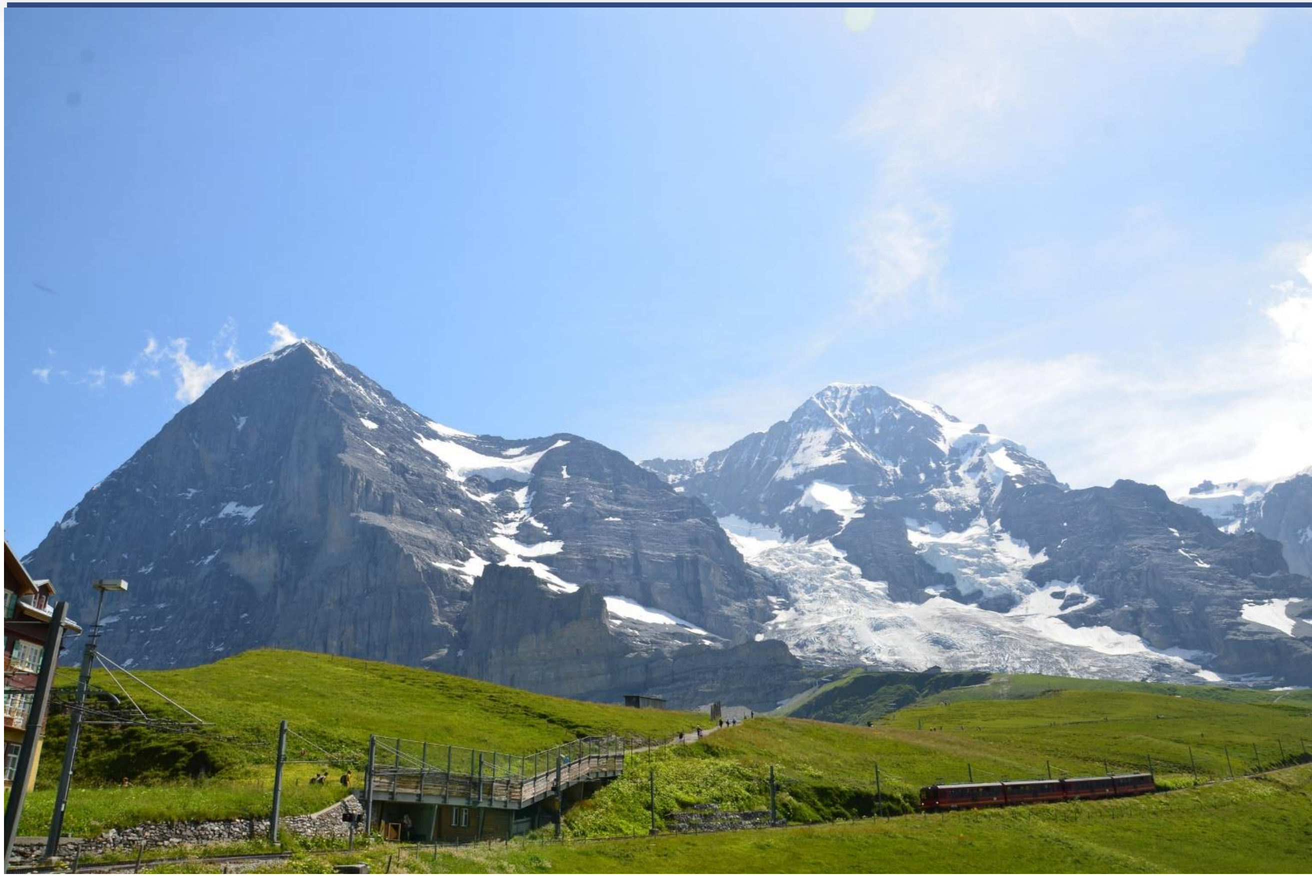
有向图强连通分支的Tarjan算法

- 证： 6）曾经在u之后入栈（访问过），又出了栈的节点 —— 从u可达，但不可达u

任取此类节点x。x之所以已经被弹出栈，定是因为以下2种原因之一：

1) low[x] = dfn[x]
2) 在栈里，x位于某个y节点上方(由y可达) ,且y满足条件:最终的 low[y] = dfn[y]。因为y曾经出现在u的上方，所以y一定晚于u。因为low[x]不可能小于等于dfn[u](否则low[y]就也会小于等于dfn[u],这和low[y]=dfn[y]矛盾），所以x到达不了u及比u早的节点。

例题1
Popular Cows



瑞士少女峰

例题1： POJ2186 Popular Cows

- 有N头牛。如果a喜欢b，b喜欢c，则a也会喜欢c。告诉你M个喜欢关系，比如(a,b)表示a喜欢b。问有多少头牛是被所有牛都喜欢的。

  N<= 10,000，M<= 50,000

例题1： POJ2186 Popular Cows

- 有N头牛。如果a喜欢b，b喜欢c，则a也会喜欢c。告诉你M个喜欢关系，比如(a,b)表示a喜欢b。问有多少头牛是被所有牛都喜欢的。
  N<= 10,000，M<= 50,000

- 本质：给定一个有向图，求有多少个顶点是由任何顶点出发都可达的。

例题1： POJ2186 Popular Cows

- 有用的定理:

有向无环图中唯一出度为0的点，一定可以由任何点出发均可达(由于无环，所以从任何点出发往前走，必然终止于一个出度为0的点)

例题1： POJ2186 Popular Cows

1. 求出所有强连通分量

2. 每个强连通分量缩成一点，则形成一个有向无环图DAG。

3. DAG上面如果有唯一的出度为0的点,则该点能被所有的点可达。那么该点所代表的连通分量上的所有的原图中的点，都能被原图中的所有点可达，则该连通分量的点数，就是答案。

4. DAG上面如果有不止一个出度为0的点，则这些点互相不可达，原问题无解，答案为0

例题1： POJ2186 Popular Cows

- 缩点构造新图：把不同强连通分量的点染不同颜色，有几种颜色，新图就有几个点。扫一遍老图所有的边，跨两种颜色的边，加到新图上（注意不要加了重边）。

  新图邻接表： vector< set<int> > G(colorNum);

  set便于去重

- 可以不构造新图，只要把不同强连通分量的点染不同颜色,然后考察各种颜色的点有没有连到别的颜色的边即可(即其对应的缩点后的DAG图上的点是否有出边）。

无向连通图求割点和桥

德国新天鹅堡

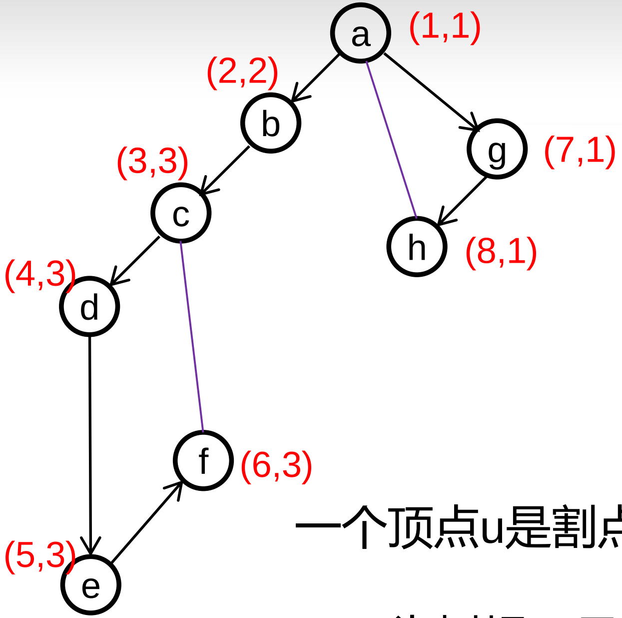
无向连通图求割点和桥

- 无向连通图中，如果删除某点后，图变成不连通，则称该点为割点。
- 无向连通图中，如果删除某边后，图变成不连通，则称该边为桥。

无向连通图求桥和割点的Tarjan算法

思路和有向图求强连通分量类似

深度优先遍历图形成一棵搜索树, dfn[u]定义和前面类似,
low[u]定义为u或者u的子树中能够通过非父子边(父子边就是搜索树上的边）到达的最早的节点的DFS开始时间



割点：

c b a

桥

ab
bc

一个顶点u是割点，当且仅当满足(1)或(2)

(1) u为树根，且u有多于一个子树。
(2) u不为树根，且存在(u,v)为树枝边(或称父子边，即u为v在搜索树中的父亲)，使得dfn(u)<=low(v)。

带箭头的边是树枝边
紫色边是反向边
非树枝边不可能是桥

一条边(u,v)是桥，当且仅当(u,v)为树枝边，且满足dfn(u)<low(v)（前提是其没有重边）。

无向连通图求桥和割点的Tarjan算法

```
Tarjan(u) {
    dfn[u]=low[u]=++index
    for each (u, v) in E {
        if (v is not visited)
            Tarjan(v)
            low[u] = min(low[u], low[v])
             dfn[u]<low[v] ⇔  (u, v) 是桥
        }
        else {
            if(v不是u 的父节点)
                low[u] = min(low[u], dfn[v])
        }
    }
    if (u is root)
        u 是割点 <=> u在搜索树上至少两个子节点
    else
        u 是割点 <=> u 有一个子节点v, 满足dfn[u]<= low[v]
}
```

无向连通图求桥和割点的Tarjan算法

- 也可以先用Tajan()进行dfs算出所有点的low和dfn值，并记录dfs过程中每个点的父节点，然后再把所有点看一遍，看其low和dfn,以找出割点和桥。

- 找桥的时候，要注意看有没有重边。有重边，则不是桥。

无重边连通无向图求割点和桥

Input: （11点13边）

11 13
1 2
1 4
1 5
1 6
2 11
2 3
4 3
4 9
5 8
5 7
6 7
7 10
11 3

output:
1
4
5
7
5,8
4,9
7,10

```
//无重边连通无向图求割点和桥的程序
#include <iostream>
#include <vector>
using namespace std;

#define MyMax 200
typedef vector<int> Edge;
vector<Edge> G(MyMax);
bool Visited[MyMax] ;
int dfn[MyMax] ;
int low[MyMax] ;
int Father[MyMax]; //DFS树中每个点的父节点
bool bIsCutVetext[MyMax]; //每个点是不是割点
int nTime; //Dfs时间戳
int n,m; //n是点数，m是边数
```

```
void Tarjan(int u, int father) //father 是u的父节点
{
      Father[u] = father;
      int i,j,k;
      low[u] = dfn[u] = nTime ++;
      for( i = 0;i < G[u].size() ;i ++ ) {
            int v = G[u][i];
            if( ! dfn[v]) {
                  Tarjan(v,u);
                  low[u] = min(low[u],low[v]);
            }
            else if( father != v ) //连到父节点的回边不考虑，否则求不出桥
                  low[u] = min(low[u],dfn[v]);
      }
}
```

```
void Count()
{ //计算割点和桥
        int i,nRootSons = 0;
        Tarjan(1,0);
        for( i = 2;i <= n;i ++ ) {
                int v = Father[i];
                if( v == 1 )
                        nRootSons ++; //DFS树中根节点有几个子树
                else if( dfn[v] <= low[i])
                        bIsCutVetext[v] = true;
        }
        if( nRootSons > 1)
                bIsCutVetext[1] = true;
        for( i = 1;i <= n;i ++ )
                if( bIsCutVetext[i] )
                        cout << i << endl;
        for( i = 1;i <= n;i ++) {
                int v = Father[i];
                if(v >0 &&  dfn[v] < low[i])
                        cout << v << "," << i <<endl;
        }
}
```

```
int main()
{
	int u,v;
	int i;

	nTime = 1;
	cin >> n >> m ; //n是点数，m是边数
	for(  i = 1;i <= m;i ++ ) {
		cin >> u >> v; //点编号从1开始
		G[v].push_back(u);
		G[u].push_back(v);
	}
	memset( dfn,0,sizeof(dfn));
	memset( Father,0,sizeof(Father));
	memset( bIsCutVetext,0,sizeof(bIsCutVetext));
	Count();
	return 0;
}
```