图论专题选讲2

Tarjan与连通性、回路问题

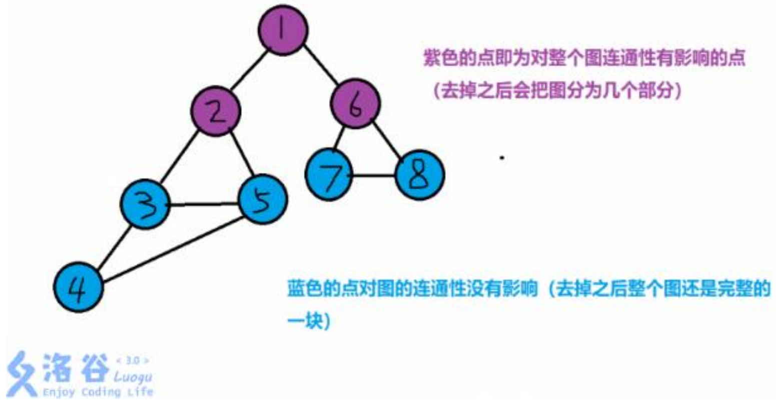
无向图连通性

Tarjan算法、割点和桥、双连通分量

割点与桥

无向图的割点

- 给定无向连通图$G = (V, E)$
- 若对于x∈V，从图中删去节点x以及所有与x关联的边后，G分裂成两个或两个以上不相连的子图，则称x为G的**割点**



割点与桥

无向图的桥

- 给定无向连通图$G = (V, E)$
- 若对于e∈E，从图中删去边e后，G分裂成两个不相连的子图，则称x为G的**桥**或**割点**

- 对于一般的无向图来说，它的割点和桥指的是每个连通块的割点和桥

TARJAN算法

Tarjan算法

- 准确地说是求图联通性的Tarjan算法（离线求LCA的算法也叫Tarjan）
- Tarjan算法不仅能用于无向图的连通性问题，也能用于有向图
- 由Lobert Tarjan提出，除了此之外，他还证明了并查集的复杂度，提出了斐波那契堆、Splay和LCT等数据结构

TARJAN算法

搜索树

- 在无向连通图中任选一个节点出发进行DFS，每个点只访问一次
- 所有发生递归的边(x,y)构成了一棵树，我们称之为**搜索树**
- 当然，对于一般的无向图来说一遍DFS形成的是**搜索森林**

时间戳

- 根据搜索树中节点的访问顺序我们可以标记时间戳，记为dfn[x]

TARJAN算法

回溯值

- 除了时间戳dfn[x]外，Tarjan算法还记录节点的回溯值low[x]
- low[x]定义为以下节点的时间戳的最小值：
- 1. 以x为根子树subtree(x)中的最小值
- 2. 通过一条不在搜索树上的边，能够到达subtree(x)的节点

TARJAN算法

回溯值的计算

- 根据定义，为计算low[x]，应先令low[x]=dfn[x]
- 然后遍历从x出发的所有边(x,y):
- 若在搜索树上x是y的父亲，则令low[x]=min(low[x],low[y])
- 若x不是y的父亲，即y∉subtree(x)，则令low[x]=min(low[x],dfn[y])

TARJAN求割边

割边(桥)判定法则

- 无向边(x,y)是桥，当且仅当搜索树上存在x的一个子节点y，满足：

$$dfn[x] < low[y]$$

性质

- 桥一定是搜索树上的边，一个简单环中的边一定都不是桥

TARJAN求割边

Tarjan算法求割边

```
void tarjan(int x, int in_edge) {
    dfn[x] = low[x] = ++num;
    for (int i = head[x]; i; i = Next[i]) {
        int y = ver[i];
        if (!dfn[y]) {
            tarjan(y, i);
            low[x] = min(low[x], low[y]);
            if (low[y] > dfn[x])
                bridge[i] = bridge[i ^ 1] = true;
        }
        else if (i != (in_edge ^ 1))
            low[x] = min(low[x], dfn[y]);
    }
}
```

TARJAN求割点

割点判定法则

- 若x不是搜索树的根节点，则x是割点当且仅当搜索树上存在x的一个子节点y，满足：

$$dfn[x] \le low[y]$$

特殊情况

- 如果x是搜索树的根节点，则需要至少两个子节点y_1和y_2，满足以上条件

TARJAN求割点

Tarjan算法求割点

```
int tarjan(int u,int fa)
{
    int child=0,lowu;
    lowu=dfn[u]=++deep;
    int sz=g[u].size();
    for(int i=0;i<sz;i++)
    {
        int v=g[u][i];
        if(!dfn[v])
        {
            child++;
            int lowv=tarjan(v,u);
            lowu=min(lowu,lowv);
            if(lowv>dfn[u])
            {
                iscut[u]=1;
            }
        }
```

TARJAN求割点

Tarjan算法求割点

```
        else
        {
            if(v!=fa&&dfn[v]<dfn[u])
            {
                lowu=min(lowu,dfn[v]);
            }
        }
    }
    if(fa<0&&child==1)
    {
        iscut[u]=false;
    }
    low[u]=lowu;
    return lowu;
}
```

双联通分量

双联通图

- 若一张无向连通图中不存在割点，则称之为点双联通图
- 若一张无向连通图中不存在桥，则称之为边双联通图

双联通分量

- 无向图的极大点双联通子图被称为点双联通分量，简记为v-DCC[1]
- 无向图的极大边双联通子图被称为边双联通分量，简记为e-DCC[2]

- 注1：v-DCC即vertex double connected component的缩写
- 注2：e-DCC即edge double connected component的缩写

双联通分量

定理1

- 若一张无向连通图是点双联通图，当且仅当满足以下两个条件之一：
- 1. 图中的顶点数不超过2
- 2. 图中的任意两点都同时包含在至少1个简单环[1]中

定理2

- 若一张无向连通图是点双联通图，当且仅当图中的任意一条边都包含在至少一个简单环中

- 注1：简单环指的是不自交的换

E-DCC

Tarjan求e-DCC

- 边双联通分量的计算很简单
- 只需要先求出无向图中所有的桥
- 把桥删除后剩下的所有联通块就是e-DCC

E-DCC

Tarjan求e-DCC

```
void dfs(int x) {
    c[x] = dcc;
    for (int i = head[x]; i; i = Next[i]) {
        int y = ver[i];
        if (c[y] || bridge[i]) continue;
        dfs(y);
    }
}
```

E-DCC

e-DCC的缩点

- 把每个e-DCC看作一个节点c[x]
- 把桥边看出连接两个e-DCC c[x]和c[y]的边
- 则无向连通图就被转化为一棵树
- 一般无向图就被转化为一片森林

E-DCC

e-DCC的缩点

```
tc = 1;
for (int i = 2; i <= tot; i++) {
    int x = ver[i ^ 1], y = ver[i];
    if (c[x] == c[y]) continue;
    add_c(c[x], c[y]);
}
```

V-DCC

Tarjan求v-DCC

- 点双联通分量的求解中一定要注意孤立点
- 除孤立点外，点双联通分量的大小至少为2
- 求解时需要维护一个栈，过程如下：
- 1. 当一个节点第一次被访问时，将节点入栈
- 2. 当dfn[x]≤low[y]成立时，无论x是否为根，都需要：
-   (1) 从栈顶不断弹出节点，直至节点y被弹出
-   (2) 刚才弹出的所有节点与节点x一起构成一个v-DCC

V-DCC

Tarjan求v-DCC

```
void tarjan(int x) {
    dfn[x] = low[x] = ++num;
    stack[++top] = x;
    if (x == root && head[x] == 0) { // 孤立点
        dcc[++cnt].push_back(x);
        return;
    }
    int flag = 0;
```

V-DCC

Tarjan求v-DCC

```
for (int i = head[x]; i; i = Next[i]) {
    int y = ver[i];
    if (!dfn[y]) {
        tarjan(y);
        low[x] = min(low[x], low[y]);
        if (low[y] >= dfn[x]) {
            flag++;
            if (x != root || flag > 1) cut[x] = true;
            cnt++;
            int z;
            do {
                z = stack[top--];
                dcc[cnt].push_back(z);
            } while (z != y);
            dcc[cnt].push_back(x);
        }
    }
    else low[x] = min(low[x], dfn[y]);
}
```

E-DCC

v-DCC的缩点

- v-DCC的缩点要复杂一些
- 因为一个割点可能属于多个v-DCC

具体做法

- 设图中有p个割点和t个v-DCC，我们可以建立一张p+v个节点的新图
- 其中所有的v-DCC作为新的节点和它所包含的所有割点连接起来
- 无向连通图就被转化为一棵树
- 一般无向图就被转化为一片森林

E-DCC

v-DCC的缩点

```
// 给每个割点一个新的编号(编号从cnt+1开始)
num = cnt;
for (int i = 1; i <= n; i++)
    if (cut[i]) new_id[i] = ++num;
// 建新图，从每个v-DCC到它包含的所有割点连边
tc = 1;
for (int i = 1; i <= cnt; i++)
    for (int j = 0; j < dcc[i].size(); j++) {
        int x = dcc[i][j];
        if (cut[x]) {
            add_c(i, new_id[x]);
            add_c(new_id[x], i);
        }
        else c[x] = i; // 除割点外，其它点仅属于1个v-DCC
    }
```

双联通分量

例题：Network(poj3694)

- 给定一张n点m边的无向图，q次操作，每次加上一条边并询问桥的数量。数据范围：$n \le 10^5$, $m \le 2*10^5$,$q \le 1000$

双联通分量

例题：Network(poj3694)

- 给定一张n点m边的无向连通图，q次操作，每次加上一条边并询问桥的数量。数据范围：$n \le 10^5$, $m \le 2*10^5$, $q \le 1000$

- 考虑如何在加入一条边后如何更新桥
- 显然一开始所有的桥构成了一棵树
- 如果加入的边x和y属于同一个e-DCC，显然桥的数量不变

双联通分量

例题：Network(poj3694)

- 给定一张n点m边的无向连通图，q次操作，每次加上一条边并询问桥的数量。数据范围：$n \le 10^5$, $m \le 2*10^5$, $q \le 1000$

- 如果x和y属于不同的e-DCC，那么c[x]与c[y]路径上的边与(x,y)构成一个简单环，即它们组成了一个新的v-DCC
- 此时桥的数量减少了，减少值等于c[x]到c[y]之间的距离
- 我们可以利用并查集维护点的合并情况
- 时间复杂度O(m+qlogn)

双联通分量

例题： Knights(poj2942)

- n个骑士，他们有些人之间有矛盾，现在要求选出一些骑士围成一圈，圈要满足如下条件：
- 1.人数大于1。2.总人数为奇数。3.有仇恨的骑士不能挨着坐。
- 问有几个骑士不能和任何人形成任何的圆圈。

- 数据范围： $n \le 1000, m \le 10^6$

双联通分量

例题：Knights(poj2942)

- 我们可以建原图的补图，即在没有憎恨关系的骑士之间连边
- 根据题意，我们需要找一个长度为奇数的环
- 所有不被任何奇环包含的点即所求的点
- 跑一遍Tarjan求出所有的v-DCC，然后检查每个v-DCC是否存在奇环即可
- 时间复杂度O(n+m)

- 当然判断奇环还可以使用染色法（二分图判断）一遍dfs搞定

有向图连通性

强联通分量、*必过点与必经边、2-SAT

流图

流图(Flow Graph)

- 给定有向图$G = (V, E)$
- 若对于r∈V，r能够到达V中的所有点，则称G一个**流图**记为(G,r)
- 其中r称为流图的源点

流图

搜索树

- 从一个流图(G,r)出发进行DFS，每个点只访问一次
- 所有发生递归的边(x,y)构成了一棵树，我们称之为**搜索树**

时间戳

- 根据搜索树中节点的访问顺序我们可以标记时间戳，记为dfn[x]

流图

搜索树

- 一个流图(G,r)中的每条边(x,y)必然属于以下四种之一：
- 1. 树枝边，搜索树中x是y的父亲
- 2. 前向边，搜索树中x是y的祖先
- 3. 后向边，搜索树中y是x的祖先
- 4. 横叉边，除了以上三种情况之外的边，满足dfn[y]<dfn[x]

强连通分量

强连通图

- 给定一张有向图，对于图中任意两点x和y，既存在x到y的路径，又存在y到x的路径，则称该图为强连通图

强连通分量

- 有向图的极大强连通子图被称为强连通分量，简记为SCC[1]
- Tarjan算法同样可以用于求有向图的SCC

- 注1：SCC即strongly connected component的缩写

强连通分量

Tarjan算法求SCC

- 显然树枝边和前向边并没有什么用处（因为和dfn排序相同）
- 我们需要利用后向边和横叉边构成的环

具体做法

- 可以维护一个栈，当访问节点x时，需要保存一下信息：
- 1. 搜索树上x的祖先节点，记为集合anc(x)
- 2. 以及访问过，并且存在一条路径到达anc(x)的节点

强连通分量

追溯值

- 设subtree(x)表示流图搜索树中以x为根的子树，则x的追溯值low[x]的定义为满足以下条件的最小时间戳：
- 1. 该点在栈中
- 2. 存在一条从subtree(x)出发的有向边，以该点为终点

强连通分量

追溯值求法

- 1. 当节点x第一次被访问时，将x入栈，令low[x]=dfn[x]
- 2. 扫描从x出发的每条边(x,y)
- (1) 若y没被访问过，则说明(x,y)是树枝边，递归访问y，回溯后令low[x]=min(low[x],dfn[y])
- (2) 若y被访问过且在栈中，则令low[x]=min(low[x],low[y])
- 3. 回溯之前，判断是否有low[x]=dfn[x]，若成立，则不断从栈中弹出x，直至x出栈

强连通分量

强连通分量判定法则

- 在追溯值的计算过程中，若从x回溯时，有low[x]=dfn[x]成立，则从栈中x到栈顶的所有节点构成一个强连通分量

强连通分量

Tarjan求SCC

```
void tarjan(int x) {
    dfn[x] = low[x] = ++num;
    stack[++top] = x, ins[x] = 1;
    for (int i = head[x]; i; i = Next[i])
        if (!dfn[ver[i]]) {
            tarjan(ver[i]);
            low[x] = min(low[x], low[ver[i]]);
        }
        else if (ins[ver[i]])
            low[x] = min(low[x], dfn[ver[i]]);
    if (dfn[x] == low[x]) {
        cnt++; int y;
        do {
            y = stack[top--], ins[y] = 0;
            c[y] = cnt, scc[cnt].push_back(y);
        } while (x != y);
    }
}
```

强连通分量

SCC缩点

- 和无向图中的e-DCC类似，我们可以把每个SCC缩成一个点
- 这样有向图就可以转换为一个DAG

有向图DP的一般步骤

- 缩点→拓扑排序→DP(SCC内部特判或解方程)

强连通分量

Tarjan求SCC缩点

```
for (int i = 1; i <= n; i++)
    if (!dfn[i]) tarjan(i);
for (int x = 1; x <= n; x++)
    for (int i = head[x]; i; i = Next[i]) {
        int y = ver[i];
        if (c[x] == c[y]) continue;
        add_c(c[x], c[y]);
    }
```

强连通分量

例题： Network of School(poj1236)

- 学校之间存在支援关系，若A学校支援B学校，则B学校能通过网络获得A学校的软件。
- 1. 问最少把软件给几个学校，让所有学校都能用软件
- 2. 问最少加几对支援关系，让所有学校都能用软件

强连通分量

例题：Network of School(poj1236)

- 学校之间存在支援关系，若A学校支援B学校，则B学校能通过网络获得A学校的软件。
- 1. 问最少把软件给几个学校，让所有学校都能用软件

- 第一问可以用Tarjan算法进行缩点，然后转成DAG后求有几个点的入度为0即可

强连通分量

例题：Network of School(poj1236)

- 学校之间存在支援关系，若A学校支援B学校，则B学校能通过网络获得A学校的软件。
- 2. 问最少加几对支援关系，让所有学校都能用软件

- 假设缩点后的DAG上有p个点入度为0，q个点出度为0
- 那么答案就是max(p,q)

强连通分量

例题： Network of School(poj1236)

- 学校之间存在支援关系，若A学校支援B学校，则B学校能通过网络获得A学校的软件。
- 2. 问最少加几对支援关系，让所有学校都能用软件

- 假设缩点后的DAG上有p个点入度为0，q个点出度为0
- 那么答案就是max(p,q)

- 特殊情况：整个图为SCC时，答案为0

*必经点与必经边

必经点

- 给定有向图，起点S，终点T
- 若S到T的每一条路径都经过点x，则称x是有向图中从S到T的必经点

必经边

- 给定有向图，起点S，终点T
- 若S到T的每一条路径都经过边(x,y)，则称(x,y)是有向图中从S到T的必经边

*必经点与必经边

求必经点与必经边

- 对于一般的有向图，求必经点与必经边不能直接将SCC缩点
- 若S到T的每一条路径都经过边(x,y)，则称(x,y)是有向图中从S到T的必经边
- 使用Lenguar-Tarjan算法计算支配树，能够在O(nlogn)时间内求出指定起点到每个点的必经边

- 具体参见：*《图连通性若干拓展问题探讨》*，*李煜东*，WC2014

*必经点与必经边

求DAG的必经点与必经边

- 对于DAG上的必经点和必经边，可以用DP求解：
- 1. 在原图中按照拓扑排序进行DP，求起点S到每个点x的路径数fs[x]
- 2. 在反图中按照拓扑排序进行DP，求终点T到每个点x的路径数ft[x]

*必经点与必经边

求DAG的必经点与必经边

- 对于DAG上的必经点和必经边，可以用DP求解：
- 1. 在原图中按照拓扑排序进行DP，求起点S到每个点x的路径数fs[x]
- 2. 在反图中按照拓扑排序进行DP，求终点T到每个点x的路径数ft[x]

- 显然，fs[T]=ft[S]为从起点到终点的路径数
- 对于一条有向边(x,y)，若fs[x]*ft[y]=fs[T]，则其为S到T的必经边
- 对于一顶点x，若fs[x]*ft[x]=fs[T]，则其为S到T必经点
- 对于大数据可以用双哈希（即多个模数）来代替高精度

2-SAT问题

2-SAT问题

- n个变量，每个变量只有两种可能的取值
- 给定m组条件，每组条件都是对两个变量的限制
- 判断所有的条件是否能够得到满足，
- SAT即satisfiability（可满足性）的缩写

2-SAT问题

形式化定义

- n个变量Ai，每个变量取值为0或1
- m个限制条件：若Ai取x，则Aj必须取y
- 问所有限制条件能否满足

2-SAT问题

判定方法

- 1. 为每个变量Ai建两个节点，一般为i和i+N
- 2. 对于“若Ai取x则Aj必须取y”的条件，在i+x*N与j+y*N之间连一条有向边
- 3. 若原命题的逆否命题不在限制条件中，则在j+(1-y)*N与i+(1-x)*N之间连一条有向边
- 4. Tarjan算法求SCC，若i与i+N处于同一个SCC中则条件无法满足

2-SAT问题

判定方法

- 1. 为每个变量Ai建两个节点，一般为i和i+N
- 2. 对于“若Ai取x则Aj必须取y”的条件，在i+x*N与j+y*N之间连一条有向边
- 3. 若原命题的逆否命题不在限制条件中，则在j+(1-y)*N与i+(1-x)*N之间连一条有向边
- 4. Tarjan算法求SCC，若i与i+N处于同一个SCC中则条件无法满足

- 时间复杂度O(n+m)

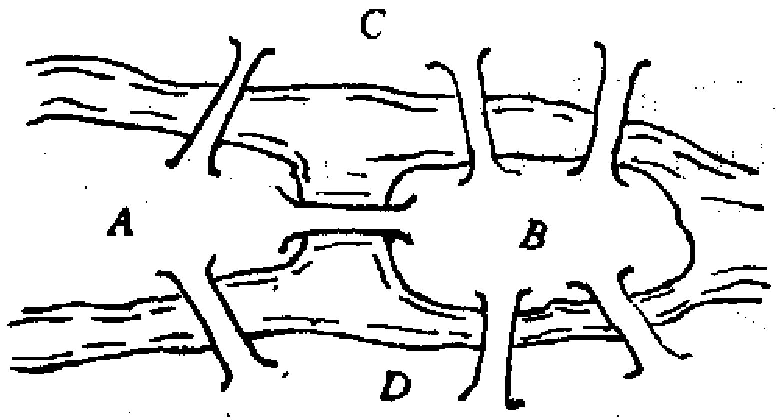
回路问题

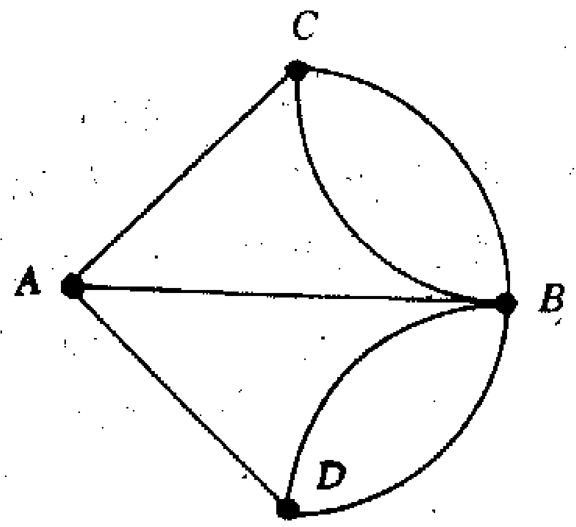
欧拉回路、*哈密尔顿回路

欧拉回路

哥尼斯堡七桥问题

- 问能否从起点出发经过七座桥一次并回到起点





欧拉回路

欧拉路(Euler Path)

- 对于无向图G，从一个点S出发经过所有的边一次，到达终点T，则称这条路径为欧拉路

欧拉回路(Euler Route)

- 对于无向图G，从一个点S出发经过所有的边一次，回到起点S，则称这条路径为欧拉回路
- 我们将存在欧拉回路的图称为欧拉图

欧拉回路

欧拉路的判断

- 对于无向图G，存在欧拉路当且仅当其度数为奇数的节点不超过2个
- 这两个奇数度数的节点就是起点S和终点T

欧拉图的判断

- 对于无向图G，它是欧拉图当且仅当所有节点的度数为偶数

欧拉回路

dfs求欧拉回路

- 1. 对于从x出发的每条边(x,y)，如果没有被访问过则：
- 2. 将边标记为访问，并递归节点y，回溯y后将y入栈
- 3. 最后倒序输出栈中的节点就是一条欧拉回路

- 欧拉路的求法只要先找到度数为奇数的起点，然后进行dfs即可
- 另外dfs递归容易爆栈，可以手动模拟递归

欧拉回路

dfs求欧拉回路

```
void euler() {
    stack[++top] = 1;
    while (top > 0) {
        int x = stack[top], i = head[x];
        // 找到一条尚未访问的边
        while (i && vis[i]) i = Next[i];
        // 沿着这条边模拟递归过程，标记该边，并更新表头
        if (i) {
            stack[++top] = ver[i];
            head[x] = Next[i];
            vis[i] = vis[i ^ 1] = true;
        }
        // 与x相连的所有边均已访问，模拟回溯过程，并记录于答案栈中
        else {
            top--;
            ans[++t] = x;
        }
    }
}
```

*哈密尔顿回路

哈密尔顿通路(Hamilton Path)

- 对于无向图G，从一个点S出发经过所有的节点一次，到达起点T，则称这条路径为哈密尔顿通路

哈密尔顿回路(Hamilton Route)

- 对于无向图G，从一个点S出发经过所有的节点一次，回到起点S，则称这条路径为哈密尔顿回路
- 我们称存在哈密尔顿回路的图为哈密尔顿图

*哈密尔顿回路

求哈密尔顿回路

- 无论是有向图还是无向图，求一般图中的哈密尔顿问题是NP问题
- 一般只讨论特殊图中的哈密尔顿回路

*哈密尔顿回路

求哈密尔顿回路

- 无论是有向图还是无向图，求一般图中的哈密尔顿问题是NP问题
- 一般只讨论特殊图中的哈密尔顿回路

Dirac定理(充分条件)

- 设一个无向图中有N个顶点,若所有顶点的度数大于等于[N/2],则哈密顿回路一定存在

*哈密尔顿回路

基本的必要条件

- 设图G=<V, E>是哈密顿图,则对于v的任意一个非空子集S
- 若以|S|表示S中元素的数目，G-S表示G中删除了S中的点以及这些点所关联的边后得到的子图
- 则W(G-S)<=|S|成立，其中W(G-S)是G-S中联通分支数

*哈密尔顿回路

基本的必要条件

- 设图G=<V, E>是哈密顿图,则对于v的任意一个非空子集S
- 若以|S|表示S中元素的数目，G-S表示G中删除了S中的点以及这些点所关联的边后得到的子图
- 则W(G-S)<=|S|成立，其中W(G-S)是G-S中联通分支数

竞赛图

- N(N>=2)阶竞赛图一定存在哈密尔顿通路
- 竞赛图即能够给每个边定向的唯一确定拓扑顺序的无向图

*哈密尔顿回路

* 求哈密尔顿回路
  - 利用竞赛图可以构造哈密尔顿通路和哈密尔顿回路
  - 这个问题较为复杂，有兴趣的同学可以课外自行研究