图论初步

天津大学 ResurrectionTX

2023 年 7 月 5 日

目录

目录

图论简介

Description

图论 (Graph theory) 是数学的一个分支，图是图论的主要研究对象。
图 (Graph) 是由若干给定的顶点及连接两顶点的边所构成的图形，这种图形通常用来描述某些事物之间的某种特定关系。
顶点用于代表事物，连接两顶点的边则用于表示两个事物间具有这种关系。

目录

结构体存储边的信息

Description

结构体数组存储每条边的两个端点 u，v 和边权 w。
例子：克鲁斯卡尔最小生成树
在需要多次建边等情况使用

结构体存边代码实现

```
1 struct Bian{
2     int u, v, w;
3     bool operator < (const Bian &rhs) const {
4         return w < rhs.w;
5 }
6 }edge[M + 50];
7 for (int i = 1; i <= m; i++)
8     scanf("%d%d%d", &edge[i].u, &edge[i].v, &edge[i].w);
9 sort(edge + 1, edge + m + 1);
```

邻接矩阵存储边的信息

Description

大小为 n^2 的二维数组存储 n 个点之间的边。

$$Graph_{u,v,1\leq v\leq n,1\leq u\leq n} = \begin{cases} w\ There\ is\ an\ edge\ (u,v,w) \\ 0\ Otherwise \end{cases}$$

例子：Floyd 求最短路

$O(1)$ 查询某条边是否存在，常用于稠密图。

邻接矩阵存边代码实现

```
1 int graph[N + 50][N + 50];
2 for (int i = 1, u, v, w; i <= m; i++) {
3     scanf("%d%d%d", &u, &v, &w);
4     graph[u][v] = graph[v][u] = w;//无向图
5     graph[u][v] = w;//有向图
6 }
```

邻接表存储边的信息

Description

使用一个支持动态增加元素的数据结构构成的数组存储和结点 u 有关的所有边的信息（另一端点，边权）
应用广泛，vector 的好处是可以 sort 然后二分，尤其适用于对边进行排序的情况。

邻接表存边代码实现

```
1  struct Node{
2      int v, dis;
3  };
4  vector<Node> graph[N + 50];
5  for (int i = 1, u, v, w; i <= m; i++) {
6      scanf("%d%d%d", &u, &v, &w);
7      graph[u].push_back((Node){v, w});
8      graph[v].push_back((Node){u, w});
9  }
10 for (vector<Node>::iterator it = graph[x].begin(); it !=
       graph[x].end(); it++) {
11     int v = it->v, w = it->w;
12     //Do Something
13 }
```

链式前向星存储边的信息

Description

使用链表实现的邻接表。
应用广泛，无向图的两条边的编号之间可以存在对应关系，令最小的边编号是 0，那么 $num_{u,v} = num_{v,u} \bigoplus 1$，常用于网络流。

链式前向星存边代码实现

```
1  struct Node {
2      int nxt, v, w;
3  } edge[M * 2 + 50];
4  void Addedge(int u, int v, int w) {
5      edge[++num] = (Node){head[u], v, w};
6      return;
7  }
8  for (int i = head[u]; i; i = edge[i].nxt) {
9      int v = edge[i].v;
10     //Do Something
11 }
```

目录

DFS

Description

应该没有不会写 DFS 的吧？
DFS 中每个点第一次被遍历的顺序称为 DFS 序，常记作 dfn。
根据 DFS 的原理，不难发现在一颗树上，若一个结点的编号为 x，那么它的子树的所有结点就会在 $[dfn_x, dfn_x + siz_x - 1]$ 这段区间上。
这样借助 dfs 遍历往往能将树上问题转化为序列问题，使用线段树，分块等维护序列信息的数据结构进行维护。
DFS 还有个东西叫括号序列，即一个结点第一次被压进栈时记录 (，第一次被弹出栈时记录)，显然一个结点的 (和) 之间就是这个结点的子树里的所有结点。

BFS

Description

应该没有不会写 BFS 的吧？
注意一下 BFS 为什么能实现第一次遍历结点就可以求出到结点的最短路的本质：BFS 的过程里实际上可以被分为 Open 和 Closed 两个容器，没有遍历过的在 Open 容器，遍历过的在 Closed 容器；而实现从两个容器之间转移的队列中的每个元素大小是呈现单调关系的。
01BFS 求最短路：边权只有 0 和 1 的图，关键是保持队列的单调性质，所以我们使用双端队列，每次 0 入队头，1 入队尾即可。
边权可以是任意呢？使用优先队列维护：Dijkstra

目录

Floyd

Description

邻接矩阵存图的情况下使用 floyd，本质上是 dp。
$f_{i,j,k}$ 表示只走编号 1-k 的结点的情况下，结点 i 和结点 j 之间的最短路。
$f_{k,i,j} = min(f_{k-1,i,j}, f_{k-1,i,k} + f_{k-1,k,j})$
第一个维度可以滚掉。
稍微修改可以解决传递闭包问题，即两个点之间是否连通。
可以使用 bitset 优化复杂度至 $O(\frac{n^3}{64})$

Bellman-Ford and SPFA

Description

松弛操作：对于边 (u, v)，$dis_v = min(dis_v, dis_u + edge_{u,v})$
注意到一点：最短路最长不会经过 n-1 条边。
每一轮操作我们对每一条边都进行松弛操作，这样最短路上的边的数量至少加一，n 轮之后一定找到了最短路，复杂度 $O(mn)$。
SPFA 是使用队列优化减少松弛次数的 Bellman-Ford。
复杂度是假的，有各种奇怪优化都可以被卡成 Bellman-Ford。小 SPFA，它已经死了
所以除非负权图，不要使用 SPFA。
负环：最短路边数不会大于 n，所以 Bellman-Ford 如果在 n 次对所有边的松弛操作之后还可以松弛操作，那就有负环；或者 SPFA 时候有个结点入队次数超过了 n。

Dijkstra

Description

本质上是两个集合 S 和 T，找到最短路的扔到 S 里面，没找到的扔到 T 里面，每一次从 T 里面拿出离 S 集合中最近的点确定最短路丢到 S 中，是个贪心，本质就是优先队列 BFS。
不能处理负权图。
优化的是找到最小的过程，一般用单调队列，复杂度是真的 $O(nlogm)$。
Dijkstra 和 SPFA 松弛的时候记录一个 Pre 就能最后递归输出最短路了。
如果要看一个点在不在两个点的最短路上就是看 $dis_{u,v}? = dis_{u,i} + dis_{i,v}$

ABC307F

Description

有一张无向带权图，在第一天有一些人生病了。
一共 d 天，每天距离生病的人的路径在 D_i 以内的人会感染，感染的人在以后的时间里一直生病。
求每个人最早是第几天被感染的。
$n, m, d <= 3e5$

ABC307F

solve

分段最短路。
建立超级源点 S 连向第一天感染的人距离设置为 0。
之后每天新感染的人距离设置为 0 重新跑 Dijkstra。
考虑 Dijkstra 的本质是划分俩集合，实际上不需要重新跑一遍，在前一天基础上继续跑就行。

目录

次短路

Description

维护最短路的同时开一个数组 *diss* 记录次短路即可。
每次更新的时候如果最短路能被松弛，就把最短路先传给次短路，再松弛最短路。
如果只有次短路能松弛，那就只松弛次短路。

目录

分层图最短路

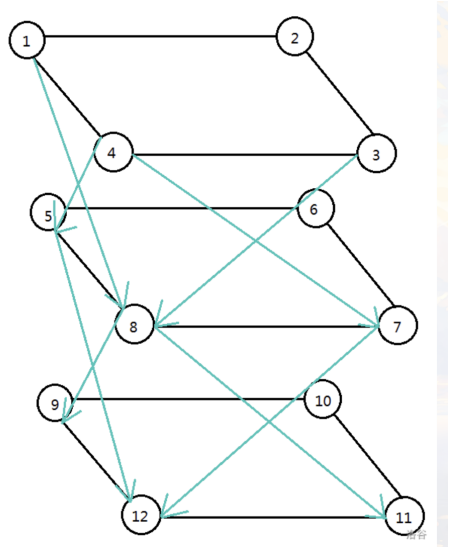
Description

分层图最短路，如：有 k 次零代价通过一条路径，求总的最小花费。
对于这种题目，我们可以采用 DP 相关的思想，设 $dis_{i,j}$ 表示当前从起点 i 号结点，使用了 j 次免费通行权限后的最短路径。显然，dis 数组可以这么转移：
$dis_{i,j} = \min\{\min\{dis_{from,j-1}\}, \min\{dis_{from,j} + w\}\}$
其中，from 表示 i 的父亲节点，w 表示当前所走的边的边权。当 $j - 1 \geq k$ 时，$dis_{from,j} = \infty$。
事实上，这个 DP 就相当于把每个结点拆分成了 k+1 个结点，每个新结点代表使用不同多次免费通行后到达的原图结点。换句话说，就是每个结点 u_i 表示使用 i 次免费通行权限后到达 u 结点。

分层图最短路

Picture

绿色是不同层之间的连边，黑色是同一层之间的连边。



图：分层图最短路示意图

目录

差分约束系统

Description

差分约束系统是描述 n 个变量之间关系的 m 个不等式，不等式形如 $a_i <= a_j + k_j$。
注意这里的不等式和我们的松弛操作不等式非常相似，所以为满足上述不等式，我们可以从 $a_j a_i$ 连一条长度为 k_j 的边，然后建立超级源点 S 向每个点连长度为 0 的边。
这样只要存在 $a_i > a_j + k_j$ 的情况，跑最短路就一定会被松弛下去，直到最后满足所有条件。
但是注意差分约数系统建立的不等式关系可能不存在解，也就是有一些条件可能不能满足，这样那些边会不断松弛下去，也就是形成负环，所以判断有没有解只需要判断是否有负环就行。
使用 SPFA 跑最短路和判断负环，一般差分约束系统的题不会卡 SPFA。

P4926 [1007] 倍杀测量者

Description

给出一系列不等式 $x_i >= (k_i - t) \times x_j$ 或 $x_i \times (k_i + t) > x_j$ 和其中一些 x 的值，求最大的 T 使得方程组无解。

P4926 [1007] 倍杀测量者

Solve

二分 t 每次建立差分约束模型跑最短路。
只需要对不等式两边取 log 即可。

目录

同余最短路

Description

当出现形如「给定 n 个整数，求这 n 个整数能拼凑出多少的其他整数(n 个整数可以重复取)」，以及「给定 n 个整数，求这 n 个整数不能拼凑出的最小（最大）的整数」，或者「至少要拼几次才能拼出模 K 余 p 的数」的问题时可以使用同余最短路的方法。

同余最短路利用同余来构造一些状态，可以达到优化空间复杂度的目的。

类比差分约束方法，利用同余构造的这些状态可以看作单源最短路中的点。同余最短路的状态转移通常是这样的 f(i+y) = f(i) + y，类似单源最短路中 f(v) = f(u) +edge(u,v)。

ARC084D - Small Multiple

Description

给定 n，求 n 的倍数中，数位和最小的那一个的数位和。

$n <= 1e5$

ARC084D - Small Multiple

Solve

观察到任意一个正整数都可以从 1 开始，按照某种顺序执行乘 10、加 1 的操作，最终得到，而其中加 1 操作的次数就是这个数的数位和。
这提示我们使用最短路。
对于所有 $0 \leq k \leq n-1$，从 k 向 10k 连边权为 0 的边；从 k 向 k+1 连边权为 1 的边。(点的编号均在模 n 意义下)
每个 n 的倍数在这个图中都对应了 1 号点到 0 号点的一条路径，求出 1 到 0 的最短路即可。某些路径不合法（如连续走了 10 条边权为 1 的边)，但这些路径产生的答案一定不优，不影响答案。

目录

最小环

Description

有向图的最小环可以只包含两个结点，但是无向图的最小环需要至少包含三个结点。
floyd 算法可以做到 $O(n^3)$ 的复杂度：考虑环上肯定有一个点是最大的，那就从这个点把环拆开，$ans = min(ans, dis_{k-1,x,y} + (k,x) + (y,k)$
在边权为 1 的图上，以每个点为起点 BFS 就可以了，
$ans = min(ans, dis_u + dis_v + 1)$ 复杂度 $O(n^2)$。

CF1325E

Description

n 个小于 1e6，因子数量不超过 7 个的正整数，最少取几个乘在一起是平方数？

$n <= 1e5$

CF1325E

Solve

某个数为 $\prod_{i=1}^{n} p_i^{k_i}$，则因子个数为 $\prod_{i=1}^{n}(k_i+1)$。
那么如果有 3 个质因子，就有 8 个因子了，所以一个数最多有 2 个质因子。
如果一个数的质因子的幂是偶数，那么它对答案没有贡献，所以只计算它的幂次是奇数的质因子有哪些。
那么就两种情况:
有两个 p 和 q，则 p 到 q 连一条无向边。
只有一个 p，则 1 到 p 连一条无向边。
之后在这个无向图上求最小环，非常的妙，因为选中一条边实际上就是选择一个数，环上的每个质因子都出现了两次那最后就是一个平方数，然后最小环所以可以保证选择的数的个数最少。
因为边权都是 1 所以直接用 BFS 找最小环，然后注意到每个数不超过 1e6，那么 [1,1000] 这个区间上肯定有在最后答案中的最小环上的点，从这些点开始 BFS 就行。

目录

Kruskal and Prim

Description

回顾一下两种算法。
K 算法使用反证法简单证明：
假设 k 算法不成立，在 mst 之外的边可以替换 mst 里面的边可以使得生成树更小。
考虑为什么一条边不会在 mst 里面，那就是在它加入的时候 u 和 v 已经连好了，再加入它会形成环。
因为排序，所以环上这条边是最大的，替换掉其它边不会使得 mst 更小。
prim 算法和 dijkstra 本质相同，都是分成两个集合然后贪心找最近的。
小思考：k 算法和 bellman-ford 都是从边的角度考虑解决问题，prim 和 Dijkstra 都是从点的角度考虑解决问题。

CF125E

Description

求 1 号点出度为 k 的最小生成树。

$n <= 5000, m <= 1e5$

CF125E

solve

先求出最小生成树。
如果 1 号点的度数小于 k 那么减去 x，如果大于 k 那么加上 x。

目录

非严格次小生成树

Description

在无向图中，边权和最小的满足边权和大于等于最小生成树边权和的生成树。
先求出 MST，再尝试拿不在 MST 中的边替换，加进去会变成一个环，替换掉环中最大的边得到 $MST - max + e$，所有得到的数值里面取最小的即可。
使用倍增维护 MST 上每个点到其 2^k 级祖先路径上的最大值即可。

严格次小生成树

Description

在无向图中，边权和最小的满足边权和严格大于最小生成树边权和的生成树。
维护到 2^i 级祖先路径上的最大边权的同时维护严格次大边权，当用于替换的边的权值与原生成树中路径最大边权相等时，用严格次大值来替换即可。

目录

树的直径

Description

树的直径是树上最长的一条路径。
可以通过树形 DP 和两次 DFS 得到。
接下来先证明两次 DFS 的正确性。
从任何一个结点 y 出发，先 dfs 找到离 y 最远的结点 t，再从 t 找到离它最远的结点 s，(s, t) 这条路径就是树的直径。
当找到 t 之后离 t 最远的结点 s 显然满足树的直径的定义，所以只需要证明从结点 y 出发找到的最远的点是 t 即可。
分三种情况讨论。

两次 DFS

Description

如果结点 y 在 (s,t) 树的直径上，假设离 y 最远的结点不是 t，而是 z，那么 (y,t) 比 (y,z) 短，将树的直径上的 (y, t) 替换为 (y,z) 比原来更长，这与 (s,t) 是树的直径矛盾。

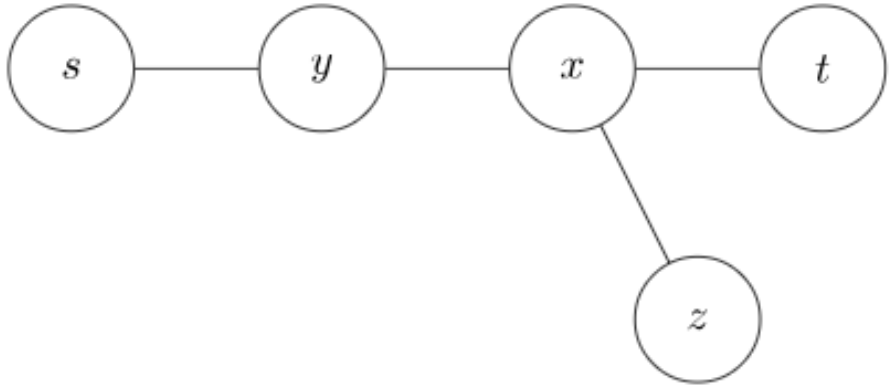


图: y 在 (s,t) 上的情况

两次 DFS

Description

如果结点 y 不在 (s,t) 树的直径上，但是 (y,z) 和树的直径有重复部分。那么 (x',t) 的距离肯定小于 (x',z)，同理将这一段进行替换会使得 (s,t) 不符合树的直径的定义。

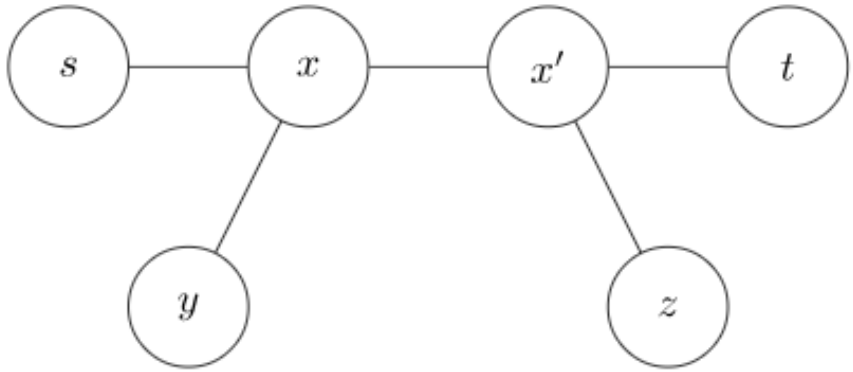


图: y 不在 (s,t) 上，但是 (y,z) 和 (s,t) 有重复部分

两次 DFS

Description

如果结点 y 不在 (s,t) 树的直径上，并且 (y,z) 和树的直径没有重复部分，则 (x',z)>(x',t)，所以 (x,z) 大于 (x,t)，同样不符合树的直径的定义。

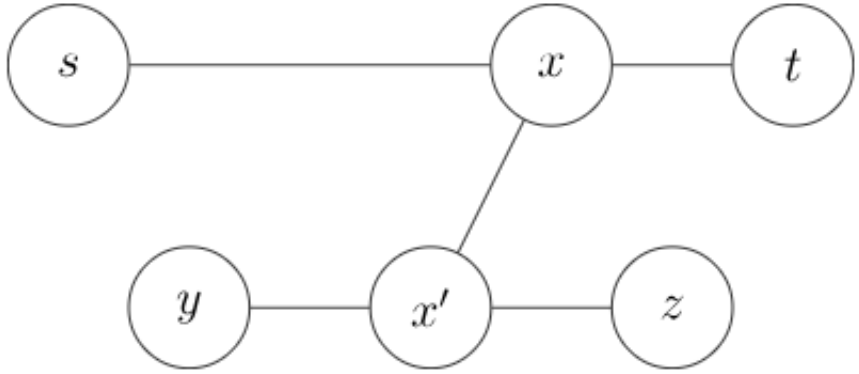


图: y 不在 (s,t) 上，且 (y,z) 和 (s,t) 无重复部分

树形 DP 代码实现

```
1  void dfs(int u, int fa) {
2    for (int v : E[u]) {
3      if (v == fa) continue;
4      d[v] = d[u] + 1;
5      if (d[v] > d[c]) c = v;
6      dfs(v, u);
7    }
8  }
9
10 int main() {
11   scanf("%d", &n);
12   for (int i = 1; i < n; i++) {
13     int u, v;
14     scanf("%d %d", &u, &v);
15     E[u].push_back(v), E[v].push_back(u);
16   }
17   dfs(1, 0);
18   d[c] = 0, dfs(c, 0);
```

树形 DP

Description

树的直径肯定在某个点可以裂开成两条链相连。
从叶子向根结点回溯遍历，维护每个结点往下连接的最长链和次长链，和前面提过的次短路同样套路，对于每个结点都将它作为直径上裂开的那个点进行尝试更新答案。

树形 DP 代码实现

```
1  void dfs(int u, int fa) {
2    d1[u] = d2[u] = 0;
3    for (int v : E[u]) {
4      if (v == fa) continue;
5      dfs(v, u);
6      int t = d1[v] + 1;
7      if (t > d1[u])
8        d2[u] = d1[u], d1[u] = t;
9      else if (t > d2[u])
10       d2[u] = t;
11   }
12   d = max(d, d1[u] + d2[u]);
13 }
```

目录

树链剖分

Description

还记得之前提过的 DFS 序吗？树链剖分将树上的一条路径剖分为不超过 $O(\log n)$ 条 DFS 序连续的树链，这样树上的路径问题转换为序列上的操作问题，可以使用常见的数据结构如线段树分块等维护答案。
定义重子节点表示其子节点中子树最大的子结点。如果有多个子树最大的子结点，取其一。如果没有子节点，就无重子节点。
定义轻子节点表示剩余的所有子结点。
从这个结点到重子节点的边为重边。
到其他轻子节点的边为轻边。
若干条首尾衔接的重边构成重链。
把落单的结点也当作重链，那么整棵树就被剖分成若干条重链。

树链剖分

Picture

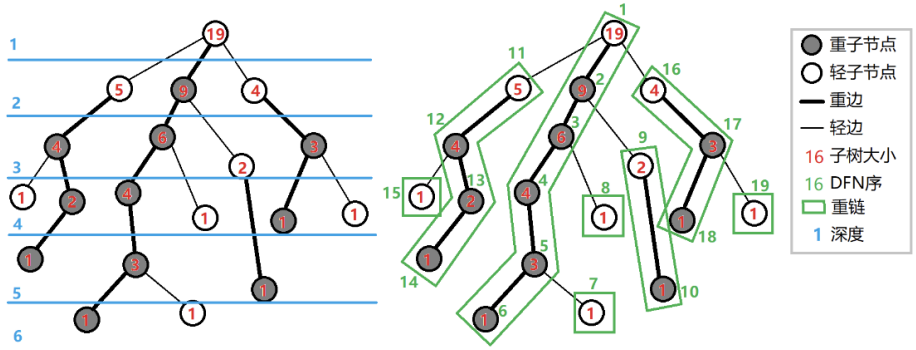


图: 重链剖分

树链剖分

Description

第一个 DFS 记录每个结点的父节点（father）、深度（deep）、子树大小（size）、重子节点（hson）。
第二个 DFS 记录所在链的链顶（top，应初始化为结点本身）、重边优先遍历时的 DFS 序（dfn）、DFS 序对应的节点编号（rank）。

树形 DP 代码实现

```
1  void dfs1(int o) {
2    son[o] = -1;
3    siz[o] = 1;
4    for (int j = h[o]; j; j = nxt[j])
5      if (!dep[p[j]]) {
6        dep[p[j]] = dep[o] + 1;
7        fa[p[j]] = o;
8        dfs1(p[j]);
9        siz[o] += siz[p[j]];
10       if (son[o] == -1 || siz[p[j]] > siz[son[o]]) son[o]
             = p[j];
11     }
12 }
```

树形 DP 代码实现

```
1  void dfs2(int o, int t) {
2    top[o] = t;
3    cnt++;
4    dfn[o] = cnt;
5    rnk[cnt] = o;
6    if (son[o] == -1) return;
7    dfs2(son[o], t);  // 优先对重儿子进行 DFS, 可以保证同一
         条重链上的点 DFS 序连续
8    for (int j = h[o]; j; j = nxt[j])
9      if (p[j] != son[o] && p[j] != fa[o]) dfs2(p[j], p[j]);
10 }
```

树链剖分

Description

树上每个节点都属于且仅属于一条重链。
重链开头的结点不一定是重子节点（因为重边是对于每一个结点都有定义的。
所有的重链将整棵树完全剖分。
在剖分时重边优先遍历，最后树的 DFN 序上，重链内的 DFN 序是连续的。按 DFN 排序后的序列即为剖分后的链。
一颗子树内的 DFN 序是连续的。
可以发现，当我们向下经过一条轻边时，所在子树的大小至少会除以二。
因此，对于树上的任意一条路径，把它拆分成从 lca 分别向两边往下走，分别最多走 $O(\log n)$ 次，因此，树上的每条路径都可以被拆分成不超过 $O(\log n)$ 条重链。

树链剖分求 LCA

Description

类比倍增向上跳重链，每次跳深度较大的那个。

树链剖分求 LCA

```
1 int lca(int u, int v) {
2   while (top[u] != top[v]) {
3     if (dep[top[u]] > dep[top[v]])
4       u = fa[top[u]];
5     else
6       v = fa[top[v]];
7   }
8   return dep[u] > dep[v] ? v : u;
9 }
```

目录

That's all! 3KU!