FOI2023 图论算法-2

张志心

Zhejiang University

2023 年 7 月 13 日

写在前面

在昨天的课程中，你们已经学会了：

- 图的定义和表示方法；
- 图的深度优先搜索方法；
- 树的定义；
- 树的深度优先搜索，欧拉序……

写在前面

在昨天的课程中，你们已经学会了：

- 图的定义和表示方法；
- 图的深度优先搜索方法；
- 树的定义；
- 树的深度优先搜索，欧拉序……

今天的课程我们主要聚焦几个在图论中非常常见的算法，包括：欧拉回路，树上最近公共祖先，图的连通性相关算法。

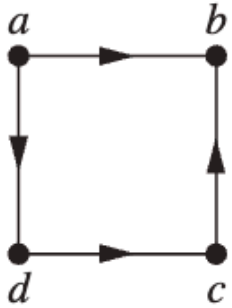
相关定义

- 欧拉回路：通过图中每条边恰好一次的回路；
- 欧拉通路：通过图中每条边恰好一次的通路；
- 欧拉图：具有欧拉回路的图；
- 半欧拉图：具有欧拉通路但不具有欧拉回路的图；

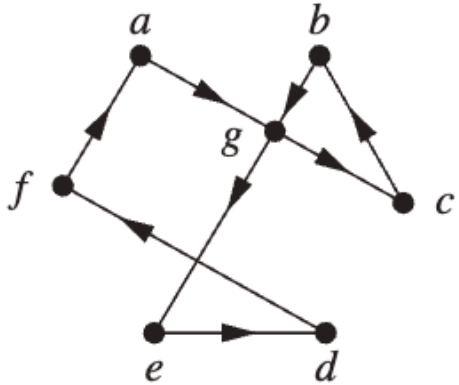
相关定义

- 欧拉回路：通过图中每条边恰好一次的回路；
- 欧拉通路：通过图中每条边恰好一次的通路；
- 欧拉图：具有欧拉回路的图；
- 半欧拉图：具有欧拉通路但不具有欧拉回路的图；

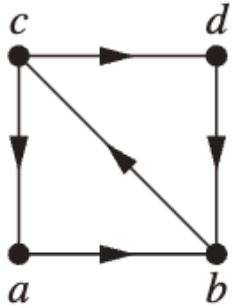
问：H_1, H_2, H_3 中哪些是欧拉图，哪些是半欧拉图？



H_1



H_2

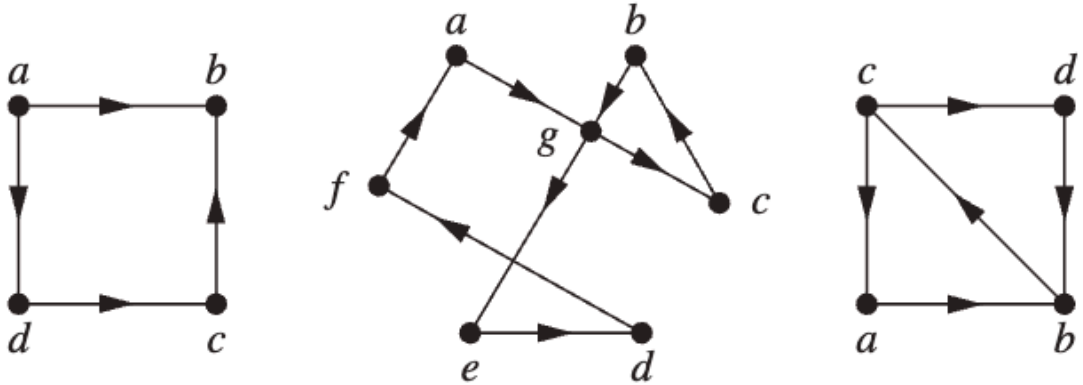


H_3

相关定义

- 欧拉回路：通过图中每条边恰好一次的回路；
- 欧拉通路：通过图中每条边恰好一次的通路；
- 欧拉图：具有欧拉回路的图；
- 半欧拉图：具有欧拉通路但不具有欧拉回路的图；

问：H_1, H_2, H_3 中哪些是欧拉图，哪些是半欧拉图？



H_2 是欧拉图，H_3 是半欧拉图，H_3 中只能找到一条欧拉路径 $a \to b \to c \to d \to b$。

性质

具有什么样性质的图是欧拉图/半欧拉图呢？

提示：

- 图的度数
  - 无向图：一个顶点的度数等于与其相连的边的数量；
  - 有向图：用“入度/出度”分别表示以该顶点为终点/起点的边的数量；

性质

具有什么样性质的图是欧拉图/半欧拉图呢？

提示：

- 图的度数
  - 无向图：一个顶点的度数等于与其相连的边的数量；
  - 有向图：用“入度/出度”分别表示以该顶点为终点/起点的边的数量；
- 考虑一条欧拉回路，显然从任何一个点开始，都一定可以走完所有的边。这是因为欧拉图上不存在“死胡同”。

性质

具有什么样性质的图是欧拉图/半欧拉图呢？

提示：

- 图的度数
  - 无向图：一个顶点的度数等于与其相连的边的数量；
  - 有向图：用“入度/出度”分别表示以该顶点为终点/起点的边的数量；
- 考虑一条欧拉回路，显然从任何一个点开始，都一定可以走完所有的边。这是因为欧拉图上不存在“死胡同”。
- 半欧拉图没有欧拉回路，但是存在欧拉路径，因此可以在图上找到一个起点和一个终点，从起点开始，每次选一条边走下去，直到到达“无路可走”的终点。

接上：欧拉图和半欧拉图的判定方法

对于一个图 G:

存在欧拉回路的充分必要条件：

1. G 连通；
2. 对于无向图，所有点点度数均为偶数；
3. 对于有向图，所有点的出度和入度相同。

接上：欧拉图和半欧拉图的判定方法

对于一个图 G:

存在欧拉回路的充分必要条件：

1. G 连通；
2. 对于无向图，所有点点度数均为偶数；
3. 对于有向图，所有点的出度和入度相同。

存在欧拉路径的充分必要条件：

1. G 连通；
2. 对于无向图，所有点点度数均为偶数（此时存在回路），或者恰有两个点度数为奇数（欧拉路径以这两个点为起止点）；
3. 对于有向图，所有点的出度和入度相同（此时存在回路），或者恰有两个点，一个点出度比入度多 1（作为起点），一个点出度比入度少 1（作为终点）。

欧拉回路的求法

- 现在你有一张图 G，你已经判定过了它确实一张欧拉图，现在考虑如何求出它的一条欧拉回路。

欧拉回路的求法

- 现在你有一张图 G，你已经判定过了它确实一张欧拉图，现在考虑如何求出它的一条欧拉回路。
- 在之前我们已经知道了，欧拉回路是不唯一的，而且可以从任意一点开始。不妨令 1 号点为起点。

欧拉回路的求法

- 现在你有一张图 G，你已经判定过了它确实一张欧拉图，现在考虑如何求出它的一条欧拉回路。
- 在之前我们已经知道了，欧拉回路是不唯一的，而且可以从任意一点开始。不妨令 1 号点为起点。
- 求欧拉回路的过程是一个不断并入小回路的过程——使用深度优先搜索的方法遍历图上的边，同时记录下每次访问的边，保证每条边都只能走一次，直到走到一个位置发现不能走了，这个时候，我们走完了第一条回路（从 1 开始到 1 结束）。

欧拉回路的求法

- 现在你有一张图 G，你已经判定过了它确实一张欧拉图，现在考虑如何求出它的一条欧拉回路。
- 在之前我们已经知道了，欧拉回路是不唯一的，而且可以从任意一点开始。不妨令 1 号点为起点。
- 求欧拉回路的过程是一个不断并入小回路的过程——使用深度优先搜索的方法遍历图上的边，同时记录下每次访问的边，保证每条边都只能走一次，直到走到一个位置发现不能走了，这个时候，我们走完了第一条回路（从 1 开始到 1 结束）。
- 接下来开始回溯，此时会选择原先回路上的一个还有边可以走的点，从它开始搜索出一条新的回路，把原先大回路上的这个点替换成这个小回路并入其中。按照这个方法进行递归，直到所有边都并入回路中。

欧拉回路的求法-Undirected

无向图版本：

```
int de[MN],st[MN],top,mark[MN<<1];
void dfs(int u){
    for(int &i=hr[u];i;i=e[i].nex)if(!mark[i]) {
        mark[i]=mark[i^1]=1;int j=i;dfs(e[i].to);//反边
        st[++top]=j&1?((j-1)>>1):j>>1;
    }
}
```

欧拉回路的求法-Directed

有向图版本：

```
int ind[MN],outd[MN],st[MN],top,mark[MN];
void dfs(int u) {
    for(int &i=hr[u];i;i=e[i].nex)if(!mark[i]) {
        mark[i]=1;int j=i;
        dfs(e[i].to);st[++top]=j-1;
    }
}
```

模板题：Luogu P1341 无序字母对

给定 n 个各不相同的无序字母对（区分大小写，无序即字母对中的两个字母可以位置颠倒）。请构造一个有 $n+1$ 个字母的字符串使得每个字母对都在这个字符串中出现。如果找不到则输出 No Solution。

样例输入：

```
4
aZ
tZ
Xt
aX
```

样例输出：XaZtX

模板题：Luogu P1341 无序字母对

- 把每个字母看成一个点，对所有字母对的两个字母进行连边，题目就是要求我们求得一条无向图的欧拉路径。

模板题：Luogu P1341 无序字母对

- 把每个字母看成一个点，对所有字母对的两个字母进行连边，题目就是要求我们求得一条无向图的欧拉路径。
- 什么，刚刚只说了欧拉回路怎么求，没有说欧拉路径怎么求呀？

模板题：Luogu P1341 无序字母对

- 把每个字母看成一个点，对所有字母对的两个字母进行连边，题目就是要求我们求得一条无向图的欧拉路径。
- 什么，刚刚只说了欧拉回路怎么求，没有说欧拉路径怎么求呀？
- 根据之前的判定方法，欧拉路径的起止点就是唯二的两个度数为奇数的点，如果所有点的度数都是偶数，那么任意选择一个点同时作为起止点即可。

模板题：Luogu P1341 无序字母对

- 把每个字母看成一个点，对所有字母对的两个字母进行连边，题目就是要求我们求得一条无向图的欧拉路径。
- 什么，刚刚只说了欧拉回路怎么求，没有说欧拉路径怎么求呀？
- 根据之前的判定方法，欧拉路径的起止点就是唯二的两个度数为奇数的点，如果所有点的度数都是偶数，那么任意选择一个点同时作为起止点即可。
- 起止点之间连边，求新图的欧拉回路。
  我们要求的欧拉路径就是欧拉回路去掉这条边。

拓展：哈密顿回路

该算法我认为比前面介绍的欧拉回路更重要，说不定会出现在下午的题目中（划掉）。

相关定义：

1. 通过图中所有顶点一次且仅一次的通路称为哈密顿通路。
2. 通过图中所有顶点一次且仅一次的回路称为哈密顿回路。
3. 具有哈密顿回路的图称为哈密顿图。
4. 具有哈密顿通路而不具有哈密顿回路的图称为半哈密顿图。

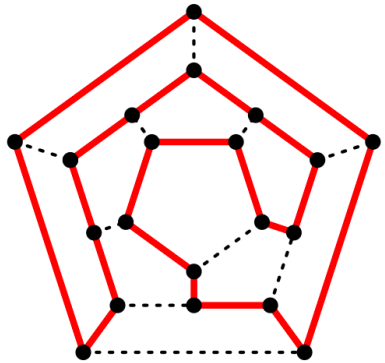


图 1: 哈密顿图

哈密顿回路的求法

- 我们已经证明求一张图的哈密顿回路/判定一张图是否是哈密顿图均为 NPC 问题。
- 目前还没有多项式时间解决求解哈密顿回路的算法。
- 一般求解哈密顿回路都是采用状态压缩 dp 算法。如何设计状态？

状态压缩-dp

- 考虑一种很暴力的方式，不妨设哈密顿回路从 1 号点开始（实际上可以从任何一个点开始）。
- 令 `f[i][S] = 0/1` 表示从 1 号点开始，当前处于 i 号节点，当前已经经过的集合为 S，该状态是否可能达到。用二进制压缩的方式来表示 S，即其值为 $\sum_{j\in S} 2^j$。其中点的编号从 0 开始标号。

状态压缩-dp

- 考虑一种很暴力的方式，不妨设哈密顿回路从 1 号点开始（实际上可以从任何一个点开始）。
- 令 `f[i][S] = 0/1` 表示从 1 号点开始，当前处于 i 号节点，当前已经经过的集合为 S，该状态是否可能达到。用二进制压缩的方式来表示 S，即其值为 $\sum_{j\in S} 2^j$。其中点的编号从 0 开始标号。一开始，$f[1][1] = 1$。
  如果有一条从 i 到 j 的边，且 $j \notin S$，则转移方程为：
  $f[i][S] \to f[j][S + (1 << j)]$。

状态压缩-dp

- 考虑一种很暴力的方式，不妨设哈密顿回路从 1 号点开始（实际上可以从任何一个点开始）。
- 令 `f[i][S] = 0/1` 表示从 1 号点开始，当前处于 i 号节点，当前已经经过的集合为 S，该状态是否可能达到。用二进制压缩的方式来表示 S，即其值为 $\sum_{j\in S} 2^j$。其中点的编号从 0 开始标号。一开始，$f[1][1] = 1$。
  如果有一条从 i 到 j 的边，且 $j \notin S$，则转移方程为：
  $f[i][S] \to f[j][S + (1 << j)]$。
- 如果要求哈密顿回路的值，则需要另外开一个数组记录一下每个状态都是由什么状态转移到的。
- 最终状态为 $f[1][(1 << n) - 1]$，表示从 1 走回 1，且经过了所有点。
- 复杂度为 $O(n^2 2^n)$。

哈密顿回路计数

如果要求一张图有多少条哈密顿回路呢？

哈密顿回路计数

如果要求一张图有多少条哈密顿回路呢？

把前面的 dp 数组改为求到达当前状态的方案数。
转移方法不变：$f[i][S] \to f[j][S + (1 << j)]$。
如果只是求所有从 1 开始到 1 结束的环的个数呢：$\sum_s f[1][s]$。

概念

最近公共祖先 LCA（Lowest Common Ancestor）：两个节点的公共祖先里深度最大的节点。一个集合的 LCA 为集合内所有节点的公共祖先中深度最大的节点。

一般可以用 $\mathrm{LCA}(x, y)$ 表示 x 和 y 的最近公共祖先。同样可以定义 $\mathrm{LCA}(S)$ 表示集合 S 的最近公共祖先。

LCA 的性质

1. 如果 x 是 y 的祖先，那么 $\mathrm{LCA}(x, y) = x$。

LCA 的性质

1. 如果 x 是 y 的祖先，那么 $\mathrm{LCA}(x, y) = x$。
2. 如果 $\mathrm{LCA}(x, y) \neq x$ $\mathrm{LCA}(x, y) \neq y$，那么一定 x 和 y 在 $\mathrm{LCA}(x, y)$ 的不同子树里。

LCA 的性质

1. 如果 x 是 y 的祖先，那么 $\mathrm{LCA}(x,y)=x$。
2. 如果 $\mathrm{LCA}(x,y)\neq x$ $\mathrm{LCA}(x,y)\neq y$，那么一定 x 和 y 在 $\mathrm{LCA}(x,y)$ 的不同子树里。
3. 结合律：$\mathrm{LCA}(x,y,z)=\mathrm{LCA}(x,\mathrm{LCA}(y,z))=\mathrm{LCA}(\mathrm{LCA}(x,y),z)$。

LCA 的性质

1. 如果 x 是 y 的祖先，那么 $\mathrm{LCA}(x,y)=x$。
2. 如果 $\mathrm{LCA}(x,y)\neq x$ $\mathrm{LCA}(x,y)\neq y$，那么一定 x 和 y 在 $\mathrm{LCA}(x,y)$ 的不同子树里。
3. 结合律：$\mathrm{LCA}(x,y,z)=\mathrm{LCA}(x,\mathrm{LCA}(y,z))=\mathrm{LCA}(\mathrm{LCA}(x,y),z)$。
4. 两点的最近公共祖先必定处在树上两点间的最短路上。

算法介绍

我们为什么需要求 LCA 呢？

算法介绍

我们为什么需要求 LCA 呢？

一个很常见的例子是树上距离。

$dis(x, y) = dep[x] + dep[y] - 2 \times dep[LCA(x, y)]$。

算法介绍

我们为什么需要求 LCA 呢？

一个很常见的例子是树上距离。

$dis(x, y) = dep[x] + dep[y] - 2 \times dep[LCA(x, y)]$。

很多需要处理树上路径的问题，都需要求两个点的 LCA：x 到 y 的路径可以分成两段，x 到 LCA，再从 LCA 到 y。

朴素算法

可以每次找深度比较大的那个点，让它向上跳。显然在树上，这两个点最后一定会相遇，相遇的位置就是想要求的 LCA。或者先向上调整深度较大的点，令他们深度相同，然后再共同向上跳转，最后也一定会相遇。

倍增的思想

二进制拆分的思想在算法竞赛中十分常见。

倍增的思想

二进制拆分的思想在算法竞赛中十分常见。
任意正整数都可以拆分成若干个 2 的整数次幂的和。

倍增的思想

二进制拆分的思想在算法竞赛中十分常见。
任意正整数都可以拆分成若干个 2 的整数次幂的和。
比如在求解完全背包的时候，有一种方法就是对于每个物品，分别将 1 个，2 个，4 个，……，2^k 个该物品视作一个整体，那么假设原先有 n 个物品，容量上限为 W，根据此方法将得到 $n \log W$ 个物品，对这些物品做 01 背包，与原问题是等价的，因为任意多个物品都可以用上述方法组合得到。

倍增的思想

二进制拆分的思想在算法竞赛中十分常见。
任意正整数都可以拆分成若干个 2 的整数次幂的和。
比如在求解完全背包的时候，有一种方法就是对于每个物品，分别将 1 个，2 个，4 个，……，2^k 个该物品视作一个整体，那么假设原先有 n 个物品，容量上限为 W，根据此方法将得到 $n \log W$ 个物品，对这些物品做 01 背包，与原问题是等价的，因为任意多个物品都可以用上述方法组合得到。
倍增的想法就是基于这样的二进制拆分。

倍增 - 树上 k 级祖先

倍增算法，常常用于解决区间问题，在基础算法的 RMQ 问题中我们已经见过了倍增的用法。
如果我们要求树上一个点的 k 级祖先，我们可以首先预处理出一个节点的所有 2^i 级别祖先。

倍增 - 树上 k 级祖先

倍增算法，常常用于解决区间问题，在基础算法的 RMQ 问题中我们已经见过了倍增的用法。
如果我们要求树上一个点的 k 级祖先，我们可以首先预处理出一个节点的所有 2^i 级别祖先。

预处理代码，预处理复杂度 $O(n \log n)$：

```
int fa[MN][LOGMN], f[MN];
// LOGMN = log2(MN)
// f : 父亲，fa[·][i] : 树上2^i级祖先
for(int i = 1; i <= n; ++i) fa[i][0] = f[i];
int K = (int)log2(N);
for(int j = 1; j <= K; ++j)
    for(int i = 1; i <= n; ++i)
        if(fa[i][j-1]) { // 如果 2^{j-1}级祖先存在
            fa[i][j] = fa[fa[i][j-1]][j-1];
        }
```

代码：求树上 k 级祖先

预处理之后，想要求 k 级祖先，只需要对 k 进行二进制拆分，然后按照 2^i 的步子往上跳即可。

代码：求树上 k 级祖先

预处理之后，想要求 k 级祖先，只需要对 k 进行二进制拆分，然后按照 2^i 的步子往上跳即可。如何对 k 进行二进制拆分？若 k 的二进制第 i 位为 1，则 (k»i&1)==1。

可以参照如下代码：

```
// 求 x 的 k 级祖先
for(int i = K; i >= 0; --i)
if(k>>i&1){
    x = fa[x][i];
}
```

倍增 LCA

如何朴素的求解 LCA？

1. 将 x, y 中深度较大的点不断向上跳，直到深度相同；
2. 将 x, y 同时向上跳，直到重合。

倍增 LCA

如何朴素的求解 LCA？

1. 将 x, y 中深度较大的点不断向上跳，直到深度相同；
2. 将 x, y 同时向上跳，直到重合。

上述做法之所以慢，是因为需要处理不断向上跳这个过程，而倍增法显然可以加速这个过程。

因此，我们将过程改为：

1. 设 x 的深度比 y 大，不妨设深度大 d，那么 x 直接跳到它的 d 级祖先，此时 x 和 y 的深度相同；

倍增 LCA

如何朴素的求解 LCA？

1. 将 x, y 中深度较大的点不断向上跳，直到深度相同；
2. 将 x, y 同时向上跳，直到重合。

上述做法之所以慢，是因为需要处理不断向上跳这个过程，而倍增法显然可以加速这个过程。

因此，我们将过程改为：

1. 设 x 的深度比 y 大，不妨设深度大 d，那么 x 直接跳到它的 d 级祖先，此时 x 和 y 的深度相同；
2. 二分找到 x 和 y 的最近公共祖先。

算法解释

- 算法的第一步就是求一个 d 级祖先；

算法解释

- 算法的第一步就是求一个 d 级祖先；
- 如果两个点的深度相同，那么公共祖先是他们共有的一个 d' 级祖先，求最近公共祖先就是要最小化这个 d'，我们显然可以用二分的方法实现这个过程。

算法解释

- 算法的第一步就是求一个 d 级祖先；
- 如果两个点的深度相同，那么公共祖先是他们共有的一个 d' 级祖先，求最近公共祖先就是要最小化这个 d'，我们显然可以用二分的方法实现这个过程。
- 但事实上并不需要采用一般的二分方法，因为我们可以在二分的过程中，如果当前深度还不是它们的共同祖先，那么可以直接将 x, y 往上移到这个深度，二分的规模就可以随着 x, y 的上移而减少。(每次向上跳，都是当前能跳的最大步长)。

算法解释

- 算法的第一步就是求一个 d 级祖先；
- 如果两个点的深度相同，那么公共祖先是他们共有的一个 d' 级祖先，求最近公共祖先就是要最小化这个 d'，我们显然可以用二分的方法实现这个过程。
- 但事实上并不需要采用一般的二分方法，因为我们可以在二分的过程中，如果当前深度还不是它们的共同祖先，那么可以直接将 x, y 往上移到这个深度，二分的规模就可以随着 x, y 的上移而减少。(每次向上跳，都是当前能跳的最大步长)。
- 从大到小考虑 2^i 级祖先，如果 x, y 的 2^i 级祖先不相同，说明向上跳 2^i 步之后，还没有到达 $\mathsf{lca}(u, v)$，这时直接把 x, y 都向上跳 2^i 步。如果相同，则 x, y 不动即可。

算法解释

- 算法的第一步就是求一个 d 级祖先；
- 如果两个点的深度相同，那么公共祖先是他们共有的一个 d' 级祖先，求最近公共祖先就是要最小化这个 d'，我们显然可以用二分的方法实现这个过程。
- 但事实上并不需要采用一般的二分方法，因为我们可以在二分的过程中，如果当前深度还不是它们的共同祖先，那么可以直接将 x, y 往上移到这个深度，二分的规模就可以随着 x, y 的上移而减少。(每次向上跳，都是当前能跳的最大步长)。
- 从大到小考虑 2^i 级祖先，如果 x, y 的 2^i 级祖先不相同，说明向上跳 2^i 步之后，还没有到达 $\mathsf{lca}(u, v)$，这时直接把 x, y 都向上跳 2^i 步。如果相同，则 x, y 不动即可。
- 最终 x, y 会跳到 $\mathsf{lca}(x, y)$ 的两个子节点上，答案极为 x, y 当前位置的父亲节点。

算法解释

- 算法的第一步就是求一个 d 级祖先；
- 如果两个点的深度相同，那么公共祖先是他们共有的一个 d' 级祖先，求最近公共祖先就是要最小化这个 d'，我们显然可以用二分的方法实现这个过程。
- 但事实上并不需要采用一般的二分方法，因为我们可以在二分的过程中，如果当前深度还不是它们的共同祖先，那么可以直接将 x, y 往上移到这个深度，二分的规模就可以随着 x, y 的上移而减少。(每次向上跳，都是当前能跳的最大步长)。
- 从大到小考虑 2^i 级祖先，如果 x, y 的 2^i 级祖先不相同，说明向上跳 2^i 步之后，还没有到达 $\mathsf{lca}(u, v)$，这时直接把 x, y 都向上跳 2^i 步。如果相同，则 x, y 不动即可。
- 最终 x, y 会跳到 $\mathsf{lca}(x, y)$ 的两个子节点上，答案极为 x, y 当前位置的父亲节点。
- 单次查询的复杂度为 $O(\log n)$

倍增 LCA - code

下面为倍增 LCA 的代码：

```
int lca(int x, int y) {
    if(dep[x] < dep[y]) swap(x, y);
    for(int i = K; i >= 0; --i) {
        if(dep[fa[x][i]] >= dep[y])
            x = fa[x][i];
    }
    if(x == y) return x; // 这步特判不能漏！
    for(int i = K; i >= 0; --i) {
        if(fa[x][i] != fa[y][i])
            x = fa[x][i], y = fa[y][i];
    }
    return fa[x][0];
}
```

RMQ 算法引入

在之前的课上，想必大家已经了解了树的欧拉序和 RMQ 问题。

RMQ 算法引入

在之前的课上，想必大家已经了解了树的欧拉序和 RMQ 问题。
将这两个知识结合在一起，就得到了我们的第二种求 RMQ 的方法。该算法只需要 $O(n)$ 的预处理（dfs 求欧拉序），并且可以做到单次查询 $O(1)$（RMQ 问题）。

RMQ 算法引入

在之前的课上，想必大家已经了解了树的欧拉序和 RMQ 问题。
将这两个知识结合在一起，就得到了我们的第二种求 RMQ 的方法。该算法只需要 $O(n)$ 的预处理（dfs 求欧拉序），并且可以做到单次查询 $O(1)$（RMQ 问题）。
该算法虽然在复杂度上完胜倍增算法，但是在实际应用中倍增算法要更为常用。而 RMQ 求 LCA 的方法，常常只在万不得已的时候才使用。

RMQ 算法求 LCA

考虑先求出树的欧拉序，考虑在 dfs 过程中在点 x 和 y 之间访问到的点。

RMQ 算法求 LCA

考虑先求出树的欧拉序，考虑在 dfs 过程中在点 x 和 y 之间访问到的点。

$\mathrm{lca}(x, y)$ 是欧拉序上任意出现 x 和 y 的位置之间，深度最小的点。

RMQ 算法求 LCA

考虑先求出树的欧拉序，考虑在 dfs 过程中在点 x 和 y 之间访问到的点。

$\mathrm{lca}(x, y)$ 是欧拉序上任意出现 x 和 y 的位置之间，深度最小的点。

听上去很绕，可以不妨记 $pos[i]$ 表示欧拉序上第一次出现 i 的位置，那么 $\mathrm{lca}(x, y)$ 就是欧拉序上 $pos[x]$ 和 $pos[y]$ 之间出现的点中深度最小的那个。

RMQ 算法求 LCA

考虑先求出树的欧拉序，考虑在 dfs 过程中在点 x 和 y 之间访问到的点。

$\mathrm{lca}(x,y)$ 是欧拉序上任意出现 x 和 y 的位置之间，深度最小的点。

听上去很绕，可以不妨记 $pos[i]$ 表示欧拉序上第一次出现 i 的位置，那么 $\mathrm{lca}(x,y)$ 就是欧拉序上 $pos[x]$ 和 $pos[y]$ 之间出现的点中深度最小的那个。

你可以用一个 `pair<int,int>` 来表示欧拉序上一个位置 (dep, id)，这样，直接用 RMQ 求出这个 pair 数组的区间最小值，取其 second 元素即为问题答案。

RMQ 算法求 LCA

考虑先求出树的欧拉序，考虑在 dfs 过程中在点 x 和 y 之间访问到的点。

$\mathrm{lca}(x,y)$ 是欧拉序上任意出现 x 和 y 的位置之间，深度最小的点。

听上去很绕，可以不妨记 $pos[i]$ 表示欧拉序上第一次出现 i 的位置，那么 $\mathrm{lca}(x,y)$ 就是欧拉序上 $pos[x]$ 和 $pos[y]$ 之间出现的点中深度最小的那个。

你可以用一个 `pair<int,int>` 来表示欧拉序上一个位置 (dep, id)，这样，直接用 RMQ 求出这个 pair 数组的区间最小值，取其 second 元素即为问题答案。

欧拉序的长度是原数组的 2 倍。

树剖-算法介绍

(该做法仅作了解。)
在后续学习算法的路上，你们将认识一个处理树上问题的利器：树链剖分。
简单来说，它的做法是将一棵树按照剖分成 $O(\log n)$ 条链，保证每一个节点到根的路径上最多 $O(\log n)$ 条链。
具体算法介绍见：https://oi-wiki.org/graph/hld/
备注：把树转化成若干条链最大好处是，可以将各种常见的处理序列的问题，比如线段树、树状树组，用于处理树上的问题。

树链剖分求 LCA

虽然大家可能还不了解树链剖分这个算法，并且该算法也不是我们今天的主线。我们仍然可以讨论一下如何使用树链剖分来求 LCA。

树链剖分求 LCA

虽然大家可能还不了解树链剖分这个算法，并且该算法也不是我们今天的主线。我们仍然可以讨论一下如何使用树链剖分来求 LCA。

根据之前的介绍，我们可以把一个点到根的路径都剖分成 $O(\log(n))$ 条链，也就是说，一个点沿着链往上跳，最多 $O(\log(n))$ 次能跳到根。

树链剖分求 LCA

虽然大家可能还不了解树链剖分这个算法，并且该算法也不是我们今天的主线。我们仍然可以讨论一下如何使用树链剖分来求 LCA。

根据之前的介绍，我们可以把一个点到根的路径都剖分成 $O(\log(n))$ 条链，也就是说，一个点沿着链往上跳，最多 $O(\log(n))$ 次能跳到根。

树剖求 LCA 的思想很简单，将深度较深的点不断往上跳（每次跳到上一条链的链底）。直到两个点位于同一个链上，取深度较小的为答案。

Tarjan 算法

我们一般称其为 Tarjan 算法求 LCA，但是事实上，该算法只涉及到 dfs 和并查集。

Tarjan 算法

我们一般称其为 Tarjan 算法求 LCA，但是事实上，该算法只涉及到 dfs 和并查集。

考虑以下问题：给你一棵 n 个点的树，同时给你 q 组询问，每次提供 x, y，查询 x, y 的 LCA。$(n \leq 10^7)$。

Tarjan 算法

我们一般称其为 Tarjan 算法求 LCA，但是事实上，该算法只涉及到 dfs 和并查集。

考虑以下问题：给你一棵 n 个点的树，同时给你 q 组询问，每次提供 x, y，查询 x, y 的 LCA。$(n \leq 10^7)$。

此时使用倍增算法已经不再使用，RMQ 算法可以解决该问题。这里我们介绍一种专门使用于这种多询问的离线算法。

离线求 LCA

提前存好所有询问，具体的，给每个点都开一个集合，存下包含它的询问的另一个点。
如果有一个询问 x, y，则需要在 x 的集合中存 y，在 y 的集合中存 x，因为你不知道它们谁会先被访问到。

离线求 LCA

提前存好所有询问，具体的，给每个点都开一个集合，存下包含它的询问的另一个点。
如果有一个询问 x, y，则需要在 x 的集合中存 y，在 y 的集合中存 x，因为你不知道它们谁会先被访问到。
在 dfs 的过程中，开一个 vis 数组来存储每个点是否已经被访问过，在 x 的回溯阶段，将 x 的并查集的根节点设置为它的父亲：`par[x] = fa[x];`。

离线求 LCA

提前存好所有询问，具体的，给每个点都开一个集合，存下包含它的询问的另一个点。
如果有一个询问 x, y，则需要在 x 的集合中存 y，在 y 的集合中存 x，因为你不知道它们谁会先被访问到。
在 dfs 的过程中，开一个 vis 数组来存储每个点是否已经被访问过，在 x 的回溯阶段，将 x 的并查集的根节点设置为它的父亲：`par[x] = fa[x];`。
dfs 到每个 x，查询其询问集合，对于集合中的一个元素 y，如果 y 此时已经被访问过，那么 $\mathrm{LCA}(x, y) = \mathit{findpar}(y)$。即它们的 LCA 是此时 y 的并查集根节点。

离线求 LCA

提前存好所有询问，具体的，给每个点都开一个集合，存下包含它的询问的另一个点。
如果有一个询问 x, y，则需要在 x 的集合中存 y，在 y 的集合中存 x，因为你不知道它们谁会先被访问到。
在 dfs 的过程中，开一个 vis 数组来存储每个点是否已经被访问过，在 x 的回溯阶段，将 x 的并查集的根节点设置为它的父亲：`par[x] = fa[x];`。
dfs 到每个 x，查询其询问集合，对于集合中的一个元素 y，如果 y 此时已经被访问过，那么 $\mathrm{LCA}(x, y) = \mathit{findpar}(y)$。即它们的 LCA 是此时 y 的并查集根节点。
总复杂度：$O(2 * m + n)$。

基本用法

多次询问树上两点最短距离：
考虑一遍 dfs 求出 $d[i]$ 为根节点到 i 的距离，则
$dis(u,v) = d[u] + d[v] - 2 \times d[lca(u,v)]$。

[JLOI2014] 松鼠的新家

原始题意：
松鼠的新家是一棵树，有 n 个房间，并且有 $n-1$ 根树枝连接，每个房间都可以相互到达，松鼠想邀请小猫前来参观，并且还指定一份参观指南，他希望小猫能够按照他的指南顺序，先去 a_1，再去 a_2，……，最后到 a_n，去参观新家。每走到一个房间，他就可以从房间拿一块糖果吃。为了保证小猫有糖果吃，他需要在每一个房间各放至少多少个糖果。
转化题意：
给出序列 $a_1, \cdots, a_n$，依次给 a_1 到 a_2，a_2 到 a_3，……，a_{n-1} 到 a_n 的路径上的所有点权加 1，求总点权和。

[JLOI2014] 松鼠的新家 - Solution

树上差分。
我们要求的是：$\sum_{i\le n} sum(i)$。
如果我们可以构造一个数组 f，满足 $sum(i)=\sum_{j在i的子树内} f(j)$。
那么我们可以递归的计算出 f 的子树和 sum(i)，然后求和得到答案。
对于一条需要加一的链 $<u,v>$，修改方式为
`f[u]+=1,f[v]+=1,f[lca(u,v)]-=1,f[fa[lca(u,v)]]-=1;`。

次小生成树

给定一张 N 个点 M 条边的无向图，求无向图的严格次小生成树。
设最小生成树的边权之和为 sum，严格次小生成树就是指边权之和大于 sum 的生成树中最小的一个。
$1 \leq N \leq 10^5, 1 \leq M \leq 3 \times 10^5$.

次小生成树 - Solution

结论：次小生成树一定可以由最小生成树通过换一条边得到。
我们可以依次考虑是否可以将一条当前不在最小生成树上的边 $< x, y >$ 换进去。
换下的边一定在 x 和 y 的路径上，要么换掉当前路径上最大的边，要么换掉严格次大的边。
用倍增预处理从 x 到 x 的 2^k 级祖先路径上最大的边和严格次大的边。

强连通和弱连通

- 强连通：如果对于图中两点 u, v，存在一条 u 到 v 的路径且也存在一条 v 到 u 的路径，那么称这两点是强连通的。

强连通和弱连通

- 强连通：如果对于图中两点 u, v，存在一条 u 到 v 的路径且也存在一条 v 到 u 的路径，那么称这两点是强连通的。
- 强连通图：如果图 G 中任意两点都是强连通的，那么 G 被称作强连通图。

强连通和弱连通

- 强连通：如果对于图中两点 u, v，存在一条 u 到 v 的路径且也存在一条 v 到 u 的路径，那么称这两点是强连通的。
- 强连通图：如果图 G 中任意两点都是强连通的，那么 G 被称作强连通图。
- 强连通分量：对图 G 来说，若 G' 为其子图且 G' 为强连通图，则 G' 被称作 G 的强连通分量。

强连通和弱连通

- 强连通：如果对于图中两点 u, v，存在一条 u 到 v 的路径且也存在一条 v 到 u 的路径，那么称这两点是强连通的。
- 强连通图：如果图 G 中任意两点都是强连通的，那么 G 被称作强连通图。
- 强连通分量：对图 G 来说，若 G' 为其子图且 G' 为强连通图，则 G' 被称作 G 的强连通分量。
- 极大强连通分量：G' 为 G 的强连通分量，且 G' 不是任何 G 的其他强连通分量的子图，那么 G' 为 G 的极大强连通分量。

强连通和弱连通

- 强连通：如果对于图中两点 u, v，存在一条 u 到 v 的路径且也存在一条 v 到 u 的路径，那么称这两点是强连通的。
- 强连通图：如果图 G 中任意两点都是强连通的，那么 G 被称作强连通图。
- 强连通分量：对图 G 来说，若 G' 为其子图且 G' 为强连通图，则 G' 被称作 G 的强连通分量。
- 极大强连通分量：G' 为 G 的强连通分量，且 G' 不是任何 G 的其他强连通分量的子图，那么 G' 为 G 的极大强连通分量。

一般默认“强连通分量”指的就是极大强连通分量。

对于一个无向图来说，强连通和连通等价，所以不单独研究强连通分量。

Tarjan 算法求强连通分量

Tarjan 算法的主体是一个 dfs，参考树的 dfs，我们试图为一张图也建立一个 dfs 树。

Tarjan 算法中涉及的一些概念：

- dfs 树：以 dfs 的形式遍历图中的每一个点一次，dfs 经过的边和经过的点构成的树 (图不连通时，dfs 树会构成一个森林)。dfs 树的树根定义为第一个 dfs 的节点。
- dfs 序：对图进行 dfs 时经过每个点的顺序构成的序列。
- 树边：dfs 树上的边。
- 返祖边：dfs 树外，由 dfs 树中的子节点指向祖先节点的边。
- 横插边：dfs 树外，两端点在 dfs 树中没有祖孙关系的边。

返祖边和横插边都属于非树边。

Tajan 算法的性质

在无向图中，dfs 树的非树边只会出现返祖边，不会出现横插边。

Tajan 算法的性质

在无向图中，dfs 树的非树边只会出现返祖边，不会出现横插边。
在有向图中，dfs 树的非树边可能出现返祖边和横插边，但横插边只会从 dfs 序大的点指向 dfs 序小的点。

Tajan 算法的性质

在无向图中，dfs 树的非树边只会出现返祖边，不会出现横插边。
在有向图中，dfs 树的非树边可能出现返祖边和横插边，但横插边只会从 dfs 序大的点指向 dfs 序小的点。

- $dfn(u)$：u 的 dfs 序；
- $low(u)$：从 u 出发，不经过指向搜索到它时已经确定了所属强连通分量的点的横插边能到达的最小 dfn；
- $bel(u)$：u 所属的强连通分量的编号。

画图演示

尝试对如下左图进行 dfs，并且判断返祖边和横插边：

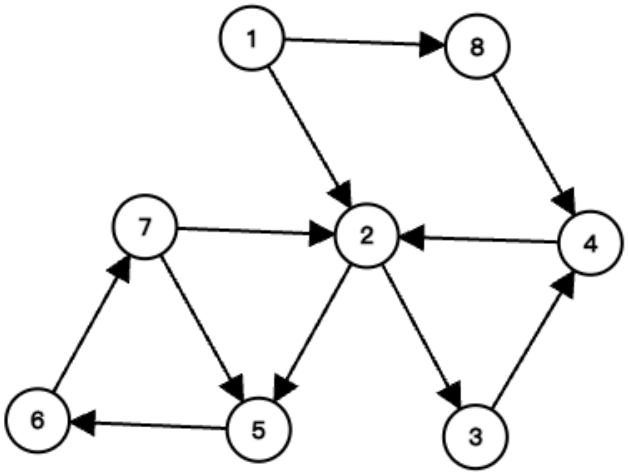


图 2: 原图

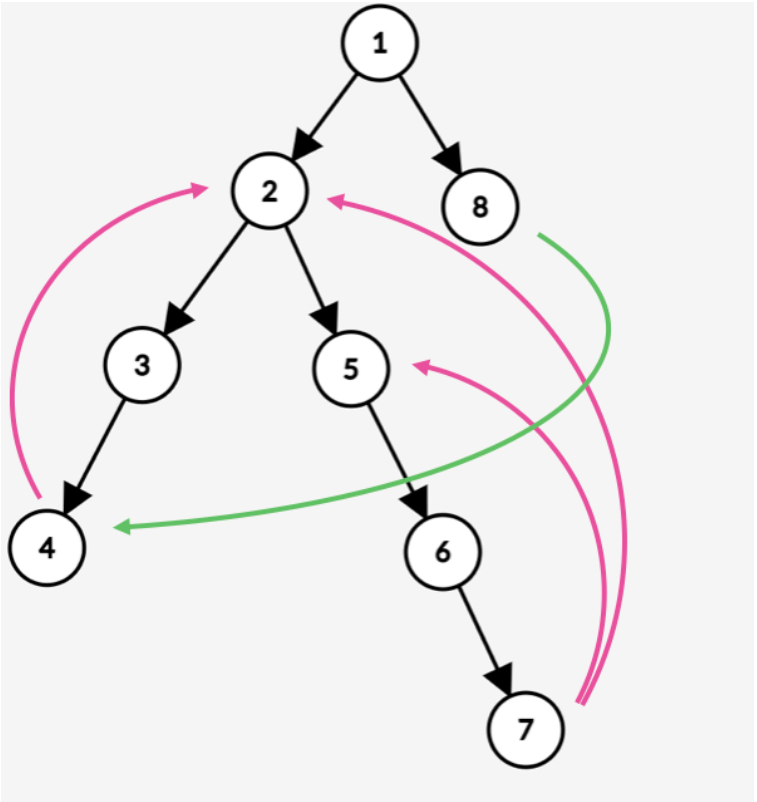


图 3: dfs 图

上图中，一共有 3 个强连通分量：$\{2,3,4,5,6,7\},\{1\},\{8\}$。

强连通的求法

对于一条横插边 $u \to v$ 来说 (u 为当前节点)，由于 u 不在 v 的子树中，所以从 v 出发不一定能到达 u，因此 u, v 不一定属于同一强连通分量，所以这条横插边没有实际意义。

强连通的求法

对于一条横插边 $u \to v$ 来说 (u 为当前节点)，由于 u 不在 v 的子树中，所以从 v 出发不一定能到达 u，因此 u, v 不一定属于同一强连通分量，所以这条横插边没有实际意义。

而对于返祖边 $u \to v$ 来说，这条边使得 u 的所有深度大于等于 v 的祖先都和 u 能互相到达，所以是有实际意义的。

强连通的求法

对于一条横插边 $u \to v$ 来说 (u 为当前节点)，由于 u 不在 v 的子树中，所以从 v 出发不一定能到达 u，因此 u, v 不一定属于同一强连通分量，所以这条横插边没有实际意义。

而对于返祖边 $u \to v$ 来说，这条边使得 u 的所有深度大于等于 v 的祖先都和 u 能互相到达，所以是有实际意义的。

通过刚刚的讨论，我们可以挖掘出 *low* 数组更深层的含义：

对 u 来说，只有它在 dfs 树中的某些祖先能到达它，而 *low* 数组代表的就是从这个点出发能到达的深度最小 (最靠上) 的祖先的 *dfn*。由此能够计算出图的各种连通性信息。

强连通分量的求法

为了求出各个强连通分量中包含的结点，在 dfs 的同时，我们还需要用一个栈来记录检查过的点。具体来说，dfs 到结点 u 时：

- 将 u 进栈；
- 继续 dfs 和用返祖边更新 u 的 low；
- 如果 $dfn(u) = low(u)$，那么栈中的所有元素构成一个新的强连通分量，将 u 和栈中在 u 之后的所有节点标记并出栈。

返祖边，横插边，树边在 dfs 中的判定

判断一条边 $e : u \to v$ 的类型：

1. 树边：$dfn(v)$ 未赋值，说明 v 未被检查，需要 dfs 到 v。
2. 返祖边：v 未被标记，说明 v 是 u 在 dfs 树中的祖先，e 是返祖边，需要更新 $low(u)$。
3. 横插边：v 被标记过，那么 e 是横插边，直接跳过即可。

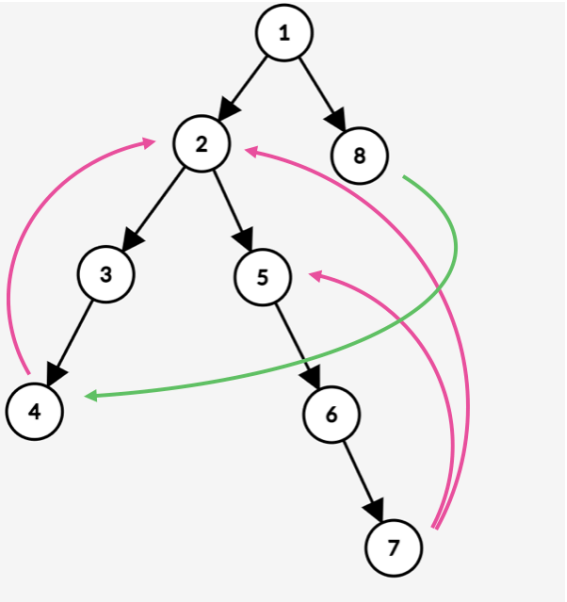


图 4: dfs 图

Tarjan 算法代码

```
int cnt = 0, dfn[MN], low[MN], st[MN], top = 0;
int scc = 0, bel[MN];
// 强连通分量个数，所属强连通分量编号。
vector<int> G[MN];
// G[x] 中保存从 x 出发的边的指向的节点编号。
void tarjan(int x) {
    dfn[x] = low[x] = ++cnt; // 更新 dfs 序
    st[++top] = u; // 点入栈
    for(auto y: G[x]) { // 所有 x 可以到达的点
        if(!dfn[y]) {
        // 如果 y 没有被访问过，那么为树边。
            tarjan(y);
            low[x] = min(low[x], low[y]);
        }
        else if(!bel[y]) {
        // y 还没有判定所属的强连通分量：
        // 说明 y 还在栈中，为反祖边。
            low[x] = min(low[x], low[y]);
        } // 否则为横插边，不考虑。
    }
    if (low[x] == dfn[x]) {
    // 说明此时 x 为一个强连通分量的顶部
        ++scc; int y;
        do {
            bel[st[top--]] = scc;
            // do somthing with node y in scc
        } while (st[top+1]!=x);
    }
}
```

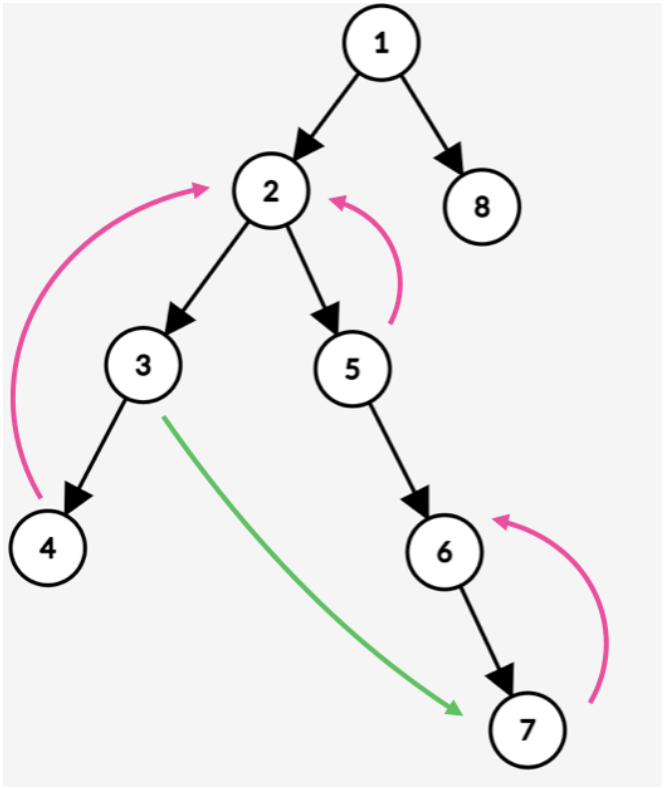


图 5: 又一张 dfs 图

强连通分量缩点

一个环是一个强连通分量。
强连通分量构成的环肯定是强连通分量。

强连通分量缩点

一个环是一个强连通分量。
强连通分量构成的环肯定是强连通分量。
因此如果将同在一个强连通分量中点缩成一个，把处于不同强连通分量中的两个点之间的边看成是两个强连通分量之间的边，那么就可以将原图变为一个有向无环图(DAG)。
很多有向无环图问题是可以通过拓扑排序用队列和动态规划解决的。
通过上述办法简化问题的过程我们称为缩点。

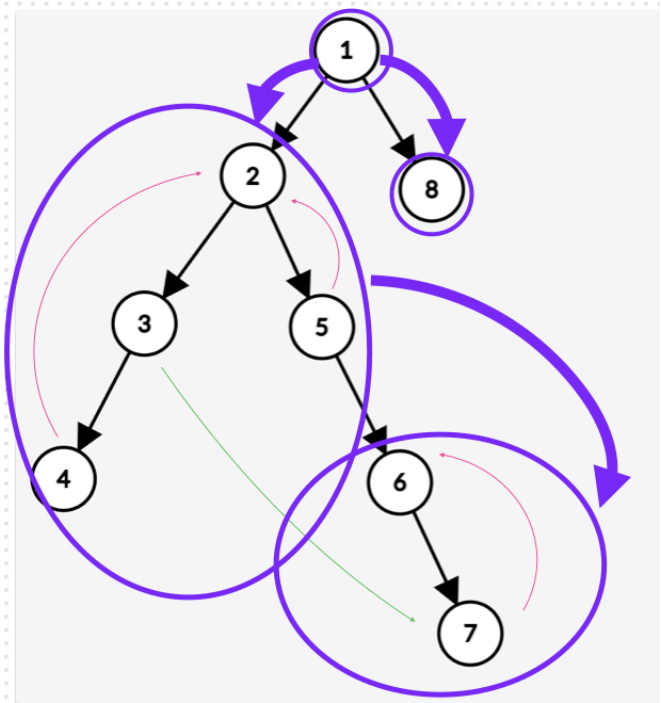


图 6: 缩点

缩点代码

遍历原图上的边，若一条边的两个点处于不同的强连通分量，那么就将两个强连通分量之间连一条边。

```
for(int x = 1; x <= n; ++x) {
    for(int y : G[x]) {
        if(bel[x] != bel[y])
            new_G[bel[x]].push_back(new_G[bel[y]]);
    }
}
```

原图为 G，新图为 new_G。

无向图的连通性说明

在无向图中，“(弱) 连通”与“强连通”是没有区别的——连通块内任意两个点都是互相可达的。
因此在无向图中，我们可以考虑更强的连通性：(点 / 边) 双连通性。

点双连通性和割点

点双连通分量：若 G 的某个子图的每两个点之间都存在至少两条不经过除起点和终点外相同的点的路径，则称该子图为点双连通分量。
特别的，一个仅含两个点一条边的子图也是点双连通分量。(尽管这样感觉很奇怪？)

点双连通性和割点

点双连通分量：若 G 的某个子图的每两个点之间都存在至少两条不经过除起点和终点外相同的点的路径，则称该子图为点双连通分量。
特别的，一个仅含两个点一条边的子图也是点双连通分量。(尽管这样感觉很奇怪？)
割点：删除该结点和该结点所连接的所有边后，图中连通块数量增加的结点称为割点。

点双连通性和割点

点双连通分量：若 G 的某个子图的每两个点之间都存在至少两条不经过除起点和终点外相同的点的路径，则称该子图为点双连通分量。
特别的，一个仅含两个点一条边的子图也是点双连通分量。(尽管这样感觉很奇怪？)
割点：删除该结点和该结点所连接的所有边后，图中连通块数量增加的结点称为割点。
对无向图来说，点双连通分量就是不存在割点的连通子图。
我们通常默认考虑的是极大点双连通分量。

点双连通性和割点

点双连通分量：若 G 的某个子图的每两个点之间都存在至少两条不经过除起点和终点外相同的点的路径，则称该子图为点双连通分量。
特别的，一个仅含两个点一条边的子图也是点双连通分量。(尽管这样感觉很奇怪？)
割点：删除该结点和该结点所连接的所有边后，图中连通块数量增加的结点称为割点。
对无向图来说，点双连通分量就是不存在割点的连通子图。
我们通常默认考虑的是极大点双连通分量。
割点属于多个点双连通分量；其他点唯一地属于某个点双连通分量；每条边唯一地属于某个点双连通分量。

Tarjan 算法求割点

由于我们现在是在无向图中考虑“双连通”性，所以对 low 的定义需要做一些调整：

$low(u)$：从 u 出发，经过至多一条返祖边能到达的最小 dfn。

Tarjan 算法求割点

由于我们现在是在无向图中考虑“双连通”性，所以对 low 的定义需要做一些调整：

$low(u)$：从 u 出发，经过至多一条返祖边能到达的最小 dfn。

设我们要判断 u 是不是割点，在 dfs 树中：

1. 若 u 为根，则 u 有两个及以上儿子等价于 u 为割点。

Tarjan 算法求割点

由于我们现在是在无向图中考虑“双连通”性，所以对 low 的定义需要做一些调整：

$low(u)$：从 u 出发，经过至多一条返祖边能到达的最小 dfn。

设我们要判断 u 是不是割点，在 dfs 树中：

1. 若 u 为根，则 u 有两个及以上儿子等价于 u 为割点。2. 若 u 不是根，则如果该结点 u 在 dfs 树中的任意一个子结点 v 满足 $low(v) \geq dfn(u)$，那么 u 就是割点。

Tarjan 算法求割点

由于我们现在是在无向图中考虑“双连通”性，所以对 low 的定义需要做一些调整：

$low(u)$：从 u 出发，经过至多一条返祖边能到达的最小 dfn。

设我们要判断 u 是不是割点，在 dfs 树中：

1. 若 u 为根，则 u 有两个及以上儿子等价于 u 为割点。2. 若 u 不是根，则如果该结点 u 在 dfs 树中的任意一个子结点 v 满足 $low(v) \geq dfn(u)$，那么 u 就是割点。

更进一步的，我们可以直接求出：

$cut(u)$：删去 u 及其相邻的边后，图中连通块数量的增量。

Tarjan 算法求割点代码

```cpp
void tarjan(int u, int pre = 0) {
// pre 记录 u 的父亲，规定根的父亲为 0
    dfn[u] = low[u] = ++tot;
    for (int v: e[u]) {
        if (v == pre) continue; // 跳过父亲
        if (!dfn[v]) {
            tarjan(v, u), low[u] = min(low[u], low[v]);
            if (low[v] >= dfn[u]) ++cut[u];
        } else low[u] = min(low[u], dfn[v]); // 返祖边
    }
    if (!pre) --cut[u]; // 根结点需额外 - 1
}
```

Tarjan 算法求点双连通分量

点双连通分量的求法和强连通分量类似，同样用栈来维护当前的分量。

注意：割点 u 在当前分量中，但是弹栈时不要弹出 u，因为 u 还在其他点双连通分量中！

```
1  void tarjan(int u, int pre = 0) {
2  // pre 记录 u 的父亲，规定根的父亲为 0
3      dfn[u] = low[u] = ++tot, stk[++top] = u;
4      for (int v: e[u]) {
5          if (v == pre) continue; // 跳过父亲
6          if (!dfn[v]) {
7              tarjan(v, u), low[u] = min(low[u], low[v]);
8              if (low[v] >= dfn[u]) {
9                  ++cut[u], ++bcc;
10                 com[bcc].push_back(u); // 将 u 加入点双连通分量
11                 int w;
12                 do { w = stk[top--], com[bcc].push_back(w); }
13                 while (v != w);
14                 // 只 pop 到 v 为止，v 与 u 之间可能还有其他点
15             }
16         } else low[u] = min(low[u], dfn[v]); // 返祖边
17     }
18     if (!pre) --cut[u]; // 根结点需额外 - 1
19 }
```

边双连通性与割边

判断 $u \to v$ 是否为割边，只需比较 $low(v)$ 和 $dfn(u)$ 的大小：
若 $low(v) > dfn(u)$ 则说明结点 v 的子树内的所有结点无法通过返祖边到达结点 u 或 u 的祖先，因此这条边为割边。

边双连通性与割边

判断 $u \to v$ 是否为割边，只需比较 $low(v)$ 和 $dfn(u)$ 的大小：
若 $low(v) > dfn(u)$ 则说明结点 v 的子树内的所有结点无法通过返祖边到达结点 u 或 u 的祖先，因此这条边为割边。
求边双联通分量时，同样使用一个栈记录访问过的结点。当 $dfn(u) = low(u)$ 时，说明 u 的子树中没有能到达 u 的祖先的返祖边，因此此时栈中的结点构成一个新的边双联通分量。(也可以在此时判定 u 连向父节点的边为桥).

边双连通性与割边

判断 $u \to v$ 是否为割边，只需比较 $low(v)$ 和 $dfn(u)$ 的大小：
若 $low(v) > dfn(u)$ 则说明结点 v 的子树内的所有结点无法通过返祖边到达结点 u 或 u 的祖先，因此这条边为割边。
求边双联通分量时，同样使用一个栈记录访问过的结点。当 $dfn(u) = low(u)$ 时，说明 u 的子树中没有能到达 u 的祖先的返祖边，因此此时栈中的结点构成一个新的边双联通分量。(也可以在此时判定 u 连向父节点的边为桥).
在边双连通的问题中，重边是不能忽略的，因此在 dfs 时不能仅传父节点而要传父节点连过来的边。

Tarjan 算法求割边和边双连通分量

```cpp
using edge = pair<int, int>;
vector<edge> e[N];
int cnt = 0;
void add_edge(int u, int v) {
    ++cnt;
    e[u].push_back(make_pair(v, cnt));
    e[v].push_back(make_pair(u, cnt));
}
int tot = 0, dfn[N], low[N];
int bcc = 0, bel[N];
int top = 0, stk[N];
bool bridge[N];
void tarjan(int u, int pre = 0) { // pre 记录走到 u 的边的编号
    dfn[u] = low[u] = ++tot, stk[++top] = u;
    for (auto ee : e[u]) {
        int v = ee.first;
        if (ee.second == pre) continue; // 跳过刚走过的边
        if (!dfn[v]) {
            tarjan(v, ee.second), low[u] = min(low[u], low[v]);
            if (dfn[u] < low[v]) bridge[ee.second] = true;
        }
        else
            low[u] = min(low[u], dfn[v]); // 返祖边
    }
    if (dfn[u] == low[u]) {
        ++bcc, bridge[pre] = true;
        int v;
        do v = stk[top--], bel[v] = bcc;
        while (v != u);
    }
}
```

总结

1. 强连通分量：是指满足任意两个结点都能互相到达的有向图的子图。
2. 割点：是指无向图中删除会使连通块数量增加的结点。
3. 点双连通分量：是指无向图中没有割点的子图。
4. 割边：是指无向图中删除会使连通块数量增加的边。
5. 边双连通分量：是指无向图中没有割边的子图。

这些算法 Tarjan 算法均可以在 $O(n)$ 时间内求解。

[USACO03FALL / HAOI2006] 受欢迎的牛

给一张有向图，问有多少个点可以被其他所有的点到达。
要求 $O(n)$ 做法。

[USACO03FALL / HAOI2006] 受欢迎的牛 - Solution

关键词：有向图，可达（连通性）。
考虑 Tarjan 求图的强连通分量，然后根据强连通分量缩点成 DAG。
在缩点得到的 DAG 上，一个点可以被其他所有的点到达需要满足如下几个条件：

1. DAG 连通；
2. 出度为 0；
3. 没有其他出度为 0 的点。

如果存在这样的点，那么它所代表的强连通分量内的所有点的个数即为答案。
否则，答案为 0。

[APIO2009] 抢掠计划

给定一个 n 个点 m 条边有向图，每个点有一个权值。
给定 p 个终止结点，求一条从 1 出发到任一终止结点的路径，使路径经过的点权值之和最大。点权全部为非负整数。允许多次经过一条边或者一个点，但是，重复经过的点，权值只计算一次。你需要求出这个权值和。
要求 $O(n)$ 算法。

[APIO2009] 抢掠计划 - Solution

关键词：有向图，最大点权和。
首先由于点权都是非负的，所以对于一个强连通分量，如果能到达其中的一个点，那么一定会将整个强连通分量的点都走一遍。
按照强连通分量缩点成 DAG。
对于缩点后的新图，定义 $dp[x]$ 表示从 x 号强连通分量开始走，能获得的最大点权和。
$dp[x] = \max_{\text{存在一条从}x\text{到}y\text{的边}} \{dp[y]\} + sum[x]$。其中 $sum[x]$ 表示强连通分量 x 的点权和。
考虑如何计算上述转移方程，要保证在计算 $dp[x]$ 的时候，所有 $dp[y]$ 都已经计算完成。
对新图进行拓扑排序，然后按照其逆序进行转移即可。
对原图进行强连通分量缩点，然后在 DAG 上拓扑排序 + dp 是很常见的做法。

[USACO15JAN] 草鉴定

约翰有 n 块草场，编号 1 到 n，这些草场由若干条单行道相连。奶牛贝西是美味牧草的鉴赏家，她想到达尽可能多的草场去品尝牧草。
贝西总是从 1 号草场出发，最后回到 1 号草场。她想经过尽可能多的草场，贝西在通一个草场只吃一次草，所以一个草场可以经过多次。
因为草场是单行道连接，这给贝西的品鉴工作带来了很大的不便，贝西想偷偷逆向行走一次，但最多只能有一次逆行。
求贝西最多能吃到多少个草场的牧草。
要求 $O(n)$ 算法。

[USACO15JAN] 草鉴定 - Solution

惯例地先缩点，缩出的点权为分量的大小。
在 DAG 上 dp，求出 f_i 和 g_i，分别表示 1 到 i 的最大点权路和 i 到 1 的最大点权路。
DAG 的性质保证了从 1 出发的路径和到 1 的路径不会有重复的结点。
对于每条边 $u \to v$，我们用 $f_v + g_u$ 更新答案即可。

UOJ#67. 新年的毒瘤

辞旧迎新之际，喜羊羊正在打理羊村的绿化带，然后他发现了一棵长着毒瘤的树。
这个长着毒瘤的树可以用 n 个结点 m 条无向边的无向图表示。这个图中有一些结点被称作是毒瘤结点，即删掉这个结点和与之相邻的边之后，这个图会变为一棵树。树也即无简单环的无向连通图。
现在给你这个无向图，喜羊羊请你帮他求出所有毒瘤结点。
$n, m \leq 10^5$。

UOJ#67. 新年的毒瘤 - Solution

删掉这个点后，原图是一棵树（等价于一个 $n-1$ 个点 $n-2$ 条边的连通图）。
根据树的等价定义：一个点是毒瘤结点，当且仅当这个点不是割点，且与这个点相连的边的数目 $= m-(n-2)$。

SPOJ STC10 - Blockade

给一张无向连通图，分别求去掉每个点连接的所有边后不连通的点对数量。

$n \leq 100000, m \leq 500000$。

SPOJ STC10 - Blockade - Solution

若删除的不是割点，则不能访问的有序点对数量为 $2n-2$。
若删除的是割点，那么我们有三种不能访问的点对：

1. 割点的子树不能访问子树外结点；
2. 割点与其他结点间不能互相访问；
3. 整棵 dfs 树中割点和子树外的部分不能访问子树内结点。

依次枚举 u 的若干儿子时，我们记录目前枚举过的儿子的子树大小之和 sum。
枚举到子树大小为 siz 的儿子时，对答案的贡献为 $(n-siz)\times siz$，然后将 $sum \leftarrow sum+siz$。
最后 $ans \leftarrow ans+(n-sum-1)\times(sum+1)+n-1$。

[HNOI2012] 矿场搭建

给一张无向图，选尽可能少的点集 S，使得图中任意一个点被删除后其他点仍与 S 中至少一个点连通。求 S 中的最小点数以及最小的 S 的方案数。
要求 $O(n)$。

[HNOI2012] 矿场搭建 - Solution

如果整张图点双连通，则答案为 2（至少是 2，因为如果只有 1 个点那么这个点本身被删除就没了），方案数就是 $\binom{n}{2} = \frac{n(n-1)}{2}$。

[HNOI2012] 矿场搭建 - Solution

如果整张图点双连通，则答案为 2（至少是 2，因为如果只有 1 个点那么这个点本身被删除就没了），方案数就是 $\binom{n}{2} = \frac{n(n-1)}{2}$。

然后考虑每个点双连通分量。如果某个点双连通分量中有两个以上的割点，则这个分量中不需要选点；

[HNOI2012] 矿场搭建 - Solution

如果整张图点双连通，则答案为 2（至少是 2，因为如果只有 1 个点那么这个点本身被删除就没了），方案数就是 $\binom{n}{2} = \frac{n(n-1)}{2}$。

然后考虑每个点双连通分量。如果某个点双连通分量中有两个以上的割点，则这个分量中不需要选点；

否则，这个分量中要选择非割点的一个点。

[HNOI2012] 矿场搭建 - Solution

如果整张图点双连通，则答案为 2（至少是 2，因为如果只有 1 个点那么这个点本身被删除就没了），方案数就是 $\binom{n}{2} = \frac{n(n-1)}{2}$。

然后考虑每个点双连通分量。如果某个点双连通分量中有两个以上的割点，则这个分量中不需要选点；

否则，这个分量中要选择非割点的一个点。

根据乘法原理，总方案数就是所有只有一个割点的点双连通分量的大小减 1 的乘积。

本题不仅要找到割点，还要找到所有点双连通分量。

[USACO06JAN] Redundant Paths G

给一张无向图，连最少的边使其边双连通。

[USACO06JAN] Redundant Paths G - Solution

把边双缩点后原图变成一棵树，然后把所有叶子（度数为 1）连在一起即可。

答案就是 $\left\lceil \frac{\text{叶子数}}{2} \right\rceil$。

Thanks for Listening!

Q&A ： Any Quetions ?

Thanks for Listening!

Q&A ： Any Quetions ?

关于下午的题：祝大家好运。