14. 图论

树与图的存储,因树是无环连通图,故都可用图的存储方式.图分为无向图和有向图,其中无向图可看作连通的两节点间有双向的边的有向图,故都可用有向图的存储方式.

有向图一般有两种存储方式:

①邻接矩阵:用一个二维数组$g[][]$存两节点间边的信息,空间复杂度$O(n^2)$,适合用于存储稠密图.求最短路时,对重边只保留一条.

1)对无权图,$g[a][b]=1$表示有一条从节点a指向节点b的有向边;$g[a][b]=0$表示节点a和b不连通.

2)对有权图,$g[a][b]=$边权,边权为INF时表示节点不连通.

②邻接表:类似于拉链法建哈希表,在每一个节点处拉一个单链表,存该节点可走向哪个点,每个链表内节点的顺序无关紧要.连一条新的有向边时,只需将连接的节点插到对应的链表头.代码:

```
const int MAXN = 1e5 + 5, MAXM = MAXN * 2;
int head[MAXN], edge[MAXM], nxt[MAXM], idx = 0;

void add(int a, int b) {  // 连一条a->b的有向边
  edge[idx] = b, nxt[idx] = head[a], head[a] = idx++;
}

int main() {
  memset(head, -1, sizeof(head));  // 初始化

  int a, int b;

  add(a, b), add(b, a);  // 连无向边

  add(a, b);  // 连a->b的有向边
}

```

一般用数组存图,vector效率不如数组.

树与图的遍历,都可用遍历有向图的方法,一般用DFS或BFS,且每个节点只遍历一次.设有n各节点,m条边,则DFS和BFS的时间复杂度都是$O(n+m)$.模板:

①DFS模板:

```
const int MAXN = 1e5 + 5, MAXM = MAXN * 2;
int head[MAXN], edge[MAXM], nxt[MAXM], idx = 0;
bool vis[MAXN];  // 记录每个节点是否被遍历过

void dfs(int u) {  // u是当前遍历到的节点的编号
  vis[u] = true;

  for (int i = head[u]; ~i; i = nxt[i]) {
    int j = edge[i];  // 下一个遍历的节点的编号
    if (!vis[j]) dfs(j);
  }
}
```

②BFS模板:

```
const int MAXN = 1e5 + 5;
int head[MAXN], edge[MAXN], nxt[MAXN], idx = 0;

void bfs() {
  queue<int> que;  // 存状态

  que.push(1);// 初始化

  while (!que.empty()) {
    auto t = que.front(); que.pop();

    // 扩展t
  }
}

```

14.1 树与图的DFS

14.1.1 树的重心

树的重心是树上的一个节点,若将其删除,则剩余各连通块内点数的最大值最小.树的重心不唯一.

题意

给定一棵包含n $(1 \leq n \leq 1e5)$个节点、编号为$1 \sim n$和$(n-1)$条无向边的树.找到树的重心,并输出将重心删除后剩余各连通块内点数的最大值.

思路

只需枚举删除每个节点后其子树的大小,用DFS.与被删除的节点的父亲节点连通的部分的大小即总点数减去被删除的节点的子树的大小.

代码

```
int n;  // 节点数
const int MAXN = 1e5 + 5, MAXM = MAXN * 2;  // 无向边开两倍
int head[MAXN], edge[MAXM], nxt[MAXM], idx = 0;
bool vis[MAXN];  // 记录每个节点是否被遍历过
int ans = MAXN;  // 记录最大值的最小值

void add(int a, int b) {
  edge[idx] = b, nxt[idx] = head[a], head[a] = idx++;
}

int dfs(int u) {  // 返回以u为根的子树的节点数
  vis[u] = true;

  int res = 0;  // 记录各连通块大小的最大值
  int sum = 1;  // 以u为根的子树的大小,u也是一个节点,故从1开始

  for (int i = head[u]; ~i; i = nxt[i]) {
    int j = edge[i];  // 下一个遍历的节点的编号

```

```
    if (!vis[j]) {
      int s = dfs(j);  // 以j为根的子树的大小
      res = max(res, s);
      sum += s;  // 更新以u为根的子树的大小
    }
  }

  res = max(res, n - sum);  // n-sum为与u的父亲节点连通的部分的大小
  ans = min(ans, res);  // 更新各连通块大小最大值的最小值

  return sum;
}

int main() {
  memset(head, -1, sizeof(head));  // 初始化

  for (int i = 0; i < n - 1; i++) {
    int a, b; cin >> a >> b;
    add(a, b), add(b, a);
  }

  dfs(1);  // 从哪个节点开始搜都行
  cout << ans;
}
```

14.1.2 Split Into Two Sets

题意 (2 s)

有t $(1 \le t \le 1e4)$组测试数据.每组测试数据输入一个偶数n $(2 \le n \le 2e5)$,表示牌数.每张牌上有$1 \sim n$中的两个数字.现需将牌分为两个集合,使得每个集合中牌上的数字不重复,若能做到,输出"YES",否则输出"NO".数据保证所有测试数据的n之和不超过$2e5$.

思路I:图匹配

显然一张牌的两数字相同,或$1 \sim n$中有数字出现超过2次时无解.

若存在牌$\{a, b\}$,则节点a与b间连双向边,并更新每个节点的度数.遍历一遍节点$1 \sim n$,检查每个节点的度数是否为2,若不为2,则不满足.同时记录连通的节点数cnt,若遍历完后cnt为奇数,则无解.

代码I

```
const int MAXN = 2e5 + 5;
int n;

int main() {
  CaseT{
    cin >> n;

    bool ok = true;
    multiset<int> ss;
    vi edges[n + 1], deg(n + 1);
```

```
    for (int i = 0; i < n; i++) {
      int a, b; cin >> a >> b;

      if (ok && a == b) ok = false;

      if (ok) {
        ss.insert(a), ss.insert(b);
        edges[a].push_back(b), edges[b].push_back(a);  // 建双向边
        deg[a]++, deg[b]++;  // 更新节点度数
      }
    }

    for (int i = 1; i <= n; i++) {
      if (ss.count(i) != 2) {
        ok = false;
        break;
      }
    }

    if (!ok) {
      cout << "NO" << endl;
      continue;
    }

    vb vis(n + 1);
    int cnt = 0;  // 遍历到的节点数
    bool flag = true;

    function<void(int)> dfs = [&](int u)->void {  // 遍历一遍连通的节点,检查度数
      if (vis[u]) return;

      vis[u] = true;
      if (deg[u] != 2) flag = false;
      cnt++;
      for (int v : edges[u]) dfs(v);
    };

    for (int i = 1; i <= n; i++) {
      if (!vis[i]) {
        cnt = 0;
        flag = true;
        dfs(i);

        if (flag && (cnt & 1)) {
          ok = false;
          break;
        }
      }
    }

    cout << (ok ? "YES" : "NO") << endl;
  }
}
```

思路II:扩展域并查集

用扩展域并查集维护每个数属于哪个集合.

代码II

```cpp
const int MAXN = 2e5 + 5;
int n;
int fa[MAXN << 1];  // 扩展域并查集

int find(int x) { return x == fa[x] ? x : fa[x] = find(fa[x]); }

void merge(int x, int y) {
  x = find(x), y = find(y);
  if (x != y) fa[x] = y;
}

int main() {
  CaseT{
    cin >> n;
    for (int i = 1; i <= n << 1; i++) fa[i] = i;

    vi deg(n + 1);
    for (int i = 0; i < n; i++) {
      int a, b; cin >> a >> b;
      deg[a]++, deg[b]++;
      merge(a, b + n), merge(a + n, b);
    }

    bool ok = true;
    for (int i = 1; i <= n; i++) {
      if (deg[i] > 2) ok = false;
      if (find(i) == find(i + n)) ok = false;
    }

    cout << (ok ? "YES" : "NO") << endl;
  }
}
```

14.1.3 Path Prefixes

原题指路:https://codeforces.com/contest/1714/problem/G

题意

给定一棵包含编号$1 \sim n$的n个节点的有根树,其中根节点为1.每条边上有两个权值a_j和b_j.求$r_2, r_3, \cdots, r_n$,其中r_i定义为:考察一条从根节点到节点i $(2 \leq i \leq n)$的路径,其上的a_j之和为A_i,则r_i定义为这条路径的最长前缀的长度,使得前缀中的b_j之和不超过A_i.

有t $(1 \leq t \leq 1e4)$组测试数据.每组测试数据第一行输入整数n $(2 \leq n \leq 2e5)$.接下来$(n-1)$行每行输入三个整数p_j, a_j, b_j $(1 \leq p_j \leq n, 1 \leq a_j, b_j \leq 1e9, 2 \leq j \leq n)$,分别表示节点$j$的父亲节点和边$< p_j, j >$上的两个权值.数据保证输入能构成一棵根节点为节点1的有根树,且所有测试数据的n之和不超过$2e5$.

对每组测试数据,输出$r_2, r_3, \cdots, r_n$.

思路I

开一个vector来维护路径和.

代码I

```cpp
int main() {
  CaseT{
    int n; cin >> n;
    vector<vector<array<int, 3>>> edges(n);  // 儿子节点、权值a、权值b
    for (int u = 1; u < n; u++) {
      int v, a, b; cin >> v >> a >> b;
      v--;  // 下标从0开始
      edges[v].push_back({ u,a,b });
    }

    vl path(1, 0);  // 路径和
    vi ans(n);

    function<void(int, ll)> dfs = [&](int u, ll s) {  // 当前节点、路径a之和
      if (u) ans[u] = lower_bound(all(path), s + 1) - path.begin() - 1;

      for (auto& item : edges[u]) {
        int v = item[0];
        path.push_back(path.back() + item[2]);  // 更新b之和
        dfs(v, s + item[1]);  // 更新a之和
        path.pop_back();
      }
    };

    dfs(0, 0);  // 从根节点开始搜,路径a之和为0

    for (int i = 1; i < n; i++) cout << ans[i] << " \n"[i == n - 1];
  }
}
```

思路II

直接维护路径a之和、b之和,将父节点到子节点的边权累加到子节点的点权.

代码II

```cpp
int main() {
  CaseT{
    int n; cin >> n;
    vector<vector<array<int, 3>>> edges(n);  // 儿子节点、权值a、权值b
    for (int u = 1; u < n; u++) {
      int v, a, b; cin >> v >> a >> b;
      v--;  // 下标从0开始
      edges[v].push_back({ u,a,b });
    }

    vl A(n), B(n), s;  // 路径a之和、路径b之和
    vi ans(n);

    function<void(int)> dfs = [&](int u) {  // 当前节点
```

```
      s.push_back(B[u]);
      if (u) ans[u] = upper_bound(all(s), A[u]) - s.begin() - 1;

      for (auto& item : edges[u]) {
        int v = item[0], a = item[1], b = item[2];
        A[v] = A[u] + a, B[v] = B[u] + b;
        dfs(v);
      }
      s.pop_back();
    };

    dfs(0);  // 从根节点开始搜

    for (int i = 1; i < n; i++) cout << ans[i] << " \n"[i == n - 1];
  }
}
```

14.1.4 The Maximum Subtree

原题指路:https://codeforces.com/problemset/problem/1238/F

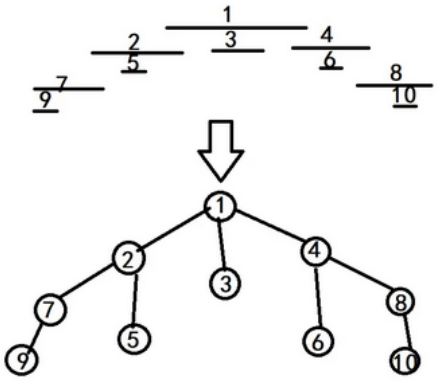
题意 (2 s)

定义一棵包含编号$1 \sim n$的n个节点的树是好的,如果可对每个节点i $(1 \le i \le n)$赋两个值l_i, r_i表示一个区间,使得节点u与v间有边当且仅当u和v的区间有交集.

有t $(1 \le t \le 1.5\text{e}5)$组测试数据.每组测试数据给定一棵包含编号$1 \sim n$的$n$ $(2 \le n \le 3\text{e}5)$个节点的树,求其最大的好的子树的大小.数据保证所有测试数据的n之和不超过$3\text{e}5$.

思路

一棵树对应的线段有如下形式:



最大的好的子树是原树的直径及与其相连的所有节点构成的子树.

设树的直径为$a_1, \cdots, a_k$,节点u的出度为$d[u]$,则$ans = \left(\sum_{i=1}^{k} d[a[i]]\right) - (k-2) = \left(\sum_{i=1}^{k} (d[a[i]] - 1)\right) + 2$,其中$(k-2)$是因为除了链端的两个节点只有入度,其他链中节点都既有入度也有长度,即都被多算了一次.

代码

```cpp
const int MAXN = 3e5 + 5;
int n;
vector<int> edges[MAXN];
int d[MAXN];  // 每个节点的度数-1
int ans, pos;  // 最大的(度数-1)之和、取得最大值的节点

void init() {
  for (int i = 1; i <= n; i++) {
    edges[i].clear();
    d[i] = -1;
  }
}

void dfs(int u, int fa, int sum) {  // 当前节点、前驱节点、当前的(度数-1)之和
  if (sum > ans) ans = sum, pos = u;

  for (auto v : edges[u]) {
    if (v == fa) continue;

    dfs(v, u, sum + d[v]);
  }
}

void solve() {
  cin >> n;
  init();

  for (int i = 1; i < n; i++) {
    int u, v; cin >> u >> v;
    edges[u].push_back(v), edges[v].push_back(u);
    d[u]++, d[v]++;
  }

  ans = -1, dfs(1, -1, d[1]);  // 从任一节点开始搜,找到其中一个(度数-1)之和最大的点
  ans = -1, dfs(pos, -1, d[pos]);  // 从最远点开始搜,找到另一个(度数-1)之和最大的点
  cout << ans + 2 << endl;
}

int main() {
  CaseT  // 单测时注释掉该行
    solve();
}
```

14.1.5 Super M

原题指路: https://codeforces.com/problemset/problem/592/D

题意 (2 s)

给定一棵包含编号$1 \sim n$的n $(1 \leq n \leq 123456)$个节点的树和m $(1 \leq m \leq n)$个关键节点, 每经过一条边需花费1单位时间. 求从任意点出发, 遍历所有关键节点所需的最小时间和取得最小时间时的起点. 若有多个起点, 输出编号最小的起点.

思路

显然应以某个关键节点为起点. 考察以某关键节点到另一个节点的路径, 称之为主链.

①若关键节点在主链上, 则沿主链行进时可遍历到, 消耗1倍距离的时间.

②若关键节点不在主链上, 则沿主链行进时还需遍历不在路径上的节点, 即需遍历支链后返回主链, 消耗2倍距离的时间.

综上, 为使得消耗的时间最少, 应使得消耗1倍距离的时间的节点尽量多, 即使得主链尽量长. 显然最长的主链是相距最远的两个关键节点间的路径, 这可类似于树的直径求得, 即从任一关键节点开始DFS, 求得与其相距最远的关键节点st, 再从st开始DFS, 求得与其相距最远的节点ed, 则最长主链为st与ed间的简单路径, 设其长度为$diam$.

若树的所有节点都是关键节点, 则$ans = 2(n-1) - diam$. 若不然, 对不在最长主链上的非关键节点, 它需被遍历到当且仅当以它为根节点的子树中存在关键节点. $have[u] = true$表示以u为根节点的子树中存在关键节点. 设有cnt个$have$值为$true$的节点, 则$ans = 2(cnt-1) - diam$.

注意特判只有一个关键节点的情况.

代码

```cpp
void solve() {
    int n, m; cin >> n >> m;
    vector<vector<int>> edges(n + 1);
    for (int i = 1; i < n; i++) {
        int u, v; cin >> u >> v;
        edges[u].push_back(v), edges[v].push_back(u);
    }
    vector<bool> key(n + 1);  // key[u]表示以u为根节点的子树中是否包含关键节点
    int last;
    while (m--) {
        cin >> last;
        key[last] = true;
    }

    if (m == 1) {
        cout << last << endl;
        cout << 0 << endl;
        return;
    }

    vector<int> dis(n + 1);  // dis[u]表示根节点到节点u的最短距离
    dis[0] = -1;  // 使得根节点到自身的距离为0

    // 求树的直径: 当前节点、前驱节点
    function<void(int, int)> dfs = [&](int u, int pre) {
        dis[u] = dis[pre] + 1;

        for (auto v : edges[u]) {
            if (v != pre) {
                dfs(v, u);
```

```
                key[u] = key[u] | key[v];
            }
        }
    };

    // 求树的直径
    dfs(last, 0);
    int st = 0, ed = 0;  // 直径的两端点
    for (int u = 1; u <= n; u++) {
        if (key[u] && dis[u] > dis[st])
            st = u;
    }
    dfs(st, 0);
    for (int u = 1; u <= n; u++) {
        if (key[u] && dis[u] > dis[ed])
            ed = u;
    }

    int cnt = 0;
    for (int u = 1; u <= n; u++)
        cnt += key[u];

    cout << min(st, ed) << endl;
    cout << 2 * (cnt - 1) - dis[ed] << endl;
}

int main() {
    solve();
}
```

14.1.6 Tree Queries

原题指路: https://codeforces.com/problemset/problem/1328/E

题意 (2 s)

给定一棵包含编号$1 \sim n$的n $(2 \leq n \leq 2\text{e}5)$个节点的树, 其中节点1为根节点. 给定$m$ $(1 \leq m \leq 2\text{e}5)$个询问, 每个询问给定一个包含$k$ $(1 \leq k \leq n)$个节点的集合$v = [v_1, \cdots, v_k]$ $(1 \leq v_i \leq n)$, 表示询问是否存在一条从根节点1到某个节点u的简单路径, 使得给定的k个节点在这条路径上或距离路径上的节点的距离为1.

思路

考察节点1到节点u的路径.

①若节点v在路径上, 则路径经过v的父亲节点.

②若节点v距离路径的距离为1, 则路径经过v的父亲节点或至少一个儿子节点.

综上, 路径经过集合中各节点的父亲节点,. 因路径是简单路径, 则这些父亲节点在一条链上.

将这些父亲节点按深度非降序排列后, 检查后一个节点是否在前一个节点的子树中即可, 这可用DFS序和子树大小判断.

代码

```cpp
struct DFSOrder {
    int n;
    vector<vector<int>> edges;
    vector<int> fa;
    vector<int> depth;
    vector<int> dfn;
    int tim;
    vector<int> siz;

    DFSOrder(int _n, const vector<vector<int>>& _edges) :n(_n), edges(_edges), tim(0) {
        fa.resize(n + 1), depth.resize(n + 1);
        dfn.resize(n + 1), siz.resize(n + 1);

        dfs(1, 1);
    }

    void dfs(int u, int pre) {
        fa[u] = pre, depth[u] = depth[pre] + 1;
        siz[u] = 1, dfn[u] = ++tim;

        for (auto v : edges[u]) {
            if (v != pre) {
                dfs(v, u);
                siz[u] += siz[v];
            }
        }
    }

    bool isInSubtree(int root, int u) {  // 判断节点u是否在以root为根节点的子树中
        return dfn[root] <= dfn[u] && dfn[u] <= dfn[root] + siz[root] - 1;
    }

    bool check(vector<int>& a) {
        for (auto& ai : a) ai = fa[ai];
        sort(all(a), [&](int u, int v) {
            return depth[u] < depth[v];
        });

        int cnt = a.size();
        for (int i = 1; i < cnt; i++) {
            if (!isInSubtree(a[i - 1], a[i]))
                return false;
        }
        return true;
    }
};

void solve() {
    int n, m; cin >> n >> m;
    vector<vector<int>> edges(n + 1);
    for (int i = 1; i < n; i++) {
        int u, v; cin >> u >> v;
        edges[u].push_back(v), edges[v].push_back(u);
    }

```

```
    DFSOrder solver(n, edges);
    while (m--) {
        int k; cin >> k;
        vector<int> a(k);
        for (auto& ai : a) cin >> ai;

        cout << (solver.check(a) ? "YES" : "NO") << endl;
    }
}

int main() {
    solve();
}
```

14.1.7 The Tag Game

原题指路: https://codeforces.com/problemset/problem/813/C

题意

给定一棵包含编号 $1 \sim n$ $(2 \leq n \leq 2e5)$ 的无向有根树, 其中节点 1 是根节点. A从节点 1 出发, B从节点 s $(2 \leq s \leq n)$ 出发. A和B轮流移动, B先手, 两人每轮可移动到相邻节点或留在当前节点. 若A与B在同个节点, 则游戏结束. A想最小化游戏结束的轮数, B想最大化游戏结束的轮数, 两人都采取最优策略, 求游戏结束的轮数的两倍.

思路

A的最优策略为一直追着B向下走, B的最优策略为先上走到某个节点 u (可能不走)后再向下走到某条链的底端 v , 随后的轮次停在节点 v 处.

显然 u 在以节点 1 和节点 x 为端点的链上, 则游戏结束的轮次即 A 到 v 的步数. 以节点 1 为根节点, 设 v 的深度为 $depth[v]$, 深度从 1 开始, 则游戏结束的轮数为 $(depth[v]-1)$.

从 s 开始DFS, 求出B到各/节点 x 的距离 $dis[x]$, 则对每个节点 $v \in [1,n]$, B 可在游戏结束前到达 v iff B到 v 的距离 $<$ A到 v 的距离, 即 $depth[v] > dis[v]$.

代码

```
void solve() {
    int n, s; cin >> n >> s;  // 节点数、Bob起点
    vector<vector<int>> edges(n + 1);
    for (int i = 1; i < n; i++) {
        int u, v; cin >> u >> v;
        edges[u].push_back(v), edges[v].push_back(u);
    }

    auto get = [&](int root) {  // 求以root为根节点的深度
        vector<int> depth(n + 1);

        function<void(int, int)> dfs = [&](int u, int fa) {
            depth[u] = depth[fa] + 1;

            for (auto v : edges[u])
                if (v != fa) dfs(v, u);
        };

```

```
        dfs(root, 0);
        return depth;
    };

    auto depth = get(1);  // 节点的深度
    auto dis = get(s);  // Bob到节点的距离

    int ans = 0;
    for (int u = 1; u <= n; u++) {
        if (depth[u] > dis[u])
            ans = max(ans, (depth[u] - 1) * 2);
    }
    cout << ans << endl;
}

int main() {
    solve();
}
```

14.1.8 Longest path in a tree

原题指路: https://vjudge.net/problem/SPOJ-PT07Z

题意 (0.5 m)

给定一棵无向无权树, 求该树中最长的简单路径的长度.

思路

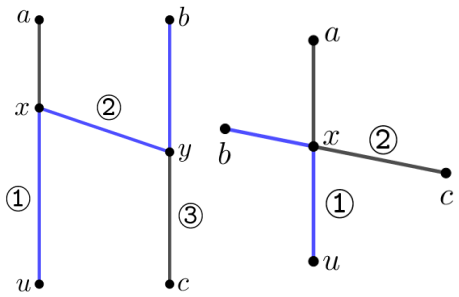
无权树中, 两点间经过的边数的路径称为树的直径. 树的直径可能不唯一, 但直径的长度唯一.

无权树求直径方法:

① 任取一节点作为起点, 找到距离该节点最远的节点 u .

② 以 u 为起点, 找到距离 u 最远的节点 v , 则 u 与 v 间的路径是一条直径.

[**证**] 先证 u 是某条直径的端点. 设从 a 出发, 距 a 最远的节点是 u .



① 如上图左图, 若 au 与树的直径 bc 不相交, 因树连通, 则 $\exists x \in$ 直径$au, y \in$ 直径$bc\ s.t.\ x$ 与 y 连通.

因距 u 最远的节点为 a , 则 ① $\geq$ ② + ③ , 进而 ① + ② $\geq$ ① − ② $\geq$ ③ .

这表明: 蓝色路径长度 $\geq$ 直径 bc 的长度, 进而 u 是某条直径的端点.

② 如上图右图, 若 au 与树的直径 bc 交于点 x .

因距 u 最远的节点为 a , 则 ① $\geq$ ② , 进而蓝色路径长度 $\geq$ 直径 bc 的长度, 进而 u 是某条直径的端点.

综上, u 是某一直径的端点, 进而以 u 为起点, 找到距离 u 最远的节点 v , 则 u 与 v 间的路径是一条直径.

代码

```cpp
vector<vector<int> > edges;
vector<int> dis;
int st = 0;  // 直径的一个端点

void dfs(int u, int fa) {  // 当前节点、前驱节点
    for (auto v : edges[u]) {
        if (v != fa) {
            dis[v] = dis[u] + 1;
            if (dis[v] > dis[st]) st = v;
            dfs(v, u);
        }
    }
}

void solve() {
    int n; cin >> n;
    edges = vector<vector<int> >(n + 1);
    for (int i = 1; i < n; i++) {
        int u, v; cin >> u >> v;
        edges[u].push_back(v), edges[v].push_back(u);
    }

    dis = vector<int>(n + 1);
    dfs(1, 0);  // 以任一节点为起点DFS, 该节点无前驱节点

    // 以st为起点DFS
    dis[st] = 0;
    dfs(st, 0);
    cout << dis[st] << endl;
}

int main() {
    solve();
}
```

14.1.9 Two Paths

原题指路: https://codeforces.com/problemset/problem/14/D

题意 (2 s)

给定一棵包含 n $(1 \le n \le 200)$ 个节点的树, 求其中两条不相交的路径, 使得两路径长度之积最大, 输出该最大值.

思路

考察两条不相交的路径, 因树连通, 则存在连接两路径的第三条路径. 为使得两路径长度之积最大, 应使得两路径尽量长, 则最优方案中连接两路径的第三条路径恰有一条边.

枚举连接两路径的边, 将该边删去, 则原树分裂为两棵树, 分别求这两棵树的直径并更新答案即可. 无需真实地将边删去, 只需DFS时认为该边的两端点互为前驱节点即可.

总时间复杂度 $O(n^2)$.

代码

```cpp
void solve() {
    int n; cin >> n;
    vector<pair<int, int>> eds;
    vector<vector<int>> edges(n + 1);
    for (int i = 1; i < n; i++) {
        int u, v; cin >> u >> v;

        eds.push_back({ u, v });
        edges[u].push_back(v), edges[v].push_back(u);
    }

    int ans = 0;
    for (auto it : eds) {  // 枚举删除的边
        int delU = it.first, delV = it.second;
        vector<int> dis(n + 1);
        int st = 0;  // 直径的一个端点

        function<void(int, int)> dfs = [&](int u, int pre) {
            for (auto v : edges[u]) {
                if (v != pre) {
                    if ((u == delU && v == delV) || (u == delV && v == delU)) continue;

                    dis[v] = dis[u] + 1;
                    if (dis[v] > dis[st]) st = v;

                    dfs(v, u);
                }
            }
        };

        // 求第一棵树的直径
        dfs(delU, delV);
        dis[st] = 0;
        dfs(st, 0);
        int diam = dis[st];

        // 清空
        fill(all(dis), 0);
        st = 0;

        // 求第二棵树的直径
        dfs(delV, delU);
        dis[st] = 0;
        dfs(st, 0);

        ans = max(ans, diam * dis[st]);
    }
    cout << ans << endl;
}

int main() {
    solve();
}
```

14.2 树与图的BFS

14.2.1 BFS求最短路(边权相等)

题意

给定一个有n $(1 \leq n \leq 1e5)$个点和m $(1 \leq m \leq 1e5)$条边的有向图,图中可能有重边和自环.各边长度都为1,点编号$1 \sim n$.输出1号点到n号点的最短距离,若无法从1号点走到n号点,则输出-1.

有向图用m行输入描述,每个输入包含两个整数a和b,表示存在一条从a走到b的长度为1的边.

思路

因边权相等,故可用BFS求最短路.

代码

```cpp
int n, m;  // 节点数和边数
const int MAXN = 1e5 + 5;
int head[MAXN], edge[MAXN], nxt[MAXN], idx = 0;
int dis[MAXN];

void add(int a, int b) {
  edge[idx] = b, nxt[idx] = head[a], head[a] = idx++;
}

int bfs() {
  queue<int> que;  // 存状态

  // 初始化
  memset(dis, -1, sizeof(dis));  // 初始化
  dis[1] = 0;
  que.push(1);

  while (!que.empty()) {
    auto t = que.front(); que.pop();

    for (int i = head[t]; ~i; i = nxt[i]) {
      int j = edge[i];  // 下一个要遍历的节点的编号
      if (dis[j] == -1) {
        dis[j] = dis[t] + 1;
        que.push(j);
      }
    }
  }

  return dis[n];
}

int main() {
  memset(head, -1, sizeof(head));  // 初始化

  while (m--) {
    int a, b; cin >> a >> b;
    add(a, b);
  }

```

```
41    cout << bfs();
42  }
43
```

14.2.2 Timofey and Black-White Tree

原题指路:https://codeforces.com/contest/1790/problem/F

题意 (4 s)

给定一棵包含编号$1 \sim n$的n个节点的树,每个节点有一个颜色,初始时除c_0号节点外,其余节点都为白色,c_0号节点为黑色.定义该树的权值为任意两个黑色节点间的距离的最小值.现进行$(n-1)$次操作,其中第$i\ \ (1 \le i \le n-1)$次操作选择一个当前为白色的节点c_i,将其变为黑色.每个操作后询问当前树的权值.

有$t\ \ (1 \le t \le 1\text{e}4)$组测试数据.每组测试数据第一行输入两个整数$n, c_0\ \ (2 \le n \le 2\text{e}5, 1 \le c_0 \le n)$.第二行输入$(n-1)$个相异的整数$c_1, \cdots, c_{n-1}\ \ (1 \le c_i \le n, c_i \ne c_0)$.接下来$(n-1)$行每行输入两个整数$u, v\ \ (1 \le u, v \le n, u \ne v)$,表示节点$u$与节点$v$间存在无向边.数据保证所有测试数据的$n$之和不超过$2\text{e}5$.

思路I

将c_0号节点作为第0次操作染成黑色的节点,当操作次数≥ 1时输出答案.

第$i\ \ (0 \le i \le n-1)$次操作将节点c_i染成黑色后,以节点c_i为起点BFS,求出c_i到其余节点的最短路$dis[]$,同时维护当前两黑色节点间距离的最小值ans.操作次数$i \ge 1$时,答案为$\min\{ans, dis[c[i]]\}$.初始时$ans = n$,即最坏情况两个黑色节点分别在一条链的两个端点处.显然操作过程中ans不增,则BFS时只需对当前路径长度$< ans$的节点更新$dis[]$值.

设节点u的$dis[u]$值会被更新cnt_u次,则上述做法的时间复杂度为$\sum\limits_u cnt_u \cdot deg_u$,其中$deg_u$为$u$的度数.下面将证明$cnt_u = O\left(\sqrt{n}\right)$,进而时间复杂度为$O\left(\sqrt{n} \cdot \sum\limits_u deg_u\right) = O\left(\sqrt{n} \cdot 2n\right) = O\left(n\sqrt{n}\right)$,可通过.

[引理I] 对$n \in \mathbb{N}^*$,在区间$[1, n]$上任取$\left(\left\lceil\sqrt{n}\right\rceil + 1\right)$个点,则任意两点间的距离的最小值不超过$\left\lceil\sqrt{n}\right\rceil$.

[证] 设$m = \left\lceil\sqrt{n}\right\rceil$,点依次为$1 \le p_1 < \cdots < p_{m+1} \le n$.

设点p_i与点$p_j\ \ (1 \le i < j \le m+1)$间的距离为$dis(p_i, p_j)$.

若$dis(p_i, p_j)$为任意两点间的距离的最小值,则$j = i+1$,否则将p_j替换为p_{i+1}可使得$dis(p_i, p_j)$减小.

WLOG,设(p_i, p_{i+1})为唯一一对取得任意两点间的距离的最小值的点对.

反证法,若$dis(p_i, p_{i+1}) = d > m$,则对$\forall (p_j, p_{j+1})\ \ (j \ne i)$,有$dis(p_j, p_{j+1}) > d > m$.

则$dis(p_1, p_{m+1})$

$= dis(p_1, p_2) + \cdots + dis(p_{i-1}, p_i) + dis(p_i, p_{i+1}) + dis(p_{i+1}, p_{i+2}) + \cdots + dis(p_m, p_{m+1})$

$> (m-1)d + d = m \cdot d > m^2 \ge n$,矛盾.

[注] 当点中有一个在1处,一个在n处,剩下的节点在$(1, n)$上均匀分布时,取得任意两节点间的距离的最小值的最大值.

[引理II] 在一棵包含n个节点的树上任取$\left(\left\lceil\sqrt{n}\right\rceil + 1\right)$个节点,则任意两节点间的距离的最小值的最大值不超过$\left\lceil\sqrt{n}\right\rceil$.

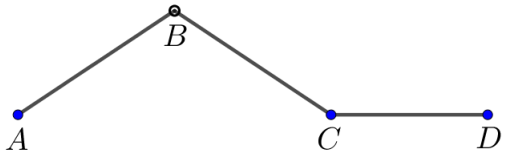
[证] 讨论树的形态,树的形态为一条链时,由**引理I**即证.下面讨论树的形态有分支的情况.

以树的直径所在的链为主链,其余链为支链.

(1)树除主链外只有一条支链.

如下图,黑色实线$ABCD$为树的主链,其包含n个节点,从中选取$(\lceil n \rceil + 1)$个节点.

设任意两节点间的距离的最小值的最大值$dis(C,D) \leq \lceil \sqrt{n} \rceil$.



下面讨论在主链的节点上挂一条支链时对任意两节点间的距离的最小值的最大值的影响.

①若在已选取的节点(蓝色)上挂一条支链,因在直径的两端点处无支链,故支链只能挂在直径上选取的非端点处.

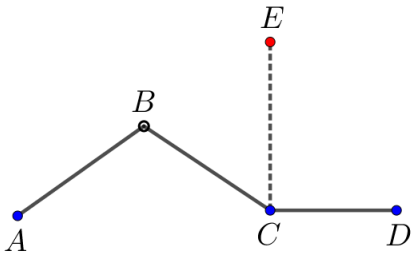
不妨设在C节点处挂一条链端为节点E的支链.(该例具有一定的特殊性,但类似地可讨论其他情况)

设支链CE上有m个节点,下证任意两节点间的距离的最小值的最大值$\leq \lceil \sqrt{n+m} \rceil$.

(i)$m = 1$时,

i)若n为完全平方数,则加入支链CE后需再选取一个节点.

如下图,显然取E节点时可使得链上的选取有机会变得更加"稀疏".

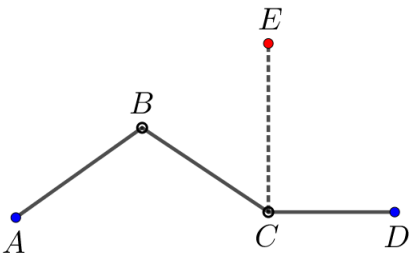


此时无论$dis(C,D)$为何≥ 1的值,$dis(C,E)$都是取得任意两节点间距离的最小值的点对.

此时$dis(C,E) = 1 \leq \lceil \sqrt{n+1} \rceil$.

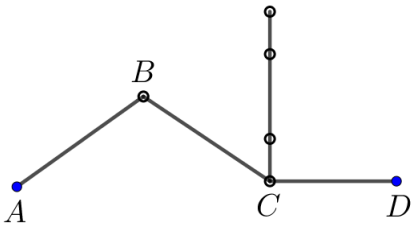
ii)若n非完全平方数,则加入支链CE后无需再取一个节点,

则将原本选取的节点之一改为节点E可使得链上的选取有机会变得更加"稀疏".

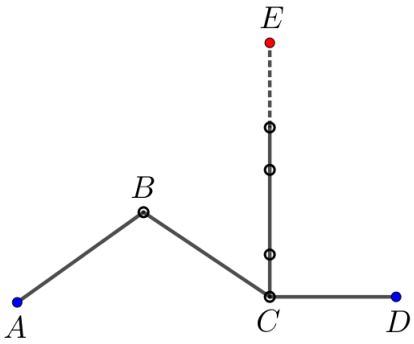


具体地,变得更加"稀疏"至多使得距离的最小值$+1$,若不然,由反证法易得矛盾.

(ii)假设$m = m_0$时结论成立.如下图,将此时的支链作为主链的一部分.



(iii)$m = m_0 + 1$时,设新加入的节点为E.类似于(i)的分析即证.



②若在未选中的节点(空心黑色)上挂一条支链,如下图,不妨设在B节点处挂一条链端为节点E的支链.

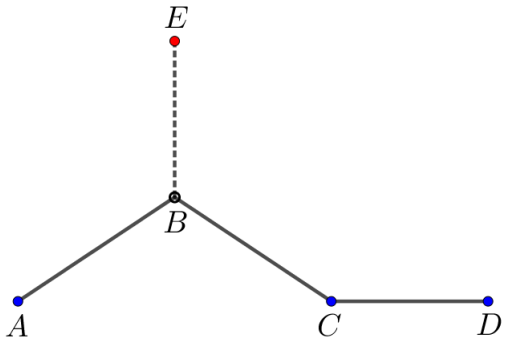
设支链BE上有m个节点,下证任意两节点间的距离的最小值的最大值$\leq \left\lceil \sqrt{n+m} \right\rceil$.

(i)$m=1$时,

i)若n为完全平方数,则加入支链BE后需再选取一个节点.

因只有主链时,$dis(C,D)$是任意两节点间距离的最小值,则$dis(A,C) \geq dis(C,D)$.

如下图,显然选取E节点可使得链上的选取有机会变得更加"稀疏".



因链$ABCD$是直径,则$dis(B,E) \leq dis(B,A)$,且$dis(B,E)=dis(B,A)$的情况是平凡的.

若$dis(B,E)<dis(B,A)$,则
$dis(C,E)=dis(B,C)+dis(B,E)<dis(B,C)+dis(B,A)=dis(C,A)$.

(i)若$dis(C,E) \geq dis(C,D)$,则$dis(C,D)$仍是任意两节点间距离的最小值,

此时$dis(C,D) \leq \left\lceil \sqrt{n} \right\rceil \leq \left\lceil \sqrt{n+1} \right\rceil$.

(ii)若$dis(C,E)<dis(C,D)$,因$dis(C,E)=dis(C,B)+dis(B,E)>dis(C,B)$,

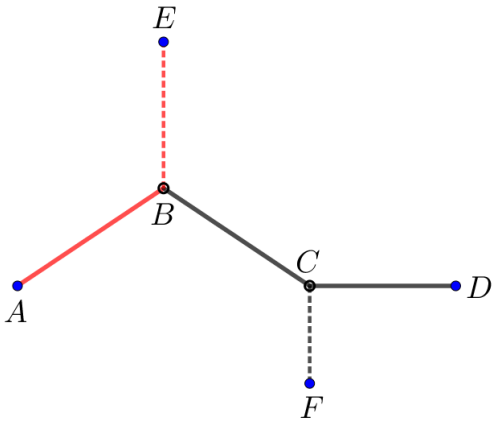
则$dis(C,B)<dis(C,D)$,与只有主链时任意两节点间的距离的最小值为$dis(C,D)$矛盾.

③显然上述分析过程可推广至任意长度的主链和支链.

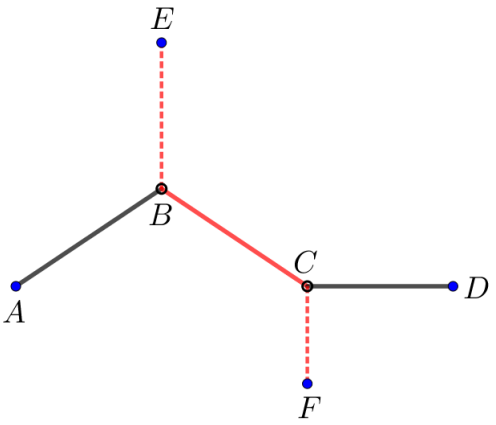
(2)树除主链外有多条支链,WLOG,考察有两条支链的情况.

若取的节点中包含支链上远离主链的节点,则其余取的节点包含支链与主链连接处的节点的情况是平凡的.

①若取得任意两节点间的距离的最小值的点对只涉及一条支链,如下图红色路径所示,则与(1)无本质区别.



②取得任意两节点间的距离的最小值的点对涉及两条支链,如下图红色路径所示.



$dis(E,F) \leq \min\{dis(A,E), dis(A,D), dis(D,F)\}$.

因$dis(A,D)$是树直径的长度,则$dis(E,F) \leq dis(A,D)$是平凡的.

(i)若$dis(A,E) \leq dis(D,F)$,即$dis(A,B)+dis(B,E) \leq dis(D,C)+dis(C,F)$时,

因只有主链时$dis(C,D)$是任意两节点间的距离的最小值,

则$dis(A,B) \geq dis(D,C)$,进而$dis(B,E) \leq dis(C,F)$.

```
因$dis(C,F)\leq dis(C,D)$,否则与链$ABCD$是树的直径矛盾.
```

因$dis(A,E) \geq dis(E,F)$,则$dis(A,B)+dis(B,E) \geq dis(E,B)+dis(B,C)+dis(C,F)$,

即$dis(A,B) \geq dis(B,C)+dis(C,F) > dis(B,C)$.

i)若$dis(B,C)+dis(C,F) \leq dis(C,D)$,则$dis(B,C) < dis(C,D)$,

与只有主链时$dis(C,D)$是任意两节点间的距离的最小值矛盾.

ii)若$dis(B,C)+dis(C,F) > dis(C,D)$,

todo: 证明比少一个节点的情况的边数至多多1

(ii)若$dis(A,E) > dis(D,F)$,

因$dis(E,F) \leq dis(D,F)$,则$dis(E,B)+dis(B,C)+dis(C,F) \leq dis(D,C)+dis(C,F)$,

即$dis(D,C) \geq dis(E,B)+dis(B,C)$,则$dis(D,C) > dis(B,C)$.

与只有主链时任意两节点间的距离的最小值为$dis(C,D)$矛盾.

[**定理**] 包含n个节点的树中每个节点u的更新次数$cnt_u = O\left(\sqrt{n}\right)$.

[**证**] 为求cnt_u的上界,则只需考察ans的上界,问题转化为求树上两节点间的距离的最小值的最大值.

由**引理II**:前$O\left(\sqrt{n}\right)$次染色会使得$ans \leq O\left(\sqrt{n}\right)$,

故$cnt_u = O\left(\sqrt{n}\right)$.

代码I

```cpp
void solve() {
  int n; cin >> n;
  vector<int> c(n);  // 每次操作变为白色的节点
  for (auto& ci : c) cin >> ci;
  vector<vector<int>> edges(n + 1);
  for (int i = 1; i < n; i++) {
    int u, v; cin >> u >> v;
    edges[u].push_back(v), edges[v].push_back(u);
  }

  int ans = n;
  vector<int> dis(n + 1, n);

  auto bfs = [&](int i) {  // 求c[i]号节点到其他节点的最短路
    queue<int> que;
    que.push(c[i]);
    dis[c[i]] = 0;

    while (que.size()) {
      int u = que.front(); que.pop();
      for (auto v : edges[u]) {
        if (dis[u] + 1 < min(dis[v], ans)) {
          dis[v] = dis[u] + 1;
          que.push(v);
        }
      }
    }
  };

  bfs(0);  // 初始时c[0]号节点为白色
  for (int i = 1; i < n; i++) {
    cout << (ans = min(ans, dis[c[i]])) << " \n"[i == n - 1];
    bfs(i);
  }
}

int main() {
  CaseT
    solve();
}
```

190936840	Jan/28/2023 14:49^{UTC+8}	Hytidel	F - Timofey and Black-White Tree	GNU C++17 (64)	Accepted	1045 ms	11400 KB

思路II

代码II

```cpp
void solve() {
  int n; cin >> n;
  vector<int> c(n);  // 每次操作变为白色的节点
  for (auto& ci : c) cin >> ci;
  vector<vector<int>> edges(n + 1);
  for (int i = 1; i < n; i++) {
    int u, v; cin >> u >> v;
    edges[u].push_back(v), edges[v].push_back(u);
  }

  vector<int> fa(n + 1);  // 节点的父亲节点

  function<void(int)> dfs = [&](int u) {  // 预处理fa[]
    for (auto v : edges[u]) {
      if (v != fa[u]) {
        fa[v] = u;
        dfs(v);
      }
    }
  };

  dfs(1);

  int ans = n;
  vector<int> dis(n + 1, n);
  for (int i = 0; i < n; i++) {
    // cur为当前节点,cnt为路径经过的边数
    for (int cur = c[i], cnt = 0; cur && cnt < ans; cur = fa[cur], cnt++) {
      ans = min(ans, dis[cur] + cnt);
      dis[cur] = min(dis[cur], cnt);
    }

    if (i) cout << ans << " \n"[i == n - 1];
  }
}

int main() {
  CaseT
    solve();
}
```

191039357	Jan/29/2023 08:32^{UTC+8}	Hytidel	F - Timofey and Black-White Tree	GNU C++17 (64)	Accepted	186 ms	32100 KB

14.3 拓扑排序

若有向图中的一点的序列满足:对图中的每条有向边(x, y),x都出现在y之前,则称该序列为拓扑序列.在拓扑序列中,所有的边都从前指向后.一个DAG的拓扑序列不唯一.

显然有环图的所有节点入度都非零,故无拓扑序列.可以证明:DAG至少存在一个入度为零的点,则DAG必存在拓扑序列,故DAG也称为拓扑图.理由:反证,若DAG的所有节点入度都非零,设DAG有n个节点.可从入度非零的节点找到指向它的节点,指向它的节点的入度也非零,故可找到指向它的节点,重复该过程.因所有节点的入度都非零,故可一直找下去.考察找到第$(n+1)$个节点的情况,因只有n个节点,由抽屉原理知:路径中至少有两点相同,即路径成环,矛盾.

14.3.1 有向图的拓扑序列

题意

给定一个包含编号 $1 \sim n$ 的 n $(1 \leq n \leq 1e5)$ 个节点和 m $(1 \leq m \leq 1e5)$ 条边的有向图, 图中可能有重边和自环. 输出任一个该有向图的拓扑序列, 若不存在则输出 -1 .

思路

所有入度为零的点都可作为拓扑序列的起点, 则先将所有入度为零的点入队, 再枚举各点的出边. 若所有点都进入队列, 则图是DAG, 存在拓扑序列, 即入队顺序; 否则不存在拓扑序列.

因要输出入队顺序, 故用数组模拟队列.

代码

```cpp
struct TopologicalSorting {
    int n;
    vector<vector<int>> edges;
    vector<int> d;
    vector<int> que;  // 存图的拓扑序列(若存在)

    TopologicalSorting(int _n, const vector<vector<int>>& _edges) :n(_n), edges(_edges) {
        d.resize(n + 1), que.resize(n + 1);
        for (int i = 1; i <= n; i++)
            for (auto v : edges[i]) d[v]++;
    }

    bool topo() {  // 返回图是否存在拓扑序列
        int hh = 0, tt = -1;
        for (int i = 1; i <= n; i++)
            if (!d[i]) que[++tt] = i;

        while (hh <= tt) {
            auto u = que[hh++];

            for (auto v : edges[u])
                if (--d[v] == 0) que[++tt] = v;
        }
        return tt == n - 1;
    }
};

void solve() {
    int n, m; cin >> n >> m;
    vector<vector<int>> edges(n + 1);
    while (m--) {
        int u, v; cin >> u >> v;
        edges[u].push_back(v);
    }
```

```
35
36      TopologicalSorting solver(n, edges);
37      if (solver.topo()) {
38          auto que = solver.que;
39          for (int i = 0; i < n; i++)
40              cout << que[i] << " \n"[i == n - 1];
41      }
42      else cout << -1 << endl;
43  }
44
45  int main() {
46      solve();
47  }
```

14.3.2 家谱树

题意

一个家族有n $(1 \leq n \leq 100)$个人,编号$1 \sim n$.输入n行描述每个人的孩子,其中第$(i+1)$行表示第i个人的孩子,输入0表示该行结束.输出一个序列,使得每个人的孩子比他晚输出.数据保证一定有解.

思路

将每个人视为节点,显然为DAG,要求的即拓扑序列.

代码

```
1   const int MAXN = 105, MAXM = MAXN * MAXN / 2;
2   int n;  // 节点数
3   int head[MAXN], edge[MAXM], nxt[MAXM], idx = 0;
4   int que[MAXN];
5   int in[MAXN];  // 每个节点的入度
6
7   void add(int a, int b) {
8     edge[idx] = b, nxt[idx] = head[a], head[a] = idx++;
9   }
10
11  void topo_sort() {
12    int hh = 0, tt = -1;
13    for (int i = 1; i <= n; i++)
14      if (!in[i]) que[++tt] = i;
15
16    while (hh <= tt) {
17      int t = que[hh++];
18      for (int i = head[t]; ~i; i = nxt[i]) {
19        int j = edge[i];
20        if (--in[j] == 0) que[++tt] = j;
21      }
22    }
23  }
24
25  int main() {
26    memset(head, -1, so(head));
27
28    cin >> n;
```

```
  for (int i = 1; i <= n; i++) {
    int tmp;
    while (cin >> tmp, tmp) {
      add(i, tmp);
      in[tmp] ++;  // 更新每个节点的入度
    }
  }

  topo_sort();

  for (int i = 0; i < n; i++) cout << que[i] << ' ';
}
```

14.3.3 奖金

题意

给n $(1 \leq n \leq 1e4)$个员工发工资,满足m $(1 \leq m \leq 2e4)$个要求,每个要求用一行输入描述,每行包含两个整数a, b,表示a号员工奖金多于b号员工.每位员工奖金至少为100元且为整数.求满足m个要求的最少总奖金,若无合理方案,输出"Poor Xed".

思路

因边权都为$1 > 0$,则可用拓扑排序求解,存在环时无解.

求最小值用最长路.

代码

```
const int MAXN = 1e4 + 5, MAXM = 2e4 + 5;
int n, m;  // 节点数、不等式数
int head[MAXN], edge[MAXM], nxt[MAXM], idx = 0;
int que[MAXN];
int in[MAXN];  // 每个节点的入度
int dis[MAXN];  // 从源点出发到每个节点的最长路

void add(int a, int b) {
  edge[idx] = b, nxt[idx] = head[a], head[a] = idx++;
}

bool topo_sort() {
  int hh = 0, tt = -1;
  for (int i = 1; i <= n; i++)
    if (!in[i]) que[++tt] = i;

  while (hh <= tt) {
    int t = que[hh++];
    for (int i = head[t]; ~i; i = nxt[i]) {
      int j = edge[i];
      if (--in[j] == 0) que[++tt] = j;
    }
  }

  return tt == n - 1;  // tt!=n-1时有环
}
```

```cpp

int main() {
  memset(head, -1, so(head));

  cin >> n >> m;
  while (m--) {
    int a, b; cin >> a >> b;
    add(b, a);  // 注意b→a
    in[a]++;
  }

  if (!topo_sort()) cout << "Poor Xed";  // 有正环,无解
  else {
    for (int i = 1; i <= n; i++) dis[i] = 100;  // 初始化为最低奖金

    // 求最长路
    for (int i = 0; i < n; i++) {
      int j = que[i];
      for (int k = head[j]; ~k; k = nxt[k])
        dis[edge[k]] = max(dis[edge[k]], dis[j] + 1);
    }

    int ans = 0;
    for (int i = 1; i <= n; i++) ans += dis[i];
    cout << ans;
  }
}
```

14.3.4 可达性统计

题意

给定一个含n $(1 \le n \le 3e4)$个节点、m $(1 \le m \le 3e4)$条边的DAG,节点编号$1 \sim n$,输出n行,第i行表示与节点i连通的节点数.

DAG图用m行输入描述,每行包含两个整数x, y,表示有一条从节点x到节点y的有向边.

思路

$dp[i]$表示从节点i出发能到达的节点的集合.设节点i能到达节点j_k $(k = 1, 2, \cdots)$,则$dis[i] = \{i\} \bigcup\limits_{k} dp[j_k]$,只需先拓扑排序后按拓扑序的逆序递推即可.

用bool数组记录每个点能到达的节点,总时间复杂度$O(nm)$,最坏$9e8$,会T.用bitset维护一个长度为n的二进制数表示节点i能到达的节点,1表示能到达对应的节点,0表示不能到达对应的节点,总时间复杂度$O\left(\frac{nm}{32}\right)$,最坏约$3e7$,可过.

代码

```cpp
const int MAXN = 3e4 + 5, MAXM = 3e4 + 5;
int n, m;  // 节点数、不等式数
int head[MAXN], edge[MAXM], nxt[MAXM], idx = 0;
int que[MAXN];
int in[MAXN];  // 每个节点的入度
bitset<MAXN> dp[MAXN];  // dp[u]表示从节点u出发能到达的点的状态

```

```cpp
void add(int a, int b) {
  edge[idx] = b, nxt[idx] = head[a], head[a] = idx++;
}

void topo_sort() {
  int hh = 0, tt = -1;
  for (int i = 1; i <= n; i++)
    if (!in[i]) que[++tt] = i;

  while (hh <= tt) {
    int t = que[hh++];
    for (int i = head[t]; ~i; i = nxt[i]) {
      int j = edge[i];
      if (--in[j] == 0) que[++tt] = j;
    }
  }
}

int main() {
  memset(head, -1, so(head));

  cin >> n >> m;
  for (int i = 0; i < m; i++) {
    int a, b; cin >> a >> b;
    add(a, b);
    in[b]++;
  }

  topo_sort();

  for (int i = n - 1; i >= 0; i--) {
    int j = que[i];
    dp[j][j] = 1;  // 能到达自己
    for (int k = head[j]; ~k; k = nxt[k])
      dp[j] |= dp[edge[k]];
  }

  for (int i = 1; i <= n; i++) cout << dp[i].count() << endl;
}
```

14.3.5 车站分级

题意

一条单向铁路上有编号$1 \sim n$的n $(1 \leq n \leq 1000)$个火车站,每个火车站有一个级别,最低1级.现有若干趟车次在该线路上形式,每一趟满足要求:若该趟车次停靠了火车站x,则始发站、终点站间所有级别$\geq x$的火车站都要停靠.起始站和终点站是事先已知要停靠的站点.

现有m $(1 \leq m \leq 1000)$躺车次的运行情况,满足要求,求这n个火车站至少分为几个不同级别.

运行情况用m行输入描述,第$(i+1)$行先输入一个整数s_i $(2 \leq s_i \leq n)$,表示第i躺车次有s_i个停靠站;接下来输入s_i个正整数,表示所有停靠站的编号,升序排列.输入保证所有车次都满足要求.

思路

一趟车次停靠的车站的等级严格大于其沿线未停靠的车站的等级,相当于给定$a > b$的关系,则给定n个节点相当于给定$O(n^2)$条边,因边权都为正,可用拓扑求出最小值.

每趟车次最坏从$1 \to 1000$,停靠其中500个站,则需建$500 \times (1000 - 500) = 2.5e5$条边,最多$1000$躺车次,则最多需建$2.5e8$条边,会MLE,即使用邻接表存,遍历一遍也会TLE.

将点分为左右两个集合,左边有n个节点,右边有m个节点.现要在两集合间连边,可建立一个虚拟节点u,让左边的边连向u,再让u连向右边的节点,时间复杂度从$O(nm)$降到$O(n + m)$.

设当前停靠的车站是左边集合,未停靠的车站是右边集合,则两集合间有明确分界线,分界线可取左边车站的等级的最大值,则右边车站的等级$\geq$最大值$+1$.建立虚拟节点u,左边节点向u连一条长度为0的边,u向右边节点连一条长度为1的边.

代码

```cpp
const int MAXN = 2005, MAXM = 1e6 + 5;
int n, m;  // 节点数、边数
int head[MAXN], edge[MAXM], w[MAXM], nxt[MAXM], idx = 0;
int que[MAXN];
int in[MAXN];  // 每个节点的入度
int dis[MAXN];  // 每个车站的最小等级
bool vis[MAXN];  // 每个车站是否停靠

void add(int a, int b, int c) {
  edge[idx] = b, w[idx] = c, nxt[idx] = head[a], head[a] = idx++;
  in[b]++;
}

void topo_sort() {
  int hh = 0, tt = -1;
  for (int i = 1; i <= n + m; i++)
    if (!in[i]) que[++tt] = i;

  while (hh <= tt) {
    int t = que[hh++];
    for (int i = head[t]; ~i; i = nxt[i]) {
      int j = edge[i];
      if (--in[j] == 0) que[++tt] = j;
    }
  }
}

int main() {
  memset(head, -1, so(head));

  cin >> n >> m;
  for (int i = 1; i <= m; i++) {
    memset(vis, 0, so(vis));
    int t; cin >> t;  // 停靠的车站数
    int start = n, end = 1;
    while (t--) {
      int stop; cin >> stop;
      start = min(start, stop), end = max(end, stop);
      vis[stop] = true;  // 记录停靠站点
    }
```

```
41
42      int vir = n + i;  // 虚拟节点
43      for (int j = start; j <= end; j++) {
44        if (!vis[j]) add(j, vir, 0);  // 左边节点向虚拟节点连一条长度为0的边
45        else add(vir, j, 1);  // 虚拟节点向右边节点连一条长度为1的边
46      }
47    }
48
49    topo_sort();
50
51    for (int i = 1; i <= n; i++) dis[i] = 1;  // 初始化等级为1,相当于建立一个超级源点
52    for (int i = 0; i < n + m; i++) {
53      int j = que[i];
54      for (int k = head[j]; ~k; k = nxt[k])
55        dis[edge[k]] = max(dis[edge[k]], dis[j] + w[k]);
56    }
57
58    int ans = 0;
59    for (int i = 1; i <= n; i++) ans = max(ans, dis[i]);
60    cout << ans;
61  }
```

14.3.6 发现环

原题指路: https://www.lanqiao.cn/problems/108/learning/

题意

在包含编号 $1 \sim n$ 的 n $(1 \leq n \leq 1e5)$ 个节点的树中的一对不相邻的节点间加一条边, 构成一棵基环树(或称"环套树"). 求环上的节点.

思路

注意本题是无向图.

拓扑排序, 每次将入度为 1 的节点入队, 结束后入度 ≥ 2 的节点在环中.

代码

```
1  struct TopologicalSorting {
2      int n;
3      vector<vector<int>> edges;
4      vector<int> d;
5      queue<int> que;
6
7      TopologicalSorting(int _n, const vector<vector<int>>& _edges) :n(_n), edges(_edges) {
8          d.resize(n + 1);
9          for (int u = 1; u <= n; u++)
10             for (auto v : edges[u]) d[v]++;
11     }
12
13     void topo() {
14         for (int u = 1; u <= n; u++)
15             if (d[u] == 1) que.push(u);
```

```
16
17          while (que.size()) {
18              auto u = que.front(); que.pop();
19              for (auto v : edges[u])
20                  if (d[v] && --d[v] == 1) que.push(v);
21          }
22      }
23  };
24
25  void solve() {
26      int n; cin >> n;
27      vector<vector<int>> edges(n + 1);
28      for (int i = 0; i < n; i++) {
29          int u, v; cin >> u >> v;
30          edges[u].push_back(v), edges[v].push_back(u);
31      }
32
33      TopologicalSorting solver(n, edges);
34      solver.topo();
35
36      auto d = solver.d;
37      for (int u = 1; u <= n; u++)
38          if (d[u] > 1) cout << u << ' ';
39      cout << endl;
40  }
41
42  int main() {
43      solve();
44  }
```

设有向图有n个节点,m条边.若$m \sim n^2$,则为稠密图;若$m \sim n$,则为稀疏图.

最短路问题一般分为两类:

1.单源最短路:求一个点到其他所有点的最短距离,如求1号点到n号点的最短路.

(1)边权都为正:

①朴素Dijkstra算法,时间复杂度$O(n^2)$,与边数无关,故适合用于稠密图.$n \geq 1e5$且时间限制$1\ s$,会T.

②堆优化Dijkstra算法,时间复杂度$O(m \log n)$,适合用于稀疏图.

(2)存在负权边:

①Bellman-Ford算法,时间复杂度$O(nm)$.可以处理经过的边数有限制的最短路.

②SPFA算法,是对Bellman-Ford算法的优化,效率更好,平均时间复杂度$O(m)$,最坏$O(nm)$.不可处理经过的边数有限制的最短路.SPFA经常被卡.

2.多源汇最短路:源指起点,汇指终点,多源汇最短路是起点和终点不确定的最短路问题.常用算法是Floyd算法,时间复杂度$O(n^3)$.

最短路问题侧重于建图,即将实际问题通过定义节点和边抽象成图,再套用相应的最短路算法.

14.4 Dijkstra算法

14.4.1 Dijkstra算法求最短路(边权为正)

题意

给定一个有n $(1 \leq n \leq 500)$个点和m $(1 \leq m \leq 1e5)$条边的有向图,点编号$1 \sim n$,图中可能有重边和自环,所有边权都为正.输出1号点到n号点的最短距离,若无法从1号点走到n号点,则输出-1.

有向图用m行输入描述,每个输入包含三个整数x, y, z,表示存在一条从节点x到y的边权为z $(0 < z \leq 1e4)$的有向边.

思路

本题最多有500个节点,1e5条边,则是稠密图,用邻接矩阵存.

开一个$dis[]$数组记录每个节点到起点的距离.初始时$dis[1] = 0, dis[i] = INF$ $(i \neq 1)$.

用bool数组$state[]$记录当前每个节点是否已确定最短距离.

循环所有节点,每次贪心一个不在$state[]$中的距离最近的节点t,将t加入$state[]$,用t更新其他点的距离,即看t的出边能否更新其他点的距离.设节点t走到节点x,若从起点到x的距离大于先从起点走到t,再从t走到x的距离,则更新$dis[x] = dis[t] + w[t][x]$,其中$w[t][x]$为边$t \to x$的边权.

外层循环所有节点n次,内层中贪心一个节点n次,更新其他点到起点的距离,即遍历所有边,共m次,总时间复杂度$O(n^2)$.

若自环的边权为正,则显然它不是最短路径.若有重边,则保留一条边权小的.

代码

```cpp
const int INF = 0x3f3f3f3f;
const int MAXN = 505;
int n, m;  // 节点数和边数
int w[MAXN][MAXN];  // 邻接矩阵存边权
int dis[MAXN];  // 存每个节点到起点的距离
bool state[MAXN];  // 记录当前每个节点是否已确定最短路径

int dijkstra() {
  // 初始化
  memset(dis, INF, sizeof(dis));
  dis[1] = 0;

  for (int i = 0; i < n - 1; i++) {  // 枚举每个节点
    int t = -1, j;
    for (j = 1; j <= n; j++) {  // 枚举不在state中且距离最近的点t
      if (!state[j] && (t == -1 || dis[t] > dis[j]))  // 注意t==-1时直接赋值
        t = j;
    }

    if (j == n) break;  // 已找到1->n的最短距离即可返回

    state[t] = true;  // t到起点的最短距离已确定

    for (j = 1; j <= n; j++)  // 用t的所有出边更新各节点到起点的最短距离
      dis[j] = min(dis[j], dis[t] + w[t][j]);
```

```
  }

  return dis[n] == INF ? -1 : dis[n];  // 节点n的距离未被更新,则1与n不连通
}

int main() {
  memset(w, INF, sizeof(w));  // 初始化邻接矩阵
  while (m--) {
    int x, y, z; cin >> x >> y >> z;
    w[x][y] = min(w[x][y], z);  // 有重边只保留一条边权最小的边
  }

  cout << dijkstra();
}

```

14.4.2 Dijkstra算法求最短路(边权非负)

题意

给定一个有n $(1 \leq n \leq 1.5e5)$个点和m $(1 \leq m \leq 1.5e5)$条边的有向图,点编号$1 \sim n$,图中可能有重边和自环,所有边权非负.输出1号点到n号点的最短距离,若无法从1号点走到n号点,则输出-1.

有向图用m行输入描述,每个输入包含三个整数x, y, z,表示存在一条从节点x到y的边权为z $(0 \leq z \leq 1e4)$的有向边.

若最短路存在,数据保证最短路长度不超过$1e9$.

思路

本题为稀疏图,用邻接表存,可不管重边.

朴素Dijkstra算法慢在贪心一个不在$state[]$中的距离最小的t,时间复杂度$O(n)$,而集合的最小值可用堆维护,时间复杂度优化为$O(1)$.用t的所有出边更新其他点与起点的距离时,因在堆中修改数的时间复杂度是$O(\log n)$,共需修改m次,故总时间复杂度$O(m \log n)$.

堆可用数组模拟,这样可保证堆中恒有n个数;也可直接用STL的优先队列,但它不支持修改,故每次更新都要插入一个元素,最后堆中可能有m个数,故时间复杂度变为$O(m \log m)$.因稀疏图中$m < n^2$,则$\log m < \log n^2 = 2 \log n$,故时间复杂度依旧可视为$O(m \log n)$,故直接用优先队列即可.

用堆维护距离的同时还需记录节点编号,故用pair<int,int>.

代码

```
const int MAXN = 1e6 + 5;
int n, m;  // 节点数和边数
int head[MAXN], val[MAXN], nxt[MAXN], idx = 0;  // 邻接表
int w[MAXN];  // 存边权
int dis[MAXN];  // 存每个节点到起点的距离
bool state[MAXN];  // 记录当前每个节点是否已确定最短路径

void add(int a, int b, int c) {  // 加一条a->b的边权为c的边
  val[idx] = b, nxt[idx] = head[a], head[a] = idx;
  w[idx++] = c;
```

```cpp
}

int dijkstra() {
  priority_queue<pii, vector<pii>, greater<pii>> heap;  // 小根堆

  // 初始化
  memset(dis, INF, sizeof(dis));
  dis[1] = 0;
  heap.push({ 0,1 });

  while (!heap.empty()) {
    auto t = heap.top(); heap.pop();

    int distance = t.first, ver = t.second;  // 节点离起点的距离和节点的编号

    if (state[ver]) continue;  // 若已更新,则忽略
    state[ver] = true;

    for (int i = head[ver]; ~i; i = nxt[i]) {
      int j = val[i];  // 节点ver的出边
      if (dis[j] > distance + w[i]) {  // 用节点ver的出边更新其他点到起点的距离
        dis[j] = distance + w[i];
        heap.push({ dis[j],j });
      }
    }
  }

  return dis[n] == INF ? -1 : dis[n];
}

int main() {
  memset(head, -1, sizeof(head));  // 初始化

  cin >> n >> m;
  while (m--) {
    int x, y, z; cin >> x >> y >> z;
    add(x, y, z);  // 不用处理重边

  }

  cout << dijkstra();
}

```

14.4.3 堆优化Dijkstra算法 (记录路径)

原题指路: https://codeforces.com/problemset/problem/20/C

题意

给定一张包含编号 $1 \sim n$ 的 n $(2 \leq n \leq 1e5)$ 个节点和编号 $1 \sim m$ 的 m $(0 \leq m \leq 1e5)$ 条边的有权无向图, 其中 i $(1 \leq i \leq m)$ 号边的边权 $w_i \in [1, 1e6]$. 求节点 1 到节点 n 的最短路, 输出任一路径. 若无解, 输出 -1 .

思路

因 n 最大为 $1e5$, 故用堆优化的Dijkstra算法求最短路. $pre[u]$ 表示最短路中节点 u 的前驱节点, 它可在求最短路的过程中求出. 从当前节点 $cur = n$ 开始, 沿 $cur = pre[cur]$ 回跳, 直至回跳至节点 1 , 再将路径倒序输出即可.

代码

```cpp
struct DijkstraWithHeap {
    typedef pair<int, int> pii;
    typedef pair<ll, int> pli;
    const ll INFF = 0x3f3f3f3f3f3f3f3f;

    int n;
    vector<vector<pii>> edges;
    priority_queue<pli, vector<pli>, greater<pli>> heap;
    vector<ll> dis;
    vector<bool> state;
    vector<int> pre;  // 前驱节点

    DijkstraWithHeap(int _n, const vector<vector<pii>>& _edges) :n(_n), edges(_edges) {
        pre.resize(n + 1);
    }

    void init() {
        dis = vector<ll>(n + 1, INFF);
        state = vector<bool>(n + 1, false);
        heap = priority_queue<pli, vector<pli>, greater<pli>>();
    }

    ll dijkstra(int s) {
        init();

        dis[s] = 0;
        heap.push({ 0, s });

        while (heap.size()) {
            auto [_, u] = heap.top(); heap.pop();
            if (state[u]) continue;

            state[u] = true;
            for (auto [v, w] : edges[u]) {
                if (dis[v] > dis[u] + w) {
                    dis[v] = dis[u] + w;
                    heap.push({ dis[v], v });
                    pre[v] = u;
                }
            }
        }
        return dis[n] == INFF ? -1 : dis[n];
    }
};

void solve() {
    int n, m; cin >> n >> m;
    vector<vector<pair<int, int>>> edges(n + 1);
    while (m--) {
        int u, v, w; cin >> u >> v >> w;
```

```
51          edges[u].push_back({ v, w }), edges[v].push_back({ u, w });
52      }
53
54      DijkstraWithHeap solver(n, edges);
55      auto dis = solver.dijkstra(1);
56      if (~dis) {
57          vector<int> ans;
58          auto pre = solver.pre;
59          int cur = n;
60          while (true) {
61              ans.push_back(cur);
62              cur = pre[cur];
63
64              if (cur == 1) {
65                  ans.push_back(cur);
66                  break;
67              }
68          }
69
70          reverse(all(ans));
71          for (auto u : ans)
72              cout << u << " \n"[u == ans.back()];
73      }
74      else cout << -1 << endl;
75  }
76
77  int main() {
78      solve();
79  }
```

14.4.4 Edge Deletion

原题指路: https://codeforces.com/problemset/problem/1076/D

题意 (2.5 s)

给定一张包含编号$1 \sim n$的n $(2 \le n \le 3\text{e}5)$个节点和m $(n-1 \le m \le 3\text{e}5)$条边的无向连通图, 边权$w$满足$1 \le w \le 1\text{e}9$. 设节点1到节点$i$ $(1 \le i \le n)$的最短路为d_i. 给定一个整数k $(0 \le k \le m)$. 现要删除图中的一些边, 使得至多剩下k条边. 称一个节点i是好的, 如果删除边后仍存在一条从节点1到节点i的长度为d_i的路径. 求好的节点的个数的最大值, 输出方案.

思路

显然保留k条边至多不改变$(k+1)$个节点的最短路, 进而好的节点数 $\le k+1$.

下面构造一种使得好的节点数为$(k+1)$的方案. 以节点1为起点做Dijkstra算法, 当确定$(k+1)$个节点的最短路时返回, 此时恰用到k条边, 保留这k条边即可. 事实上, 这k条边和$(k+1)$个节点构成一棵最短路树.

实现时, 用数组$last[]$记录更新节点的最短距离的上一条边的编号即可.

代码

```cpp
const ll INFF = 0x3f3f3f3f3f3f3f3f;

typedef pair<ll, int> pli;

struct Dijkstra {
    int n;
    vector<vector<tuple<int, int, int>>> edges;  // v, w, id
    vector<bool> state;
    vector<ll> dis;
    vector<int> last;  // last[u]表示更新dis[u]的上一条边的编号

    Dijkstra(int _n, const vector<vector<tuple<int, int, int>>>& _edges) :n(_n), edges(_edges) {}

    vector<int> dijkstra(int s, int k) {
        vector<int> res;
        state = vector<bool>(n + 1);
        dis = vector<ll>(n + 1, INFF);
        last = vector<int>(n + 1);

        priority_queue<pli, vector<pli>, greater<pli>> heap;
        heap.push({ 0, s });
        dis[s] = 0;

        int cnt = 0;
        while (heap.size()) {
            auto [d, u] = heap.top(); heap.pop();
            if (state[u]) continue;

            state[u] = true;
            if (last[u]) res.push_back(last[u]);

            if (++cnt == k + 1) return res;

            for (auto [v, w, id] : edges[u]) {
                if (dis[v] > dis[u] + w) {
                    dis[v] = dis[u] + w;
                    heap.push({ dis[v], v });
                    last[v] = id;
                }
            }
        }
        return res;
    }
};

void solve() {
    int n, m, k; cin >> n >> m >> k;
    vector<vector<tuple<int, int, int>>> edges(n + 1);  // v, w, id
    for (int i = 1; i <= m; i++) {
        int u, v, w; cin >> u >> v >> w;
        edges[u].push_back({ v, w, i }), edges[v].push_back({ u, w, i });
    }

    Dijkstra solver(n, edges);
```

```
55
56      auto ans = solver.dijkstra(1, k);
57      cout << ans.size() << endl;
58      for (auto ai : ans)
59          cout << ai << " \n"[ai == ans.back()];
60  }
61
62  int main() {
63      solve();
64  }
```

14.5 Bellman-Ford算法

有负权回路时,若从起点到终点的最短路径需经过负环,则最短路不存在;若无需经过负环,则最短路存在.但若限制经过的边的数量,则有负环也无所谓.

14.4.1 有边数限制的最短路

题意

给定一个有n $(1 \leq n \leq 500)$个点和m $(1 \leq m \leq 1e4)$条边的有向图,点编号$1 \sim n$,图中可能有重边和自环,边权可能为负,可能存在负权回路.输出1号点到n号点的最多经过k $(1 \leq k \leq 500)$条边最短距离,若无法从1号点走到n号点,则输出"impossible".

有向图用m行输入描述,每个输入包含三个整数x, y, z,表示存在一条从节点x到y的边权为z $(0 \leq |z| \leq 1e4)$的有向边.

思路

枚举n个节点,每次遍历所有边并更新每个节点到起点的距离,即松弛操作$dis[b] = \min(dis[b], dis[a] + w)$.因每次都需遍历所有边,故只需开一个结构体数组即可,无需邻接表.最后所有节点都满足$dis[b] \leq dis[a] + w$.

每次更新$dis[]$前需备份,否则枚举所有边时可能发生串联,可能超出允许经过的边数限制.故每次更新$dis[]$要在上一次迭代的结果基础上更新.

注意判断是否有可行解时不能$dis[n] == INF$,因为若1号点与n号点和$(n-1)$号点都不连通,但$(n-1)$号点到n号点有一条负权边,则$dis[n]$会被更新为$< INF$.因最多500条边,每条边边权最小为$-1e4$,故$dis[]$最小会被更新为$INF - 5e6$,则判断条件可为$dis[n] > INF/2$.

Bellman-Ford算法可用于判断是否有负环,但时间复杂度不如SPFA.设外层循环枚举到第k个节点,则此时的$dis[]$表示从起点出发到各个点经过不超过k条边的最短距离.若第n次遍历时又更新了$dis[]$,则存在一条最短路径,其上的边数大于等于n,则至少有$(n+1)$个节点,由抽屉原理知至少有两节点编号相同,即存在环.

代码

```
1  const int INF = 0x3f3f3f3f;
2  int n, m, k;  // 节点数、边数、最多经过的边数
3  const int MAXN = 505, MAXM = 1e4 + 5;
4  struct Edge { int u, v, w; }edges[MAXM];
5  int dis[MAXN];  // 存各节点到起点的最短距离
6  int backup[MAXN];  // 备份dis[]数组
```

```cpp

int Bellman_Ford() {
  // 初始化
  memset(dis, INF, sizeof(dis));
  dis[1] = 0;

  for (int i = 0; i < k; i++) {
    memcpy(backup, dis, sizeof(dis));  // 每次在备份的基础上修改

    for (int j = 0; j < m; j++) {  // 枚举所有出边
      int u = edges[j].u, v = edges[j].v, w = edges[j].w;
      dis[v] = min(dis[v], backup[u] + w);  // 注意用备份更新长度
    }
  }

  return dis[n] > INF / 2 ? -1 : dis[n];
}

int main() {
  for (int i = 0; i < m; i++) {
    int x, y, z; cin >> x >> y >> z;
    edges[i] = { x,y,z };
  }

  int ans = Bellman_Ford();
  if (~ans) cout << ans;
  else cout << "impossible";
}
```

14.6 SPFA

不出现负环才可用SPFA.

正权图的最短路很多也可以用SPFA过,而且比Dijistra算法快.

SPFA算法容易被网格图卡成$O(nm)$.

14.6.1 SPFA求最短路 (边权可能为负)

题意

给定一张包含编号$1 \sim n$的n $(1 \leq n \leq 1e5)$个节点和m $(1 \leq m \leq 1e5)$条边的有向图, 图中可能有重边和自环, 边权w满足$|w| \leq 1e4$, 但不存在负权回路 求节点1到节点n的最短路. 若无解, 则输出"impossible".

思路

Bellman-Ford算法中每次迭代都要枚举所有边, 但事实上并不是所有$dis[]$都会被更新. 注意到$dis[b] = \min(dis[b], dis[a] + w)$中, 若$dis[b]$减小, 则必为$dis[a]$减小. SPFA对此用BFS优化, 用一个队列存所有减小的$dis[a]$, 初始时起点入队. 每次循环取出队头t, 用t更新其所有出边, 若t减小, 则所有以t为起点的边的$dis[]$都可能减小. 若该节点不在队列中, 将其加入队列. 思想: 用更新过的节点来更新其他节点, 没有被更新过的节点忽略, 故队列中存待用于更新其他节点的节点.

用一个bool数组$state[]$记录每个点是否在队列中, 防止队列中存重复的点.

代码

```cpp
const ll INFF = 0x3f3f3f3f3f3f3f3f;
struct SPFA {
    int n;
    vector<vector<pair<int, int>>> edges;
    vector<bool> state;
    vector<ll> dis;

    SPFA(int _n, const vector<vector<pair<int, int>>>& _edges) :n(_n), edges(_edges) {}

    void spfa(int s) {
        state = vector<bool>(n + 1, false);
        dis = vector<ll>(n + 1, INFF);

        queue<int> que;
        que.push(s);
        dis[s] = 0, state[s] = true;

        while (que.size()) {
            auto u = que.front(); que.pop();
            state[u] = false;

            for (auto [v, w] : edges[u]) {
                if (dis[v] > dis[u] + w) {
                    dis[v] = dis[u] + w;

                    if (!state[v]) {
                        que.push(v);
                        state[v] = true;
                    }
                }
            }
        }
    }
};

void solve() {
    int n, m; cin >> n >> m;
    vector<vector<pair<int, int>>> edges(n + 1);
    while (m--) {
        int u, v, w; cin >> u >> v >> w;
        edges[u].push_back({ v, w });
    }

    SPFA solver(n, edges);
    solver.spfa(1);
    auto ans = solver.dis[n];
    cout << (ans == INFF ? "impossible" : to_string(ans)) << endl;
}

int main() {
    solve();
}
```

14.6.2 SPFA判断负环

题意

给定一张包含编号$1 \sim n$的n $(1 \leq n \leq 2000)$个节点和m $(1 \leq m \leq 1e4)$条边的有向图, 边权w满足$|w| \leq 1e4$. 判断图中是否出现负权回路, 若是则输出"Yes", 否则输出"No".

思路

SPFA判断负环的复杂度较高, 故一般数据范围不大.

$dis[x]$表示当前1号点到x号点的最短距离, 再维护一个数组$cnt[x]$表示当前最短路的边数, 在更新$dis[]$时顺便更新$cnt[]$. 需要更新$dis[x]$表明从1号点先到t号点, 再到x号点的距离比当前的最短路小, 故更新$cnt[x] = cnt[t] + 1$, 其中$cnt[t]$是从1号点到t号点的边数.

$cnt[x] \geq n$表明: 从1号点到x号点至少经过n条边, 则至少经过$(n + 1)$个节点 ,由抽屉原理知: 路径包含重复节点, 即有环. 而经过该环时最短路长度减小, 故该环的权值为负.

因无需求最短路长度, 故无需初始化$dis[]$数组.

题目要求判断是否存在负环, 而不是判断含有节点1的负环, 故初始时将所有节点入队.

代码

```cpp
const ll INFF = 0x3f3f3f3f3f3f3f3f;
struct SPFA {
    int n;
    vector<vector<pair<int, int>>> edges;
    vector<bool> state;
    vector<ll> dis;
    vector<int> cnt;  // 起点到节点的最短路经过的边数

    SPFA(int _n, const vector<vector<pair<int, int>>>& _edges) :n(_n), edges(_edges) {}

    bool spfa() {  // 返回图中是否有负环
        state = vector<bool>(n + 1, false);
        dis = vector<ll>(n + 1, INFF);
        cnt = vector<int>(n + 1);

        queue<int> que;

        // 初始时所有节点入队
        for (int i = 1; i <= n; i++) {
            que.push(i);
            state[i] = true;
        }

        while (que.size()) {
            auto u = que.front(); que.pop();
            state[u] = false;

            for (auto [v, w] : edges[u]) {
                if (dis[v] > dis[u] + w) {
                    dis[v] = dis[u] + w;
                    cnt[v] = cnt[u] + 1;

                    if (cnt[v] >= n) return true;

```

```
                    if (!state[v]) {
                        que.push(v);
                        state[v] = true;
                    }
                }
            }
        }
        return false;
    }
};

void solve() {
    int n, m; cin >> n >> m;
    vector<vector<pair<int, int>>> edges(n + 1);
    while (m--) {
        int u, v, w; cin >> u >> v >> w;
        edges[u].push_back({ v, w });
    }

    SPFA solver(n, edges);
    cout << (solver.spfa() ? "Yes" : "No") << endl;
}

int main() {
    solve();
}
```

14.7 Floyd算法

14.7.1 Floyd算法求最短路

题意

给定一张包含编号$1 \sim n$的n个节点和m $(1 \leq m \leq 2e5)$条边的有向图, 图中可能有重边和自环, 边权w满足$|w| \leq 1e4$, 但不存在负权回路. 现有q $(1 \leq q \leq n^2)$个询问, 每个询问包含两个整数x和y, 输出x号点到y号点的最短距离. 若路径不存在, 输出"impossible".

思路

用邻接矩阵$dis[][]$存所有边. 重边只需保留一条边权最小的即可.

$dp[k][i][j]$表示从i号点出发, 只经过$1 \sim k$号点到达j号点的最短距离.

状态转移方程$dp[k][i][j] = dp[k-1][i][k] + dp[k-1][k][j]$, 即只经过$1 \sim (k-1)$号点, 先从$i$先到$k$, 再从$k$到$j$. 第一维可去掉, 循环时先循环$k$.

三重循环, 每次用三角不等式更新, 即$dis[i][j] = \min(dis[i][j], dis[i][k] + dis[k][j])$, 最终$dis[i][j]$是$i$号点到$j$号点的最短距离.

代码

```
const int INF = 0x3f3f3f3f;
const int MAXN = 205;
int n;  // 节点数
int dis[MAXN][MAXN];  // 邻接矩阵存边,同时作为dp数组
```

```cpp

void Floyd() {
  for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++)
        dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
    }
  }
}

int main() {
  // 初始化dis[][]数组
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      if (i == j) dis[i][j] = 0;
      else dis[i][j] = INF;
    }
  }

  while (m--) {
    int x, y, z; cin >> x >> y >> z;
    dis[x][y] = min(dis[x][y], z);
  }

  Floyd();

  while (q--) {
    int x, y; cin >> x >> y;
    if (dis[x][y] > INF / 2) cout << "impossible" << endl;
    else cout << dis[x][y] << endl;
  }
}

```

对带权无向图$G=(V,E)$,由V中的全部n个顶点和E中的$(n-1)$条边构成的无向连通子图称为G的一棵生成树,其中边权和最小的生成树称为G的最小生成树.当且仅当所有顶点都连通时才存在生成树.

无向图的最小生成树常用算法:

1.Prim算法

(1)稀疏图一般用朴素Prim算法,时间复杂度$O(n^2)$.

(2)稠密图一般用堆优化Prim算法,优化思路与堆优化Dijkstra相同,时间复杂度$O(m\log n)$.(一般用不到,会用Kruskal算法代替)

2.Kruskal算法,时间复杂度$O(m\log m)$.

14.8 Prim算法

14.8.1 Prim算法求最小生成树

题意

给定一张包含编号$1 \sim n$的n $(1 \leq n \leq 500)$个节点和m $(1 \leq m \leq 1e5)$条边的无向图, 图中可能存在重边和自环, 边权w满足$|w| \leq 1e4$. 求该图的最小生成树的边权之和, 若不存在最小生成树则输出"impossible".

思路

稠密图, 用邻接矩阵存图, 重边只保留一条长度最小的边.

先初始化距离$dis[]$为INF. 用bool数组$state[]$记录点当前是否已在连通块中. 迭代n次,每次取一个不在$state[]$中且距离最小的点t, 用t更新其他点到连通块的距离, 注意更新前要先更新连通块的边权和(否则会将负权环加入生成树中来减小边权和), 再将t加入$state[]$. 点到连通块的距离定义为: 该点到连通块中任一点的边中边权的最小值. 若该点与连通块不连通, 则距离定义为INF.

同Dijkstra算法, 总时间复杂度$O(n^2)$.

Prim算法与Dijkstra算法的不同: Dijkstra算法一开始选了一个起点, 故只需迭代$(n-1)$次; 而Prim算法一开始没选起点, 故迭代n次.

代码

```cpp
struct Prim {
    int n;
    vector<vector<int>> edges;
    vector<bool> state;
    vector<int> dis;

    Prim(int _n, const vector<vector<int>>& _edges) :n(_n), edges(_edges) {
        state.resize(n + 1);
    }

    int prim() {
        dis = vector<int>(n + 1, INF);

        int res = 0;
        for (int i = 0; i < n; i++) {  // 迭代n次
            int u = -1;  // 当前不在连通块中的离连通块最近的节点
            for (int j = 1; j <= n; j++) {
                if (!state[j] && (u == -1 || dis[u] > dis[j]))
                    u = j;
            }

            if (i && dis[u] == INF) return INF;  // 图不连通

            if (i) res += dis[u];  // 第一个节点无边
            state[u] = true;

            // 先更新res, 再更新dis[]
            for (int v = 1; v <= n; v++)
                dis[v] = min(dis[v], edges[u][v]);  // 注意不是dis[v] + edges[u][v]
        }
        return res;
    }
};
```

```cpp

void solve() {
    int n, m; cin >> n >> m;
    vector<vector<int>> edges(n + 1, vector<int>(n + 1, INF));
    while (m--) {
        int u, v, w; cin >> u >> v >> w;
        edges[u][v] = edges[v][u] = min(edges[u][v], w);
    }

    Prim solver(n, edges);
    auto ans = solver.prim();
    cout << (ans == INF ? "impossible" : to_string(ans)) << endl;
}

int main() {
    solve();
}
```

14.9 Kruskal算法

14.9.1 Kruskal算法求最小生成树

题意

给定一张包含编号$1 \sim n$的n $(1 \leq n \leq 1e5)$个节点和m $(1 \leq m \leq 2e5)$条边的无向图, 图中可能存在重边和自环, 边权w满足$|w| \leq 100$. 求该图的最小生成树的边权之和, 若不存在最小生成树, 输出"impossible".

思路

无需用邻接表、临接矩阵存图, 只需用结构体数组存边.

Kruskal算法的步骤:

①先将所有边按边权从小到大排序, 时间复杂度$O(m \log m)$, 这是Kruskal算法最慢的部分, 但sort的常数很小.

②再按边权从小到大贪心一条边(a, b, c), 若a与b不连通, 则将该边加入集合, 用并查集维护连通块. 并查集合并集合$O(1)$, 最多加m条边, 故时间复杂度$O(m)$.

③看最后连了多少条边, 若连的边数小于$(n - 1)$, 则该图不连通, 因为连通n个节点至少需$(n - 1)$条边.

代码

```cpp
const ll INFF = 0x3f3f3f3f3f3f3f3f;

struct Kruskal {
    int n, m;
    vector<tuple<int, int, int>> edges;
    vector<int> fa;

    Kruskal(int _n, int _m, const vector<tuple<int, int, int>>& _edges)
        :n(_n), m(_m), edges(_edges) {
        fa.resize(n + 1);
        iota(all(fa), 0);
        sort(all(edges));
```

```
    }

    int find(int x) {
        return x == fa[x] ? x : fa[x] = find(fa[x]);
    }

    ll kruskal() {
        ll res = 0;
        int cnt = 0;  // 当前连的边数

        for (auto [w, u, v] : edges) {
            if ((u = find(u)) != (v = find(v))) {
                fa[u] = v;
                res += w, cnt++;
            }
        }
        return cnt == n - 1 ? res : INFF;
    }
};

void solve() {
    int n, m; cin >> n >> m;
    vector<tuple<int, int, int>> edges(m);  // u, v, w
    for (auto& [w, u, v] : edges) cin >> u >> v >> w;

    Kruskal solver(n, m, edges);
    auto ans = solver.kruskal();
    cout << (ans == INFF ? "impossible" : to_string(ans)) << endl;
}

int main() {
    solve();
}
```

14.9.2 Build Roads

原题指路:https://codeforces.com/gym/103118/problem/B

题意 (2 s)

有编号$1 \sim n$的n个城市,城市i $(1 \le i \le n)$处有一个经验值为a_i的施工队.现要建造$(n-1)$条路连接这些城市,连接城市i与城市j时的花费为$\gcd(a_i, a_j)$.求连接n个城市的最小代价.

$a_1, \cdots, a_n$由如下代码产生:

```
#include <stdio.h>

int n, L, R, a[200001];
unsigned long long seed;

unsigned long long xorshift64() {
  unsigned long long x = seed;
  x ^= x << 13;
  x ^= x >> 7;
  x ^= x << 17;
  return seed = x;
}

int gen() {
  return xorshift64() % (R - L + 1) + L;
}

int main() {
  scanf("%d%d%d%llu", &n, &L, &R, &seed);
  for (int i = 1; i <= n; i++) {
    a[i] = gen();
  }
  // ...
}
```

第一行输入四个整数$n, L, R, seed$ $(2 \le n \le 2e5, 1 \le L \le R \le 2e5, 1 \le seed \le 1e18)$.

思路

显然MST,但节点数最大为$2e5$,用Prim算法会TLE;图是n阶无向完全图,边数为$\frac{n(n-1)}{2}$,最大约$2e10$,用Kruskal算法会TLE.考察Kruskal算法最多能跑几阶无向完全图.令$m \log m = 1e8$,解得$m = e^{W(1e8)} \approx 6.4e6$.令$\frac{n(n-1)}{2} = 6.4e6$,解得$n \approx 3758$,此时求出各边边权的时间复杂度为$\frac{n(n+1)}{2} \approx 7e6$.故$n \le 3758$的情况可用Kruskal算法求解.

考察$n > 3758$的情况:

(1)$L = R$时,显然答案为$(n-1)L$.

*以下的解释不严谨,甚至可能错误,但代码可以AC.

(2)$L \neq R$时,答案为$(n-1)$,因为n较大时,元素间的$\frac{n(n-1)}{2}$对gcd中至少有$(n-1)$个为1,

而数组$a[]$随机,可认为这$(n-1)$个1即构成MST的边的边权.

可以证明这样发生错误的概率足够小.

代码

```
const int MAXN = 2e5 + 5, MAXM = MAXN << 2;
int L, R;
ull seed;
int a[MAXN];

ull xorshift64() {
  ull x = seed;
  x ^= x << 13, x ^= x >> 7, x ^= x << 17;
  return seed = x;
}
```

```cpp

int gen() { return xorshift64() % (R - L + 1) + L; }

namespace Kruskal {
  int n, m;  // 节点数、边数
  struct Edge {
    int u, v;
    ll w;

    bool operator<(const Edge& B) { return w < B.w; }
  }edges[MAXM];
  int fa[MAXN];  // 并查集的fa[]数组

  void init() {  // 初始化fa[]
    for (int i = 1; i <= n; i++) fa[i] = i;
  }

  int find(int x) { return x == fa[x] ? x : fa[x] = find(fa[x]); }

  ll kruskal() {  // 返回最小生成树的最长边,图不连通时返回INFF
    sort(edges, edges + m);  // 按边权升序排列

    ll res = 0;  // 最小生成树的边权和
    int cnt = 0;  // 当前连的边数
    for (int i = 0; i < m; i++) {
      auto [u, v, w] = edges[i];
      u = find(u), v = find(v);
      if (u != v) {
        fa[u] = v;
        res += w;
        cnt++;
      }
    }

    if (cnt < n - 1) return INFF;  // 图不连通
    else return res;
  }
}
using namespace Kruskal;

void solve() {
  cin >> n >> L >> R >> seed;
  for (int i = 1; i <= n; i++) a[i] = gen();

  if (L == R) {
    cout << (ll)L * (n - 1);
    return;
  }

  if (n > 3758) {
    cout << n - 1;
    return;
  }

  for (int i = 1; i <= n; i++) {
    for (int j = i + 1; j <= n; j++)
      edges[m++] = { i,j,gcd(a[i], a[j]) }, edges[m++] = { j,i,gcd(a[i], a[j]) };
  }
```

```
69
70    init();
71    cout << kruskal();
72  }
73
74  int main() {
75    solve();
76  }
```

14.9.3 Rikka with Minimum Spanning Trees

原题指路:https://codeforces.com/gym/102012/problem/A

题意 (6 s)

```
unsigned long long k1, k2;

unsigned long long xorShift128Plus() {
    unsigned long long k3 = k1, k4 = k2;
    k1 = k4;
    k3 ^= k3 << 23;
    k2 = k3 ^ k4 ^ (k3 >> 17) ^ (k4 >> 26);
    return k2 + k4;
}

int n, m, u[100001], v[100001];
unsigned long long w[100001];

void gen() {
    scanf("%d%d%llu%llu", &n, &m, &k1, &k2);
    for(int i = 1; i <= m; ++i) {
        u[i] = xorShift128Plus() % n + 1;
        v[i] = xorShift128Plus() % n + 1;
        w[i] = xorShift128Plus();
    }
}
```

有t $(1 \leq t \leq 100)$组测试数据.每组测试数据第一行输入四个整数 n, m, k_1, k_2 $(1 \leq n \leq 1e5, m = 1e5, 1e8 \leq k_1, k_2 \leq 1e12)$.用上图所示的程序生成一个包含$n$个节点和$m$条边的无向有权图.求该图的不同MST的边权和之和,答案对$(1e9 + 7)$取模.两MST不同当且仅当它们的边集不同.

思路

设unsigned long long的最大值为$MAXW$.注意到若一个图中无边权相等的边,则该图的MST唯一,假设该随机数生成器足够随机,则图中至少出现两条边权相等的边的概率为$\frac{1}{MAXW^2}$,这是小到可以忽略的概率,则可认为该图的MST唯一,故MST的边权和即为答案.

代码

```
1   typedef unsigned long long ull;
2   const int MAXN = 1e5 + 5;
3   const int MOD = 1e9 + 7;
4   int n, m;
5   ull k1, k2;
6   int u[MAXN], v[MAXN];
7   ull w[MAXN];
8
9   ull xorShift128Plus() {
10    ull k3 = k1, k4 = k2;
```

```
  k1 = k4;
  k3 ^= k3 << 23;
  k2 = k3 ^ k4 ^ (k3 >> 17) ^ (k4 >> 26);
  return k2 + k4;
}

vector<tuple<ull, int, int>> gen() {
  cin >> n >> m >> k1 >> k2;
  vector<tuple<ull, int, int>> edges;
  for (int i = 1; i <= m; i++) {
    u[i] = xorShift128Plus() % n + 1;
    v[i] = xorShift128Plus() % n + 1;
    w[i] = xorShift128Plus();
    edges.push_back({ w[i], u[i], v[i] });
  }
  return edges;
}

struct Kruskal {
  int n;
  vector<tuple<ull, int, int>> edges;
  vector<int> fa;

  Kruskal(int _n, vector<tuple<ull, int, int>> _edges) :n(_n), edges(_edges) {
    fa.resize(n + 1);
    iota(all(fa), 0);
  }

  int find(int x) {
    return x == fa[x] ? x : fa[x] = find(fa[x]);
  }

  ull kruskal() {
    sort(all(edges));
    int cnt = 0;
    ull res = 0;
    for (auto [w, u, v] : edges) {
      u = find(u), v = find(v);
      if (u != v) {
        fa[u] = v;
        res = (res + w) % MOD;
        cnt++;
      }
    }
    return cnt == n - 1 ? res : 0;
  }
};

void solve() {
  auto edges = gen();
  cout << Kruskal(n, edges).kruskal() << endl;
}

int main() {
  CaseT
    solve();
}
```

14.9.4 Build Roads

原题指路: https://codeforces.com/gym/103118/problem/B

题意 (2 s)

思路

代码

```
1
```

14.9.5 Make It Connected

原题指路: https://codeforces.com/problemset/problem/1095/F

题意 (2 s)

给定一张包含编号 $1 \sim n$ 的 n $(1 \le n \le 2\text{e}5)$ 个节点的无向图, 其中节点 i $(1 \le i \le n)$ 的点权为 a_i $(1 \le a_i \le 1\text{e}12)$, 节点 u $(1 \le u \le n)$ 与节点 v $(1 \le v \le n)$ 间的边权为 $(a_u + a_v)$. 除了上述边外, 还有 m $(0 \le m \le 2\text{e}5)$ 条特殊边, 每条边用三个整数 x, y, w $(1 \le x, y \le n, x \ne y, 1 \le w \le 1\text{e}12)$ 描述, 表示节点 x 与节点 y 间存在一条边权为 w 的边. 求该图的MST的权值.

思路

$O(n^2)$ 建图显然不可行. 考虑如何减少图的边数.

先不考虑特殊边, 任一节点向其他所有节点连边构成的菊花图是该完全图的一棵生成树. 为使得该生成树的权值最小, 应选择任一点权最小的节点向其他所有节点连边, 故原图中的边只需考察以任一点权最小的节点为端点的 $(n - 1)$ 条边 . 再考虑特殊边能否减小生成树的边权, 只需将特殊边加入后, 求该图的MST即可.

用Kruskal算法求MST时, 时间复杂度 $O((n + m) \log (n + m))$.

代码

```
const ll INFF = 0x3f3f3f3f3f3f3f3f;

struct Kruskal {
    int n;
    vector<tuple<ll, int, int>> edges;
    vector<int> fa;

    Kruskal(int _n, const vector<tuple<ll, int, int>>& _edges) :n(_n), edges(_edges) {
        fa.resize(n + 1);
        iota(all(fa), 0);
        sort(all(edges));
    }

    int find(int x) {
```

```
        return x == fa[x] ? x : fa[x] = find(fa[x]);
    }

    ll kruskal() {
        ll res = 0;
        int cnt = 0;  // 当前连的边数

        for (auto it : edges) {
            ll w = get<0>(it);
            int u = get<1>(it), v = get<2>(it);

            if ((u = find(u)) != (v = find(v))) {
                fa[u] = v;
                res += w, cnt++;
            }
        }
        return cnt == n - 1 ? res : INFF;
    }
};

void solve() {
    int n, m; cin >> n >> m;
    vector<ll> a(n + 1);
    for (int i = 1; i <= n; i++) cin >> a[i];
    vector<tuple<ll, int, int>> edges(m);
    for (auto& [w, u, v] : edges) cin >> u >> v >> w;

    a[0] = INFF;
    int pos = min_element(all(a)) - a.begin();  // 最小点权的下标
    for (int i = 1; i <= n; i++) {
        if (i != pos)
            edges.push_back({ a[i] + a[pos], i, pos });
    }

    Kruskal solver(n, edges);
    cout << solver.kruskal() << endl;
}

int main() {
    solve();
}
```

14.10 二分图

若可将一个二分图的顶点分为两个集合,使得所有边都在集合之间,而集合内部无边,这样的图称为二分图.

一个图是二分图当且仅当图中不含奇数环(边数为奇数的环),证明:

(必要性) 若二分图中含奇数环,设奇数环的起点1号点属于左边集合,则下一个点2号点属于右边集合,3号点属于左边集合,···,最后推出1号点属于右边集合,矛盾.

(充分性) 给每个连通块中的一个点染为白色,与白色相邻的点染为黑色,则若图连通,则整个图都会被染色.因该图不存在奇数环,则染色过程不出现矛盾(下证),再按白色和黑色将顶点分为两个集合即得二分图.

染色不出现矛盾的证明:若不然,则出现了矛盾,即环的最后一个点与起点同色,此时该环的顶点数为奇数(因为在环的最后一个点与起点间加一个点即可保证染色不出现矛盾,即白色点与黑色点数量相同,此时顶点的数量为偶数,去掉该点后点的数量为奇数),故存在奇数环,矛盾.

综上,可用染色法判断一个图是否为二分图.显然连通块中一个点颜色确定,则整个连通块的颜色确定.若染色过程中出现矛盾,则存在奇数环,即不是二分图.染色过程可用DFS或BFS实现,一般用前者,因为代码短.

给定二分图G,在G的一个子图M中,若M的边集$\{E\}$中任两条边都不依附于同一顶点,则称M是一个匹配.所有匹配中含边数最多的一组匹配称为二分图的最大匹配,其边数称为最大匹配数.在匹配中的点称为匹配点,否则称为非匹配点.在匹配中的边称为匹配边,否则称为非匹配边.从一个非匹配点出发,先走非匹配边,再走匹配点,再走非匹配边,再走匹配边,$\cdots$,最后到达一个非匹配点的路径称为增广路径,简称增广路.

二分图的一个匹配是最大匹配的充要条件是:图中不存在增广路.

求二分图最大匹配的经典算法是匈牙利算法(Hungarian算法).

给定一个图,从中选出若干个节点,使得每条边的两顶点中至少有一个顶点被选中,这样选出的最小点集称为该图的最小点覆盖.

二分图的最小点覆盖$=$最大匹配数.

[**证**] (1)先证最小点覆盖$\geq$最大匹配数.

因二分图的最大匹配中任意两条边无公共点,设最大匹配数为m,则点覆盖至少要选m个节点.

(2)再证等号可成立.下面构造一种方案.设二分图的最大匹配数为m.

从二分图的最大匹配中左半边的非匹配点出发做一遍增广,标记所有经过的点,注意两半边都要标记.

下证左半边所有未被标记的点和右半边所有被标记的点的并集是该二分图的一个最小点覆盖,即证选中的点的个数为m.

显然左半边所有非匹配点都被标记,右半边所有非匹配点都未被标记(否则构成增广路,与初始时是最大匹配矛盾).

所有匹配边的两顶点要么同时被标记要么同时不被标记,这是因为左半边的匹配点的前驱都是从右半边的匹配点,

故它们要么都被遍历到,要么都不被遍历到.

综上,左半边所有未被标记的点都是匹配点,右半边所有被标记的点都是匹配点,这表明:所有选出的点都是匹配边的顶点.

①对每条匹配点,若两顶点都被标记,则选其在右半边的顶点,否则选其在左半边的顶点,

即每个匹配边选且只选一个顶点,则选中的节点数等于匹配边数m.

这同时证明了每条匹配边都被一个节点覆盖.

②对不在匹配中的边,有两种情况:

i)左半边的非匹配点连到右半边的匹配点.因以左半边的非匹配点为起点做增广,则右半边的节点无论是否为匹配点,只要能走到都会被标记,故右半边的匹配点被覆盖.

ii)左半边的匹配点连到右半边的非匹配点.

左半边的匹配点一定不被标记.若不然,则它会标记右半边的非匹配点,与右半边的非匹配点都不被标记矛盾.

故左半边的匹配点被覆盖.

③不存在一条边的两顶点都是非匹配点的情况,否则可在最大匹配中加一条边,与它是最大匹配矛盾.

给定一个图,从中选出一些节点,使得选出的节点间都不存在边,这样选出的最大的点集称为该图的最大独立集.与之互补的,给定一个图,从中选出一些节点,使得选出的节点间都存在边,这样选出的最大的点集称为改图的最大团.原图的最大独立集是其补图的最大团.

二分图的最大独立集=(两半边的)总节点数-最小点覆盖

[**证**] 从二分图中选出最多的节点使得选出的节点间都不存在边,等价于选出最少的节点来破坏所有边.

给定一个DAG,从中选出最少的互不相交(节点不重复)的路径覆盖所有节点,这样选出的路径称为该DAG的最小路径点覆盖,简称最小路径覆盖.

[**拆点**] 设原图有编号$1 \sim n$的n个节点.将每个节点i拆成两个节点,分别是出点i和入点i'.

若存在有向边$u \to v$,则也存在有向边$u \to v'$,即u的出点指向v的入点.

将$u \to v'$视为无向边,则原图中的有向边转化为从出点的集合到入点的集合的边.

这将原来的DAG转化为一个二分图,不妨设出点在左半边,入点在右半边.

DAG的最小路径覆盖=(原图的)总节点数-最大匹配数.

[**证**] 考察DAG中的每条路径转化到二分图中的路径后的性质.

因原图中路径互不相交,则每个节点都属于且只属于一条路径,进而每个节点都至多有1的入度和1的出度.

①原图中只有1的出度的节点在二分图中体现为左半边的对应节点只有一条出边.

②原图中只有1的入度的节点在二分图中体现为右半边的对应节点只有一条入边.

综上,

①原图中的一条路径对应二分图中的一个匹配,反之亦然.

②原图中每条路径的终点出度为0,对应二分图中左半边的非匹配点,反之亦然.

求原图中的最小路径覆盖转化为求二分图的左半边中非匹配点数的最小值,即原图的总节点数-二分图的最大匹配数.

给定一个DAG,从中选出最少的路径(可相交)覆盖所有节点,这样选出的路径称为该DAG的最小路径重复点覆盖,简称最小路径重复覆盖.

为求DAG的最小路径重复覆盖,先对原图G求传递闭包得到G',则G的最小路径重复覆盖是G'的最小路径覆盖.

[**证**] ①对G的每条路径,从第二条路径开始,用传递闭包跳过之前路径上出现的节点.

不会出现一条路径上的节点都被跳过,否则该路径可从G中删去,与它是G的最小路径重复覆盖矛盾.

故G的每条路径与G'的每条路径一一对应,故两者的路径数相等.

②对G'中的每条路径,将通过间接边的路径展开为原路径即得G的每条路径.

二分图中,最大匹配数=最小点覆盖=(两半边的)总节点数-最大独立集.

DAG中,最小路径覆盖=(原图的)总节点数-最大匹配数.

14.10.1 染色法判定二分图

题意

给定一张包含编号 $1 \sim n$ 的 n $(1 \leq n \leq 1e5)$ 个节点和 m $(1 \leq m \leq 1e5)$ 条边的无向图, 图中可能存在重边和自环. 判断该图是否为二分图, 若是则输出"Yes", 否则输出"No".

思路

用邻接表存图, 本题为无向图, 故边要开两倍.

开一个int数组 $color[]$ 记录每个点的染色情况, 0 表示未被染色, 1 和 2 分别表示染成白色和黑色, 则颜色取反可用 $(3-c)$ 实现. 染色过程用一个 $flag$ 记录是否发生矛盾, 染色过程中让DFS返回一个bool来判断是否出现矛盾.

遍历各节点时, 若该节点未被染色, 则给它染与当前节点相反的颜色, 并判断是否有矛盾; 若该节点已被染色, 则判断是否矛盾.

总时间复杂度 $O(n+m)$.

代码

```cpp
void solve() {
    int n, m; cin >> n >> m;
    vector<vector<int>> edges(n + 1);
    while (m--) {
        int u, v; cin >> u >> v;
        edges[u].push_back(v), edges[v].push_back(u);
    }

    vector<int> colors(n + 1);  // 记录节点染色情况, 0表示未染色, 1表示染白色, 2表示染黑色

    function<bool(int, int)> dfs = [&](int u, int c) {  // 将节点u染c色, 返回染色是否成功
        colors[u] = c;

        for (auto v : edges[u]) {
            if (!colors[v]) {
                if (!dfs(v, 3 - c)) return false;
            }
            else if (colors[v] == c) return false;
        }
        return true;
    };

    for (int u = 1; u <= n; u++) {
        if (!colors[u]) {
            if (!dfs(u, 1)) {
                cout << "No" << endl;
                return;
            }
        }
    }
```

```
    cout << "Yes" << endl;
}

int main() {
    solve();
}
```

14.10.2 二分图的最大匹配

题意

给定一张二分图, 其左半边包含编号为$1 \sim n_1$的n_1 $(1 \leq n_1 \leq 500)$个节点, 右半边包含编号为$1 \sim n_2$的n_2 $(1 \leq n_2 \leq 500)$个节点, 二分图共包含m $(1 \leq m \leq 1e5)$条边, 且任一条边的两端点不在同个部分中. 求该二分图的最大匹配数.

思路

匈牙利算法的过程:

枚举一半边(不妨设为左半边)的每个点A, 枚举与其连通的右半边的点B:

①若B未与其他左半边的点匹配, 则将A与B匹配.

②若B已与其他左半边的点C匹配, 则检查C能否换一个匹配, 若能, 则C换一个匹配, 让A与B匹配; 否则该方案不合法.

左半边的点数是$O(n)$级别的, 最坏的情况每个点遍历m条边, 故最坏的总时间复杂度$O(nm)$, 但实际运行时间远小于$O(nm)$, 甚至可能是$O(n)$, 因为几乎没有点需遍历m条边.

若枚举左半边的点, 则只存左半边指向右半边的边, 尽管这是无向图.

代码

```
void solve() {
    int n1, n2, m; cin >> n1 >> n2 >> m;
    vector<vector<int>> edges(n1 + 1);
    while (m--) {
        int u, v; cin >> u >> v;
        edges[u].push_back(v);
    }

    vector<bool> state(n2 + 1);  // 记录右半边的节点是否已匹配
    vector<int> match(n2 + 1);  // 与右半边的节点匹配的左半边的节点

    function<bool(int u)> find = [&](int u) {
        for (auto v : edges[u]) {
            if (!state[v]) {
                state[v] = true;

                if (!match[v] || find(match[v])) {
                    match[v] = u;
                    return true;
                }
            }
```

```
        }
        return false;
    };

    int ans = 0;
    for (int u = 1; u <= n1; u++) {
        fill(all(state), false);  // 清空右半边节点的匹配状态
        ans += find(u);
    }
    cout << ans << endl;
}

int main() {
    solve();
}
```

14.10.3 关押罪犯

题意

有两个监狱,关押着编号$1 \sim n$的n个罪犯.用一个怒气值表示两罪犯间的仇恨程度.若两怒气值为c的罪犯被关押在同一监狱,则会发生影响力为c的冲突事件.求一种分配方案,使得两监狱中发生的冲突事件的影响力的最大值最小,输出最小的影响力,若不会发生冲突事件,输出0.

第一行输入两个整数n, m $(1 \leq n \leq 2\text{e}4, 1 \leq m \leq 1\text{e}5)$,分别表示罪犯的个数和存在仇恨的罪犯的对数.接下来m行每行输入三个整数a, b, c $(1 \leq a < b \leq n, 1 \leq c \leq 1\text{e}9)$,表示$a$号罪犯与$b$号罪犯间存在仇恨,怒气值为$c$.数据保证每对罪犯的组合只出现一次.

思路

对整数mid,只保留权值$> mid$的边后是否是二分图,即将权值$> mid$的边的两顶点分在两个集合中.

考察该性质是否有二段性,设最优解为ans.

①$\geq ans$的值是在ans的二分图的基础上去掉一些边,显然还是二分图,满足性质.

②$< ans$的值都不满足性质,若不然,则$\exists x < ans\ s.t.\ x$对应的图是二分图,与ans是最大影响力的最小值的矛盾.

二分下界即不发生冲突事件的影响力,即$l = 0$;二分上界为冲突事件的影响力的最大值,即$r = 1\text{e}9$.

用染色法判断二分图即可,时间复杂度$O(\log c(n + 2m))$,其中边数需要2倍是因为这是无向图.

代码

```
const int MAXN = 2e4 + 5, MAXM = 2e5 + 5;
int n, m;
vii edges[MAXN];
int color[MAXN];  // 0表示未染色,1表示染白色,2表示染黑色

bool dfs(int u, int c, int mid) {  // 只考虑当前节点u的权值>mid的出边,将节点u染成c色,返回染色是否成功
  color[u] = c;
  for (auto& [v, w] : edges[u]) {
    if (w <= mid) continue;

```

```
    if (color[v]) {
      if (color[v] == c) return false;
    }
    else if (!dfs(v, 3 - c, mid)) return false;
  }
  return true;
}

bool check(int mid) {
  memset(color, 0, so(color));

  for (int i = 1; i <= n; i++)
    if (!color[i] && !dfs(i, 1, mid)) return false;
  return true;
}

void solve() {
  cin >> n >> m;
  while (m--) {
    int a, b, c; cin >> a >> b >> c;
    edges[a].push_back({ b,c }), edges[b].push_back({ a,c });
  }

  int l = 0, r = 1e9;
  while (l < r) {
    int mid = l + r >> 1;
    if (check(mid)) r = mid;
    else l = mid + 1;
  }
  cout << l;
}

int main() {
  solve();
}
```

14.10.4 棋盘覆盖

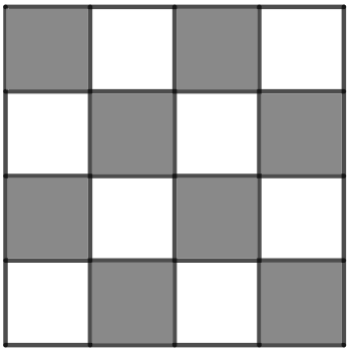
题意

给定一个$n \times n$的棋盘,有些格子禁止放置.问最多能放多少个2×1的长方形,要求长方形的边平行于棋盘的边,且任意两长方形不重叠.

第一行输入两个整数n, t $(1 \leq n \leq 100, 0 \leq t \leq 100)$,分别表示棋盘边长和禁止放置的格子数.接下来$t$行每行输入两个整数$x, y$ $(1 \leq x, y \leq n)$,表示格子(x, y)禁止放置.

思路

将棋盘的每个格子视为一个节点,若用一个2×1的矩形覆盖,则在它们间连边.问题转化为最多能连多少条边,使得任意两条边间无公共点,即求最大匹配.



将棋盘如上图染色,则显然边的顶点都是一个白格子和一个黑格子,故该图是二分图,问题转化为求二分图的最大匹配.

容易发现坐标(i,j)是黑格子的充要条件是$i+j$是奇数.

代码

```
const int MAXN = 105;
int n, m;  // 棋盘大小、被禁用的点数
bool g[MAXN][MAXN];  // 记录棋盘被禁用的点
pii match[MAXN][MAXN];  // 记录当前与右半边的点匹配的左半边的点
bool state[MAXN][MAXN];  // 记录当前右半边的点是否已匹配

bool find(int x, int y) {
  for (int i = 0; i < 4; i++) {
    int curx = x + dx[i], cury = y + dy[i];
    if (curx >= 1 && curx <= n && cury >= 1 && cury <= n && !g[curx][cury] && !state[curx][cury]) {
      state[curx][cury] = true;
      pii tmp = match[curx][cury];
      if (tmp.first == -1 || find(tmp.first, tmp.second)) {
        match[curx][cury] = { x,y };
        return true;
      }
    }
  }
  return false;
}

void solve() {
  memset(match, -1, so(match));

  cin >> n >> m;
  while (m--) {
    int x, y; cin >> x >> y;
    g[x][y] = true;
  }

  int ans = 0;
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      if ((i + j & 1) && !g[i][j]) {
        memset(state, false, so(state));
        if (find(i, j)) ans++;
      }
    }
  }
  cout << ans;
```

```
}

int main() {
  solve();
}
```

14.10.5 机器任务

题意

有A、B两台机器和编号$0 \sim (k-1)$的k个任务.机器A有编号$0 \sim (n-1)$的n种模式,机器B有编号$0 \sim (m-1)$的m种模式,初始时两机器都处于模式0.每个任务可在机器A上进行,也可在机器B上进行.对每个任务i,用两个整数a_i和b_i分别表示它在机器A、B上进行所需的模式.任务可以任意顺序执行,但每台机器转换一次模式需重启一次.求一种任务安排方案,使得机器重启次数最少,输出最少的重启次数.

有多组测试数据.每组测试数据第一行输入三个整数n, m, k $(1 \leq n, m < 100, 1 \leq k < 1000)$.接下来$n$行每行输入三个整数$i, a_i, b_i$ $(0 \leq i < k, 0 \leq a_i < n, 0 \leq b_i < m)$,分别表示任务$i$所需的模式.最后一行输入0表示输入结束.

思路

①若$a_i = 0$或$b_i = 0$,则这样的任务可在一开始完成,无需重启机器.

②对$a_i, b_i > 0$的任务,问题转化为从机器A和B的$(n + m - 2)$个模式中至少选多少个模式可将任务完成.

将任务i视为二分图中左半边的a_i号节点连向右半边的b_i号节点的边,问题转化为求二分图的最小点覆盖.

代码

```
const int MAXN = 105;
int n, m, k;  // A机器模式数、B机器模式数、任务数
bool g[MAXN][MAXN];  // 邻接矩阵
int match[MAXN];  // 记录当前与右半边的点匹配的左半边的点
bool state[MAXN];  // 记录当前右半边的点是否已匹配

bool find(int x) {
  for (int y = 0; y < m; y++) {
    if (!state[y] && g[x][y]) {
      state[y] = true;
      if (match[y] == -1 || find(match[y])) {
        match[y] = x;
        return true;
      }
    }
  }
  return false;
}

void solve() {
  while (cin >> n, n) {
    memset(g, false, so(g));
    memset(match, -1, so(match));

    cin >> m >> k;
    while (k--) {
      int t, a, b; cin >> t >> a >> b;
      if (!a || !b) continue;
```

```
      g[a][b] = true;
    }

    int ans = 0;
    for (int i = 1; i < n; i++) {
      memset(state, false, so(state));
      if (find(i)) ans++;
    }
    cout << ans << endl;
  }
}

int main() {
  solve();
}
```

14.10.6 骑士放置

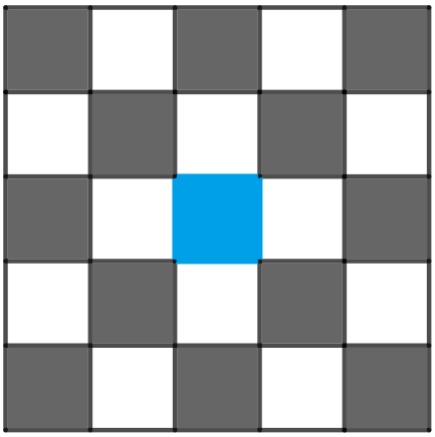
题意

给定一个$n \times m$的棋盘,有些格子禁止放置.问至多能在棋盘上放多少个不能互相攻击的骑士(类似于中国象棋的"马",以"日"字型攻击,但无"别马腿"的规则).

第一行输入三个整数n, m, t $(1 \leq n, m \leq 100)$,分别表示棋盘大小和禁止放置的格子数.接下来t行每行输入两个整数x, y $(1 \leq x \leq n, 1 \leq y \leq m)$,表示格子$(x, y)$禁止放置.

思路

将两个能互相攻击到的格子间连边,问题转化为从图中选出最多的节点使得选出的节点间都不存在边,即求该图的最大独立集.



将棋盘如上图染色,则在蓝色格子处的骑士能攻击到的格子都是白色格子,故该图是二分图,转化为求二分图的最大独立集,总点数为$nm - t-$最大匹配数.

代码

```
const int dx[8] = { -2,-1,1,2,2,1,-1,-2 }, dy[8] = { 1,2,2,1,-1,-2,-2,-1 };
const int MAXN = 105;
int n, m, t;  // 棋盘大小、禁止放置的格子数
bool g[MAXN][MAXN];  // 记录棋盘被禁用的点
pii match[MAXN][MAXN];  // 记录当前与右半边的点匹配的左半边的点
bool state[MAXN][MAXN];  // 记录当前右半边的点是否已匹配
```

```
7
8  bool find(int x, int y) {
9    for (int i = 0; i < 8; i++) {
10     int curx = x + dx[i], cury = y + dy[i];
11     if (curx >= 1 && curx <= n && cury >= 1 && cury <= m && !g[curx][cury] && !state[curx]
   [cury]) {
12       state[curx][cury] = true;
13       pii tmp = match[curx][cury];
14       if (tmp.first == -1 || find(tmp.first, tmp.second)) {
15         match[curx][cury] = { x,y };
16         return true;
17       }
18     }
19   }
20   return false;
21 }
22
23 void solve() {
24   memset(match, -1, so(match));
25
26   cin >> n >> m >> t;
27   for (int i = 0; i < t; i++) {
28     int x, y; cin >> x >> y;
29     g[x][y] = true;
30   }
31
32   int ans = 0;
33   for (int i = 1; i <= n; i++) {
34     for (int j = 1; j <= m; j++) {
35       if ((i + j & 1) || g[i][j]) continue;
36
37       memset(state, false, so(state));
38       if (find(i, j)) ans++;
39     }
40   }
41   cout << n * m - t - ans;
42 }
43
44 int main() {
45   solve();
46 }
```

14.10.7 捉迷藏

题意

给定一张包含n个节点和m条有向边的DAG.若节点u与v间存在路径,则称节点u与v能相互看见.A和B玩捉迷藏,A选出k个节点作为藏身点,要求任意两藏身点间无路径,B专门检查藏身点.为使得B难找到A,A最多可选多少个藏身点.

第一行输入两个整数n, m $(1 \leq n \leq 200, 1 \leq m \leq 3e4)$.接下来$m$行每行输入两个整数$u, v$ $(1 \leq u, v \leq n)$,表示存在一条从节点u到节点v的有向边.

思路

ans为该DAG的最小路径重复覆盖数.

[证] 设最小路径重复覆盖数为cnt.

(1)因cnt条路径能将图覆盖,则要选的k个点是路径上的点.

(2)一条路径上至多选一个点,否则在前面的点能看见在后面的点,故$ans \leq cnt$.

(3)下面构造一个$ans = cnt$的方案.

将所有路径的cnt个终点放在cnt个集合E中,下证取E中的点即为方案.

设集合E中的节点能到达的节点构成的集合为$next(E)$.

①若$E \bigcap next(E) = \varnothing$,则$E$中的节点不能相互到达,则$|E| = cnt$.

②若$E \bigcap next(E) \neq \varnothing$,对每个节点$e_i \in E$,沿以它为终点的路径回退,直至走到一个不在$next(E)$中的节点$e_i'$.

设操作后得到的节点构成的集合为E',则$E' \bigcap next(E) = \varnothing$,此时$ans = cnt$.

下证该过程可终止,且每个节点至多回退到其所在路径的起点.

若存在一条路径的起点$\in next(E)$,则该路径上的所有点都能被其他路径到达,则该路径可删去,

与它是图的最小路径重复覆盖矛盾.

代码

```cpp
const int MAXN = 205, MAXM = 3e4 + 5;
int n, m;  // 节点数、边数
bool g[MAXN][MAXN];  // 邻接矩阵
int match[MAXN];  // 记录当前与右半边的点匹配的左半边的点
bool state[MAXN];  // 记录当前右半边的点是否已匹配

void floyd() {  // Floyd算法求传递闭包
  for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++) g[i][j] |= g[i][k] & g[k][j];
  }
}

bool find(int x) {
  for (int y = 1; y <= n; y++) {
    if (g[x][y] && !state[y]) {
      state[y] = true;
      if (match[y] == -1 || find(match[y])) {
        match[y] = x;
        return true;
      }
    }
  }
  return false;
}

void solve() {
  memset(match, -1, so(match));

  cin >> n >> m;
  while (m--) {
```

```
    int a, b; cin >> a >> b;
    g[a][b] = true;
  }

  floyd();

  int ans = 0;
  for (int i = 1; i <= n; i++) {
    memset(state, false, so(state));
    if (find(i)) ans++;
  }
  cout << n - ans;
}

int main() {
  solve();
}
```

14.10.8 Graph Without Long Directed Paths

原题指路: https://codeforces.com/problemset/problem/1144/F

题意 (2 s)

给定一张包含编号$1 \sim n$的n $(2 \leq n \leq 2\text{e}5)$个节点和$m$ $(n-1 \leq m \leq 2\text{e}5)$条边的简单无向连通图. 给每条边定向, 使得图中不存在长度≥ 2的路径. 若有解, 输出方案.

思路

[**定理1**] 若有解, 则图中的节点不能同时有非零的入度和非零的出度, 即节点要么入度为0, 要么出度为0.

[**证**] 若不然, 设节点u的入度和出度都为1, 则至少存在一条经过u的长度≥ 3的路径.

[**定理2**] 将节点按入度为0、出度为0分为两类. 若有解, 则图中相邻的两节点u和v属于不同类.

[**证**] 若不然, 则连接u和v的边有两个方向.

问题转化为: 将图中的节点分为两类, 使得相邻两节点属于不同类. 用染色法判定二分图即可.

输出方案时, 对每条边(u, v), 若节点即u的入度为0, 则该边方向为$u \to v$; 否则该边方向为$u \leftarrow v$.

代码

```
void solve() {
    int n, m; cin >> n >> m;
    vector<vector<int>> edges(n + 1);
    vector<pair<int, int>> eds;
    while (m--) {
        int u, v; cin >> u >> v;

        edges[u].push_back(v), edges[v].push_back(u);
        eds.push_back({ u, v });
    }
```

```
11
12      vector<int> colors(n + 1);  // 节点的颜色: 0表示未染色, 1表示染白色, 2表示染黑色
13
14      function<bool(int, int)> dfs = [&](int u, int c) {  // 给节点u染c色, 返回染色是否成功
15          colors[u] = c;
16
17          for (auto v : edges[u]) {
18              if (!colors[v]) {
19                  if (!dfs(v, 3 - c)) return false;
20              }
21              else if (colors[v] == c) return false;
22          }
23          return true;
24      };
25
26      if  (dfs(1, 1)) {  // 因图连通, 故只需从一个起点开始染色
27          cout << "YES" << endl;
28          for (auto [u, v] : eds) cout << (colors[u] < colors[v]);
29          cout << endl;
30      }
31      else cout << "NO" << endl;
32  }
33
34  int main() {
35      solve();
36  }
```

14.12 差分约束

差分约束用于求形如$x_i \le x_j + c_k$或$x_i \ge x_j + c_k$不等式组的可行解、最优解(最值).

设求完最短路后的图中有边$j \xrightarrow{c} i$,则$dis[i] \le dis[j] + c$.若图中不存在负环,则可通过求源点到汇点的最短路求得一组$(x_i.x_j)$满足$x_i \le x_j + c_k$,它是原不等式组的一组可行解,其中要求从选取的源点出发可遍历到图中的所有的边.显然差分约束问题与不含负环的单源最短路问题等价.注意每个不等式对应到图中的一条边而不是一个点,因为点可能是孤立的.从源点能到达任意点必能到达任意边,反之不然.

若不等式组形如$x_i \ge x_j + c_k$,则连边$j \xrightarrow{c_k} i$,转化为最长路.

图中存在负环与不等式组$x_i \le x_j + c_k$无解等价,同理图中存在正环与不等式组$x_i \ge x_j + c_k$无解等价.

[证] 以证明前者为例.①若存在负环,不妨设$x_1 \xrightarrow{c_1} x_2 \xrightarrow{c_2} x_3 \to \cdots \xrightarrow{c_{k-1}} x_k \xrightarrow{c_k} x_1$是负环,则
$x_2 \le x_1 + c_1, x_3 \le x_2 + c_2, \cdots, x_k \le x_{k-1} + c_{k-1}, x_1 \le x_k + c_k$,进而
$x_2 \le x_1 + c_1 \le x_k + c_k + c_1 \le \cdots \le x_2 + (c_2 + c_3 + \cdots + c_k + c_1)$,因该环是负环,则$c_1 + \cdots + c_k < 0$,进而$x_2 < x_2$,矛盾,这表明:存在负环的不等式组无解.

②若不等式组存在矛盾,则会推出$x_i < x_i$,这对应到图中的边即$dis[i] < dis[i]$,这表明:不等式组存在矛盾时,图中存在负环.

求可行解的步骤:①将每个不等式$x_i \le x_j + c_k$转化为一条从x_j到x_i的长度为c_k的边;②找一个超级源点或虚拟源点,使得它能遍历到所有边;③从源点求一遍单源最短路,若有负环则不等式组无解;否则$x_i = dis[i]$是不等式组的一个可行解.

求最优解时,求最小值用最长路,求最大值用最短路.求最优解的不等式组一般会有$x_i \geq c_k$或$x_i \leq c_k$的条件,否则只能求出各变量间的相对关系.下面以求最大值为例:对不等式$x_i \leq c_k$,可建立一个超级源点0号点,建立边$0 \xrightarrow{c_k} i$.最终$x_i \leq x_j + c_1 \leq x_k + c_1 + c_2 \leq \cdots \leq 0 + \sum c$,在所有形如$x_i \leq c_j$的不等式链中取$c_j$的最小值,即为$x_i$的最大值.显然每个不等式链对应一条从源点0走到节点i的最短路,每个x_i的上界对应一条最短路的长度.

14.12.1 糖果

题意

给n $(1 \leq n < 1e5)$个小朋友分糖果,编号$1 \sim n$,满足k $(1 \leq k \leq 1e5)$个要求,求至少要多少个糖果才能保证每个小朋友都分到糖果并满足他们的所有要求,若无法满足所有要求,输出-1.

要求用k行输入描述,每行包含三个整数X $(1 \leq X \leq 5)$, A, B $(1 \leq A, B \leq n)$:

X	要求
$X = 1$	第A个小朋友分到的糖果等于第B个小朋友分到的糖果
$X = 2$	第A个小朋友分到的糖果少于第B个小朋友分到的糖果
$X = 3$	第A个小朋友分到的糖果不少于第B个小朋友分到的糖果
$X = 4$	第A个小朋友分到的糖果多于第B个小朋友分到的糖果
$X = 5$	第A个小朋友分到的糖果不多于第B个小朋友分到的糖果

思路

求最小值用最长路,把所有不等式转化为$\geq$的形式.①$A = B \Leftrightarrow A \geq B, B \geq A$;②$A < B \Leftrightarrow B \geq A + 1$;③$A \geq B \Leftrightarrow A \geq B$;④$A > B \Leftrightarrow A \geq B + 1$;⑤$A \leq B \Leftrightarrow B \geq A$.每个小朋友都分到糖果即$x_i \geq 1$,建立超级源点0号点$x_0 = 0$,转化为$x_i \geq x_0 + 1$.从0号点求一遍单源最长路即可求得每个$x_i$的最小值,存在正环时无解.

因对$\forall i$,有$x_i \geq 1$,则0号点可到达任意点i,进而从0号点出发能到达任意边.

最坏的情况是$X = 1$,都要建双向边,再超级源点向每个节点连一条边,故开三倍边数.最多有n^2个数,它们之和最大$1e10$,要开ll.

本题SPFA用队列会T,要用栈.

代码

```cpp
const int MAXN = 1e5 + 5, MAXM = 3e5 + 5;
int n, m;  // 点数、不等式数
int head[MAXN], edge[MAXM], w[MAXM], nxt[MAXM], idx = 0;
ll dis[MAXN];
int cnt[MAXN];  // 每个节点被遍历到的时间
bool vis[MAXN];

void add(int a, int b, int c) {
  edge[idx] = b, w[idx] = c, nxt[idx] = head[a], head[a] = idx++;
}

```

```cpp
bool spfa() {
  memset(dis, -INF, so(dis));

  stack<int> stk;
  dis[0] = 0;
  stk.push(0);
  vis[0] = true;

  while (stk.size()) {
    int t = stk.top(); stk.pop();
    vis[t] = false;

    for (int i = head[t]; ~i; i = nxt[i]) {
      int j = edge[i];
      if (dis[j] < dis[t] + w[i]) {
        dis[j] = dis[t] + w[i];
        cnt[j] = cnt[t] + 1;
        if (cnt[j] >= n + 1) return false;  // 有正环

        if (!vis[j]) {
          stk.push(j);
          vis[j] = true;
        }
      }
    }
  }
  return true;
}

int main() {
  memset(head, -1, so(head));

  cin >> n >> m;
  while (m--) {
    int x, a, b; cin >> x >> a >> b;
    switch (x) {
      case 1:add(b, a, 0), add(a, b, 0); break;
      case 2:add(a, b, 1); break;
      case 3:add(b, a, 0); break;
      case 4:add(b, a, 1); break;
      case 5:add(a, b, 0); break;
    }
  }

  for (int i = 1; i <= n; i++) add(0, i, 1);  // 建立超级源点

  if (!spfa()) cout << -1;
  else {
    ll ans = 0;
    for (int i = 1; i <= n; i++) ans += dis[i];
    cout << ans;
  }
}
```

14.12.2 区间

题意

给定n $(1 \leq n \leq 5e4)$个区间$[a_i, b_i]$ $(0 \leq a_i, b_i \leq 5e4)$和$n$个整数$c_i$ $(1 \leq c_i \leq b_i - a_i + 1)$,构造一个整数集合$S$ $s.t.$ 对$\forall i \in [1, n]$, S中满足$a_i \leq x \leq b_i$的整数x不少于c_i个.求S至少包含几个数.

思路

最坏$1 \sim 5e4$都选必然满足要求,故本题必有解.

因差分约束需建立超级源点,则可a_i++, b_i++将0号节点空出,显然这样最终答案不变.S_i表示$1 \sim i$中被选出的数的个数,则S_{5e4+1}的最小值即答案.求最小值用最长路,将不等式化为大于等于的形式.

考察S_i的性质:①$S_i \geq S_{i-1}$ $(1 \leq i \leq 5e4 + 1)$;②$S_i - S_{i-1} \leq 1$,表示是否选第i个数,等价于$S_{i-1} \geq S_i - 1$;③$S_b - S_{a-1} \geq c$,表示区间$[a, b]$中至少选c个数,等价于$S_b \geq S_{a-1} + c$.①保证了0号节点能到1号节点,1号节点能到2号节点,$\cdots$,则从0号节点出发可遍历到所有节点,进而可遍历到所有边;③保证了符合题目要求,故满足这三条性质的解与题目的解相等.

代码

```cpp
const int MAXN = 5e4 + 5, MAXM = 1.5e5 + 5;
int n;  // 不等式数
int head[MAXN], edge[MAXM], w[MAXM], nxt[MAXM], idx = 0;
int dis[MAXN];
bool vis[MAXN];
int que[MAXN];  // SPFA的队列

void add(int a, int b, int c) {
  edge[idx] = b, w[idx] = c, nxt[idx] = head[a], head[a] = idx++;
}

bool spfa() {
  memset(dis, -INF, so(dis));

  dis[0] = 0;
  vis[0] = true;
  int hh = 0, tt = 1;
  que[0] = 0;

  while (hh != tt) {
    int t = que[hh++];
    if (hh == MAXN) hh = 0;  // 循环队列
    vis[t] = false;

    for (int i = head[t]; ~i; i = nxt[i]) {
      int j = edge[i];
      if (dis[j] < dis[t] + w[i]) {
        dis[j] = dis[t] + w[i];
        // 本题必有解,无需判环
        // cnt[j] = cnt[t] + 1;
        // if (cnt[j] >= n + 1) return false;

        if (!vis[j]) {
          que[tt++] = j;
          if (tt == MAXN) tt = 0;  // 循环队列
          vis[j] = true;
```

```
        }
      }
    }
  }
  return true;
}

int main() {
  memset(head, -1, so(head));
  for (int i = 1; i < MAXN; i++)
    add(i - 1, i, 0), add(i, i - 1, -1);

  cin >> n;
  for (int i = 0; i < n; i++) {
    int a, b, c; cin >> a >> b >> c;
    a++, b++;
    add(a - 1, b, c);
  }

  spfa();

  cout << dis[50001];
}
```

14.12.3 排队布局

题意

编号$1 \sim n$的n头牛排成一条直线,牛在队伍中的顺序与其编号相同,可能有若干头牛在同一位置处.有m_l对牛间存在好感,它们希望彼此间的距离不超过l;有m_d对牛间存在反感,它们希望彼此间的距离不低于d.问是否存在满足所有要求的方案,若不存在,输出-1;若1号牛和n号牛间的距离可任意大,输出-2;否则输出1号牛与n号牛间的可能的最大距离.

第一行输入三个整数n, m_l, m_d $(2 \le n \le 1000, 1 \le m_l, m_d \le 1e4)$.接下来$m_l$行每行输入三个整数$a, b, l$ $(1 \le a, b \le n, 1 \le l \le 1e6)$,表示$a$号牛与$b$号牛间的距离不超过$l$.接下来$m_d$行每行输入三个整数$a, b, d$ $(1 \le a, b \le n, 1 \le d \le 1e6)$,表示$a$号牛与$b$号牛间的距离不低于$d$.

思路

求最大可行解,转化为求最短路.保证$a < b$后,因牛在队伍中的顺序与其编号相同,则$x_i \le x_{i+1}$ $(1 \le i < n)$,此时a号牛与b号牛间的距离不超过l即$x_b - x_a \le l$,等价于$x_b \le x_a + l$;a号牛与b号牛间的距离不低于d即$x_b - x_a \ge d$,等价于$x_a \le x_b - d$.

注意到上述三个不等式组无法保证存在一个节点能到达图中的所有边,故需建立超级源点.因两点间的距离与两点分别在哪个位置无关,只与它们间的横坐标之差有关,不妨设$x_i \le 0$ $(1 \le i \le n)$,则0号节点向每个i $(1 \le i \le n)$号节点连一条长度为0的边,则0号节点可到达图中的所有边.因0号节点到每个i $(1 \le i \le n)$号节点都有一条长度为0的边,则以0号节点为源点求最短路时,$dis[i] = 0$ $(1 \le i \le n)$,故实现时无需建立超级源点,只需初始时将$1 \sim n$号节点加入SPFA的队列中即可.

不存在合法方案等价于图中存在负环.

为判断1号牛和n号牛间的距离是否可以任意大,不妨固定$x_1 = 0$,只需判断x_n能否任意大,这等价于求1号节点到n号节点的最短路,并判断$dis[n]$是否为INF,若是则不存在$x_1 \le \cdots \le x_n$这样的不等式链,即x_1与x_n间无无限制,亦即1号牛和n号牛间的距离可以任意大;否则$dis[n]$即为1号牛与n号牛间的可能的最大距离.

总边数为初始边数加$x_i \le x_{i+1}\ \ (1 \le i < n)$连的边数,即$m_l + m_d + n$,最大为21000.

代码

```cpp
const int MAXN = 1005, MAXM = 21005;
namespace SPFA {
  int n;
  vector<pair<int, int>> edges[MAXN];
  int dis[MAXN];
  int cnt[MAXN];  // 起点到每个节点的最短路经过的边数
  bool state[MAXN];

  void init() {
    memset(dis, INF, sizeof(dis));
    memset(cnt, 0, sizeof(cnt));
    memset(state, false, sizeof(state));
  }

  bool spfa(int siz) {  // 初始时将siz个节点加入队列中,返回是否存在负环
    init();

    queue<int> que;
    for (int i = 1; i <= siz; i++) {
      que.push(i);
      dis[i] = 0, state[i] = true;
    }

    while (que.size()) {
      auto u = que.front(); que.pop();
      state[u] = false;

      for (auto [v, w] : edges[u]) {
        if (dis[v] > dis[u] + w) {
          dis[v] = dis[u] + w;
          cnt[v] = cnt[u] + 1;
          if (cnt[v] >= n) return true;  // 存在负环

          if (!state[v]) {
            que.push(v);
            state[v] = true;
          }
        }
      }
    }
    return false;
  }
}
using namespace SPFA;

void solve() {
  int ml, md; cin >> n >> ml >> md;

  for (int i = 1; i < n; i++) edges[i + 1].push_back({ i,0 });
  while (ml--) {
    int a, b, l; cin >> a >> b >> l;
    if (a > b) swap(a, b);
    edges[a].push_back({ b,l });
```

```
  }
  while (md--) {
    int a, b, d; cin >> a >> b >> d;
    if (a > b) swap(a, b);
    edges[b].push_back({ a,-d });
  }

  if (spfa(n)) cout << -1 << endl;  // 存在负环
  else {
    spfa(1);  // 求1号节点到其他节点的最短路
    cout << (dis[n] == INF ? -2 : dis[n]) << endl;
  }
}

int main() {
  solve();
}
```

14.12.4 雇佣收银员

题意

某超市24小时营业,现需雇佣收银员.不同时间段所需的收银员人数不同,$00:00\sim01:00$最少需要r_0个收银员,$01:00\sim02:00$至少需要r_2个收银员,$\cdots$,$23:00\sim00:00$至少需要r_{23}个收银员.有n个人申请岗位,其中第i $(1\le i\le n)$个人可从t_i时刻开始连续工作8小时,收银员间不换班,一定完整地工作8小时.问最少需要多少名收银员.

有t $(1\le t\le 20)$组测试数据.每组测试数据第一行输入24个整数$r_0,\cdots,r_{23}$ $(0\le r_i\le 1000)$.第二行输入一个整数n $(0\le n\le 1000)$.接下来n行每行输入一个整数,其中第i $(1\le i\le n)$行输入的整数t_i $(0\le t_i\le 23)$表示第i个申请人可以开始工作的时刻.

对每组测试数据,若存在合法的安排,则输出最少需要多少名收银员;否则输出"No Solution".

思路

求最小值,转化为求最长路.设时刻i开始工作的人数为st_i,最优方案中从st_i中保留x_i人,则$0\le x_i\le st_i$ $(1\le i\le n)$.对每个时刻i,有$x_{i-7}+x_{i-6}+\cdots+x_i\ge r_i$.

注意到每次求和都是求连续的一段和,可用前缀和优化.将下标后移一位变为$r_1,\cdots,r_{24}$和$x_1,\cdots,x_{24}$.设$x[]$的前缀和数组为$pre[]$,则上述两个不等式分别变为$0\le pre_i-pre_{i-1}\le st_i$ $(1\le i\le 24)$,$\begin{cases}pre_i+pre_{24}-pre_{i+16}\ge r_i, 0<i<7\\ pre_i-pre_{i-8}\ge r_i, i\ge 8\end{cases}$,即:①$pre_i\ge pre_{i-1}$;②$pre_{i-1}\ge pre_i-st_i$;③$pre_i\ge pre_{i+16}-pre_{24}+r_i$;④$pre_i\ge pre_{i-8}+r_i$.①保证了0号节点是超级源点.除③外,其余式子都是差分约束的标准形式.对③,可枚举pre_{24}的所有取值$[0,n]$将其转化为常数,进而将③转化为差分约束的标准形式.对每个固定的$pre_{24}=c$,等价于$pre_{24}\le c, pre_{24}\ge c$,即$pre_{24}\le pre_0+c, pre_0\le pre_{24}-c$.

因$ans=pre_{24}$,从小到达枚举pre_{24},找到第一个使得方案合法的值即为答案.枚举完$[0,n]$都未使得方案合法,则无解.

因本题数据范围较小,暴力枚举$[0,n]$也可通过.数据范围较大时,可二分pre_{24}的值.

代码

```
const int MAXN = 35, MAXM = 105;
int r[MAXN];
int st[MAXN];  // st[i]表示从时刻i开始工作的人数
namespace SPFA {
  vector<pair<int, int>> edges[MAXN];
```

```
  int dis[MAXN];
  int cnt[MAXN];  // 起点到每个节点的最长路经过的边数
  bool state[MAXN];

  void build(int c) {  // 以pre[24]=c建图
    for (int i = 0; i < MAXN; i++) edges[i].clear();

    edges[0].push_back({ 24,c }), edges[24].push_back({ 0,-c });  // pre[24]=c
    for (int i = 1; i <= 24; i++)
      edges[i].push_back({ i - 1,-st[i] }), edges[i - 1].push_back({ i,0 });
    for (int i = 1; i <= 7; i++) edges[i + 16].push_back({ i,r[i] - c });
    for (int i = 8; i <= 24; i++) edges[i - 8].push_back({ i,r[i] });
  }

  void init() {
    memset(dis, -INF, sizeof(dis));  // 注意初始化为-INF
    memset(cnt, 0, sizeof(cnt));
    memset(state, false, sizeof(state));
  }

  bool spfa(int c) {  // 当前pre[24]=c,返回方案是否合法
    build(c);
    init();

    queue<int> que;
    que.push(0);
    dis[0] = 0, state[0] = true;

    while (que.size()) {
      auto u = que.front(); que.pop();
      state[u] = false;

      for (auto [v, w] : edges[u]) {
        if (dis[v] < dis[u] + w) {
          dis[v] = dis[u] + w;
          cnt[v] = cnt[u] + 1;
          if (cnt[v] >= 25) return false;  // 存在正环

          if (!state[v]) {
            que.push(v);
            state[v] = true;
          }
        }
      }
    }
    return true;
  }
}
using namespace SPFA;

void solve() {
  memset(st, 0, sizeof(st));

  for (int i = 1; i <= 24; i++) cin >> r[i];
  int n; cin >> n;
  for (int i = 0; i < n; i++) {
    int t; cin >> t;
    st[t + 1]++;  // 下标后移一位
```

```
  }

  for (int i = 0; i <= n; i++) {  // 枚举pre[24]
    if (spfa(i)) {
      cout << i << endl;
      return;
    }
  }
  cout << "No Solution" << endl;
}

int main() {
  CaseT
  solve();
}
```

14.13 负环

一般用SPFA算法求负环,有如下两种方法:

①统计每个节点的入队次数,若某个节点入队次数为n,则存在负环.该方法与Bellman-Ford算法等价.

该方法在某些图上会有玄学优化,但在$1 \xrightarrow{-1} 2 \xrightarrow{-1} \cdots \xrightarrow{-1} n \xrightarrow{-1} 1$这样的环中,转完一圈后还需继续转才会出现节点$1$入队次数为$n$,时间复杂度$n(n-1)+1=O(n^2)$.

②统计起点到每个节点的最短路中包含的边数,若某个节点的最短路的边数$\geq n$,则存在负环.该方法较常用.

该方法在$1 \xrightarrow{-1} 2 \xrightarrow{-1} \cdots \xrightarrow{-1} n \xrightarrow{-1} 1$这样的环中,转一圈回到节点$1$时即找到一个节点的最短路的边数$\geq n$,时间复杂度$O(n)$.

若不要求负环能从起点到达,则初始时将所有节点入队,并将$dis[]$初始化为0,这是因为可建立一个超级源点S指向每个节点,此时图中的负环能从S到达,且第一步更新会将所有节点的$dis[]$更新为0.事实上$dis[]$的初始值不影响结果,因为如若图中存在负环,则SPFA算法会迭代无限次,将节点的$dis[]$更新为$-INF$.因图的节点数和边权都是有限值,故会更新无限次,无限次$\geq n$.

SPFA算法求负环时时间复杂度可能达到$O(nm)$,可能TLE.此时有实际上未必正确,但经验上正确的优化,即统计所有节点的入队次数之和,若它超过某个值(一般取$2m \sim 3m$),则认为存在负环.此外,存在负环时将SPFA算法中的队列换成栈可能会变快.

14.13.1 虫洞

题意

虫洞可视为一条单向路径,经过它可回到过去的某个时刻(相比于进入虫洞前).农场包含n块田地、m条双向路径和w个虫洞,田地编号$1 \sim n$.问能否从某个田地出发经过一条路径,在出发时刻之前回到出发地.

有t $(1 \leq t \leq 5)$组测试数据.每组测试数据第一行输入三个整数 n, m, w $(1 \leq n \leq 500, 1 \leq m \leq 2500, 1 \leq w \leq 200)$.接下来$m$行每行输入三个整数 u, v, t $(1 \leq u, v \leq n, 1 \leq t \leq 1e4)$,表示田地$u$与田地$v$间存在双向路径,经过该路径花费的时间为$t$.接下来$m$行每行输入三个整数$u, v, t$ $(1 \leq u, v \leq n, 1 \leq t \leq 1e4)$,表示存在一条从田地$u$到田地$v$的虫洞,经过该虫洞可回到$t$秒前.

思路

若图中存在负环则能回到过去.

用SPFA算法求负环,总时间复杂度$O(tnm)$,最坏$5 \times 500 \times (2500 \times 2 + 200)$,约1e7.

代码

```cpp
const int MAXN = 505, MAXM = 5205;
namespace SPFA {
  int n;
  vector<pair<int, int>> edges[MAXN];
  bool state[MAXN];
  int dis[MAXN];
  int cnt[MAXN];  // 节点的最短路包含的边数

  void init() {
    for (int i = 1; i <= n; i++) {
      edges[i].clear();
      state[i] = false, dis[i] = cnt[i] = 0;
    }
  }

  bool spfa() {  // 返回是否存在负环
    queue<int> que;
    for (int i = 1; i <= n; i++) que.push(i), state[i] = true;

    while (que.size()) {
      auto u = que.front(); que.pop();
      state[u] = false;

      for (auto [v, w] : edges[u]) {
        if (dis[v] > dis[u] + w) {
          dis[v] = dis[u] + w;
          cnt[v] = cnt[u] + 1;
          if (cnt[v] >= n) return true;

          if (!state[v]) {
            que.push(v);
            state[v] = true;
          }
        }
      }
    }
    return false;
  }
}
using namespace SPFA;

void solve() {
  int m1, m2; cin >> n >> m1 >> m2;
  init();

  while (m1--) {
    int u, v, w; cin >> u >> v >> w;
    edges[u].push_back({ v,w }), edges[v].push_back({ u,w });
  }
  while (m2--) {
```

```
    int u, v, w; cin >> u >> v >> w;
    edges[u].push_back({ v,-w });
  }

  cout << (spfa() ? "YES" : "NO") << endl;
}

int main() {
  CaseT
  solve();
}
```

14.13.2 观光奶牛

题意

给定一个包含编号$1 \sim n$的n个节点和编号$1 \sim m$的m条边的有向图,i号节点的权值为f_i,i号边的权值为t_i.求图中的一个环,使得环上各点的权值除以环上各边的权值最大,结果保留两位小数.

第一行输入两个整数n, m $(2 \le n \le 1000, 2 \le m \le 5000)$.接下来$n$行每行输入一个整数,其中第$i$ $(1 \le i \le n)$行输入的整数f_i $(1 \le f_i \le 1000)$表示i号节点的权值.接下来m行每行输入三个整数a, b, t_i $(1 \le a, b \le n, 1 \le t_i \le 1000)$,表示存在一条从节点$a$到节点$b$的有向边,权值为$t_i$.数据保证图中存在环.

思路

在图论问题中,求形如$\frac{\sum f_i}{\sum t_i}$的最大值的问题称为**0/1分数规划**,常用做法是二分.在本题中,该值的最小值为0,最大值在分子都取1000,分母都取1时取得.二分每个值mid,检查图中是否存在一个环的该值$> mid$.

$\sum f_i > mid \cdot \sum t_i \Leftrightarrow \sum f_i - mid \cdot \sum t_i$.因既有点权又有边权不方便,故可将点权转移到边上.因图中的边为有向边,故放在出边或入边上等价.下面以放在出边上为例,如有向边$u \xrightarrow{t_k} v$中节点u的权值为f_i时,该边的边权变为$f_i - mid \cdot t_k$,问题转化为图中是否存在一个环的$\sum (f_i - mid \cdot t_k) > 0$,即求正环,用SPFA算法即可.

代码

```
const double eps = 1e-4;

int cmp(double a, double b) {
  if (fabs(a - b) < eps) return 0;
  else return a < b ? -1 : 1;
}

const int MAXN = 1005, MAXM = 5105;
namespace SPFA {
  int n;
  int wf[MAXN];  // 点权
  vector<pair<int, int>> edges[MAXN];
  bool state[MAXN];
  double dis[MAXN];
  int cnt[MAXN];  // 节点的最长路包含的边数

  void init() {
```

```
    memset(state, false, sizeof(state));
    memset(dis, 0, sizeof(dis));
    memset(cnt, 0, sizeof(cnt));
  }

  bool spfa(double mid) {  // 返回是否存在正环
    init();

    queue<int> que;
    for (int i = 1; i <= n; i++) que.push(i), state[i] = true;

    while (que.size()) {
      auto u = que.front(); que.pop();
      state[u] = false;

      for (auto [v, w] : edges[u]) {
        if (dis[v] < dis[u] + wf[u] - mid * w) {
          dis[v] = dis[u] + wf[u] - mid * w;
          cnt[v] = cnt[u] + 1;
          if (cnt[v] >= n) return true;

          if (!state[v]) {
            que.push(v);
            state[v] = true;
          }
        }
      }
    }
    return false;
  }
}
using namespace SPFA;

void solve() {
  int m; cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> wf[i];
  while (m--) {
    int u, v, w; cin >> u >> v >> w;
    edges[u].push_back({ v,w });
  }

  double l = 0, r = 1e6;
  while (cmp(l, r)) {
    double mid = (l + r) / 2;
    if (spfa(mid)) l = mid;
    else r = mid;
  }
  cout << fixed << setprecision(2) << l << endl;
}

int main() {
  solve();
}
```

14.13.3 单词环

题意

有n个只包含小写英文字母的字符串.若字符串A的结尾两个字符与字符串B开头两个字符匹配,则称A与B能相连(注意A与B能相连不代表B与A能相连).现需从给定的字符串中选出一些单词,将它们收尾相连形成一个环串(一个串也可收尾相连),使得该环串的平均长度最大.如字符串"ababc"、"bckjaca"、"caahoynaab"依次首尾相连形成一个环串,长度为$5+7+10=22$(重复部分算两次),共用3个字符串,故平均长度为$\frac{27}{3}\approx 7.33$.

有多组测试数据.每组测试数据第一行输入一个整数n $(1\le n\le 1e5)$.接下来n行每行输入一个长度不超过1000的只包含小写英文字母的字符串.输入$n=0$时结束.

对每组测试数据,若不存在环串,输出"No solution";否则输出最长环的平均长度,误差不超过0.01.

思路

一种直接的建图方式:若字符串A能与字符串B相连,则连一条节点A到节点B的有向边.但这样最坏有$O(n^2)$个点,无法接受.

考虑优化,将每个字符串的头两个字母代表的节点向尾两个字母连一条长度为串长的边,这样最多有$26\times 26=676$个节点和$1e5$条边.正确性:每条边是一个字符串,节点处表示字符串相连.

平均长度$=\frac{\sum w_i}{\sum 1}$,可用$0/1$分数规划求解,平均长度最小为0,最大为1000.

$$\frac{\sum w_i}{\sum 1}>mid\Leftrightarrow\sum w_i>mid\cdot\sum 1\Leftrightarrow\sum w_i-mid\cdot\sum 1>0.$$

若$mid=0$时图中都不存在正环,则无解.

本题朴素的SPFA算法会TLE,需用节点更新次数较多时认为有负环的优化.

代码

```cpp
const double eps = 1e-4;

int cmp(double a, double b) {
  if (fabs(a - b) < eps) return 0;
  else return a < b ? -1 : 1;
}

const int MAXN = 685, MAXM = 1e5 + 5;
namespace SPFA {
  int n;
  int head[MAXN], edge[MAXM], w[MAXM], nxt[MAXM], idx;
  bool state[MAXN];
  double dis[MAXN];
  int cnt[MAXN];  // 节点的最长路包含的边数

  void add(int a, int b, int c) {
    edge[idx] = b, w[idx] = c, nxt[idx] = head[a], head[a] = idx++;
  }

  void init() {
```

```
    memset(state, false, sizeof(state));
    memset(cnt, 0, sizeof(cnt));
  }

  bool spfa(double mid) {  // 返回是否存在正环
    init();

    queue<int> que;
    for (int i = 0; i < 676; i++) que.push(i), state[i] = true;

    int tot = 0;  // 所有节点被更新的次数之和
    while (que.size()) {
      auto u = que.front(); que.pop();
      state[u] = false;

      for (int i = head[u]; ~i; i = nxt[i]) {
        int v = edge[i];
        if (dis[v] < dis[u] + w[i] - mid) {
          dis[v] = dis[u] + w[i] - mid;
          cnt[v] = cnt[u] + 1;
          if (cnt[v] >= MAXN) return true;
          if (++tot > 1e4) return true;  // 节点被更新的次数过多,大概率有正环

          if (!state[v]) {
            que.push(v);
            state[v] = true;
          }
        }
      }
    }
    return false;
  }
}
using namespace SPFA;

void solve() {
  while (cin >> n, n) {
    memset(head, -1, sizeof(head));
    idx = 0;

    while (n--) {
      string s; cin >> s;
      int len = s.length();
      if (len >= 2) {
        int left = (s[0] - 'a') * 26 + s[1] - 'a';
        int right = (s[len - 2] - 'a') * 26 + s[len - 1] - 'a';
        add(left, right, len);
      }
    }

    if (!spfa(0)) cout << "No solution" << endl;
    else {
      double l = 0, r = 1000;
      while (cmp(l, r)) {
        double mid = (l + r) / 2;
        if (spfa(mid)) l = mid;
        else r = mid;
      }
```

```
      cout << l << endl;
    }
  }
}

int main() {
  solve();
}
```

14.13.4 Coins Respawn

原题指路:https://atcoder.jp/contests/abc137/tasks/abc137_e

题意 (2 s)

给定一张包含编号$1 \sim n$的n $(2 \leq n \leq 2500)$个节点和编号$1 \sim m$的m $(1 \leq m \leq 5000)$条边的有向图, 其中i $(1 \leq i \leq m)$号边从节点a_i $(1 \leq a_i \leq n)$指向节点b_i $(1 \leq b_i \leq n)$, 经过该边可获得c_i $(1 \leq c_i \leq 1\mathrm{e}5)$个金币, 保证存在从节点1到节点$n$的路径. 某人初始时有0个金币, 他要从节点1走到节点n, 一条边可经过多次. 到达节点n时, 可选择继续游戏或终止游戏. 给定一个整数p $(0 \leq p \leq 1\mathrm{e}5)$, 设他终止游戏时经过的边数为$t$, 则需扣除$t \cdot p$个金币, 若剩余金币数不足$t \cdot p$, 则需扣除所有金币. 求他剩余金币数的最大值, 若不存在最大值, 输出-1.

思路

将经过i $(1 \leq i \leq m)$号边可获得的金币改为$(c_i - p)$, 问题转化为求节点1到节点n的最长路, 若图中存在在节点1到节点n的路径上的正环, 则无解.

判断正环是否在节点1到节点n的路径上可从节点n开始按逆邻接表DFS.

代码

```
struct SPFA {
    int n;
    vector<vector<pair<int, int>>> edges;  // 邻接表
    vector<vector<int>> iedges;  // 逆邻接表
    vector<bool> vis;  // vis[u]表示节点u是否能从节点n到达
    vector<bool> state;
    vector<ll> dis;
    vector<int> cnt;  // cnt[u]表示起点到节点u的最长路包含的边数
    bool flag = false;  // 无解flag

    SPFA(int _n, const vector<vector<pair<int, int>>>& _edges,
        const vector<vector<int>>& _iedges)
        :n(_n), edges(_edges), iedges(_iedges) {
        vis.resize(n + 1), state.resize(n + 1);
        dis.resize(n + 1), cnt.resize(n + 1);
        fill(all(dis), -INF);  // 求最长路时将dis[]初始化为-INF

        function<void(int)> dfs = [&](int u)->void {
            if (vis[u]) return;

            vis[u] = true;
            for (auto v : iedges[u]) dfs(v);
        };
```

```cpp

        dfs(n);  // 预处理vis[]
    }

    void spfa(int s) {
        queue<int> que;
        que.push(s);
        state[s] = true;
        dis[s] = 0;

        while (que.size()) {
            auto u = que.front(); que.pop();
            state[u] = false;
            for (auto [v, w] : edges[u]) {
                if (!vis[v]) continue;  // 节点v不可到达节点n

                if (dis[v] < dis[u] + w) {
                    dis[v] = dis[u] + w;
                    if ((cnt[v] = cnt[u] + 1) >= n) {
                        flag = true;
                        return;
                    }

                    if (!state[v]) {
                        que.push(v);
                        state[v] = true;
                    }
                }
            }
        }
    }
};

void solve() {
    int n, m, p; cin >> n >> m >> p;
    vector<vector<pair<int, int>>> edges(n + 1);  // 邻接表
    vector<vector<int>> iedges(n + 1);  // 逆邻接表
    while (m--) {
        int u, v, w; cin >> u >> v >> w;

        edges[u].push_back({ v, w - p });
        iedges[v].push_back(u);
    }

    SPFA solver(n, edges, iedges);
    solver.spfa(1);
    cout << (solver.flag ? -1 : max(solver.dis[n], (ll)0)) << endl;
}

int main() {
    solve();
}
```

14.14 单源最短路的建图

14.14.1 热浪

题意

有编号$1 \sim n$的n个城市,除起点和终点外,每个城市都由双向道路连向至少两个其他城市,每条路有一个通过费用.给定一个地图,其中包含m条直接连接两城市的道路,每条道路的起点为R_s,终点为R_e,费用为C_i.求从起始城市S到终点城市T的最小总费用.

第一行输入四个整数n, m, S, T $(1 \leq n \leq 2500, 1 \leq m \leq 6200, 1 \leq S, T \leq n)$.接下来$m$行每行输入三个整数$R_s, R_e, C_i$ $(1 \leq R_s, R_e \leq n, 1 \leq C_i \leq 1000)$,描述一条道路.

输出从S到T的最小费用,数据保证有解.

代码

```cpp
const int MAXN = 2505, MAXM = 6205 << 1;
int n, m, S, T;  // 节点数、边数、起点、终点
int head[MAXN], edge[MAXM], w[MAXM], nxt[MAXM], idx;
int dis[MAXN];
int que[MAXN];
bool vis[MAXN];

void add(int a, int b, int c) {
  edge[idx] = b, w[idx] = c, nxt[idx] = head[a], head[a] = idx++;
}

void spfa() {
  memset(dis, INF, so(dis));

  int hh = 0, tt = 1;
  dis[S] = 0, que[0] = S, vis[S] = true;

  while (hh != tt) {
    int tmp = que[hh++];
    vis[tmp] = false;
    if (hh == MAXN) hh = 0;  // 循环队列

    for (int i = head[tmp]; ~i; i = nxt[i]) {
      int j = edge[i];
      if (dis[j] > dis[tmp] + w[i]) {
        dis[j] = dis[tmp] + w[i];
        if (!vis[j]) {
          que[tt++] = j;
          if (tt == MAXN) tt == 0;
          vis[j] = true;
        }
      }
    }
  }
}

int main() {
  memset(head, -1, so(head));

  cin >> n >> m >> S >> T;
  for (int i = 0; i < m; i++) {
    int u, v, w; cin >> u >> v >> w;
```

```
    add(u, v, w), add(v, u, w);
  }

  spfa();
  cout << dis[T];
}
```

14.14.2 信使

题意

有个n个城市,每个城市可能与其他若干城市有通信联系.信使负责在城市间传递信息,花费的时间以天为单位.指挥部设在第一个城市,下达命令后,指挥部派出若干信使向与指挥部相连的城市送信.一个城市收到信后,该城市内的信使以相同方式向其他城市送信,信在一个城市内停留的时间忽略不计.n个城市全部收到信后任务完成.每个城市都安排了足够的信使(若一个城市与其他k个城市有通信联系,则该城市至少有k个信使).求任务完成的最短时间.

第一行输入整数n, m $(1 \leq n \leq 100, 1 \leq m \leq 200)$,表示城市数和通信线路数.接下来$m$行每行输入三个整数$i, j, k$ $(1 \leq i, j \leq n, 1 \leq k \leq 1000)$,表示城市$i$与城市$j$间存在双向的通信线路,且该线路需花费$k$天.

若能完成任务,输出所需的最短时间;否则输出-1.

思路

每个城市第一次收到信的时间是它与指挥部的最短距离,故最晚收到信的城市是离指挥部最远的城市.

代码

```
const int MAXN = 105;
int n, m;
int dis[MAXN][MAXN];

void floyd() {
  for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++)
        dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
    }
  }
}

int main() {
  memset(dis, INF, so(dis));

  cin >> n >> m;
  for (int i = 1; i <= n; i++) dis[i][i] = 0;  // 初始化
  for (int i = 0; i < m; i++) {
    int a, b, c; cin >> a >> b >> c;
    dis[a][b] = dis[b][a] = min(dis[a][b], c);
  }

  floyd();

  int ans = 0;
  for (int i = 1; i <= n; i++) {
    if (dis[1][i] == INF) {
```

```
      ans = -1;
      break;
    }
    else ans = max(ans, dis[1][i]);
  }
  cout << ans;
}
```

14.14.3 香甜的黄油

题意

每头奶牛在自己喜欢的牧场中,一个牧场中可能有多头牛.给定牧场的地图和每头牛所在的牧场,求一个牧场使得所有牛到该牧场的距离之和最短,输出该最短距离.

第一行输入三个整数n, p, c $(1 \leq n \leq 500, 2 \leq p \leq 800, 1 \leq c \leq 1450)$,分别表示奶牛数、牧场数、牧场间的道路数.第二行输入n个整数,分别表示编号$1 \sim n$的奶牛所在的牧场.接下来c行每行输入三个数u, v, w $(1 \leq u, v \leq n, 1 \leq w \leq 255)$,表示牧场$u$与$v$间相连的双向道路长度为$d$.

输出最短距离,数据保证有解.

思路

以每个牧场为起点跑一遍SPFA.

代码

```
const int MAXN = 805, MAXM = 1500 << 1;
int n, p, m;
int cow[MAXN];
int head[MAXN], edge[MAXM], w[MAXM], nxt[MAXM], idx;
int dis[MAXN];
bool vis[MAXN];

void add(int a, int b, int c) {
  edge[idx] = b, w[idx] = c, nxt[idx] = head[a], head[a] = idx++;
}

int spfa(int start) {
  memset(dis, INF, so(dis));

  qi que;
  que.push(start);
  dis[start] = 0, vis[start] = true;

  while (que.size()) {
    int tmp = que.front(); que.pop();
    vis[tmp] = false;

    for (int i = head[tmp]; ~i; i = nxt[i]) {
      int j = edge[i];
      if (dis[j] > dis[tmp] + w[i]) {
        dis[j] = dis[tmp] + w[i];
        if (!vis[j]) {
          que.push(j);
```

```
          vis[j] = true;
        }
      }
    }
  }

  int res = 0;
  for (int i = 0; i < n; i++) {
    int j = cow[i];
    if (dis[j] == INF) return INF;
    res += dis[j];
  }
  return res;
}

int main() {
  memset(head, -1, so(head));

  cin >> n >> p >> m;
  for (int i = 0; i < n; i++) cin >> cow[i];
  for (int i = 0; i < m; i++) {
    int a, b, c; cin >> a >> b >> c;
    add(a, b, c), add(b, a, c);
  }

  int ans = INF;
  for (int i = 1; i <= p; i++) ans = min(ans, spfa(i));
  cout << ans;
}
```

14.14.4 最小花费

题意

n个人中某些人的银行账号间可相互转账,这些人转账的手续费不同.给定这些人相互转账时需从转账金额中扣除百分之几的手续费,问A至少需转账多少钱才能使B收到100元.

第一行输入两两整数n, m $(1 \leq n \leq 2000, 1 \leq m \leq 1\text{e}5)$,分别表示总人数和可相互转账的人的对数.接下来$m$行每行输入三个整数$x, y, z$ $(1 \leq x, y \leq n, 0 < z < 100)$,表示编号为$x$的人和编号为$y$的人间相互转账需扣除$z$ %的手续费.最后一行输入A和B $(1 \leq A, B \leq n)$,数据保证A与B可直接或间接转账.

输出A至少需转账多少钱才能使B收到100元,精确到小数点后8位.

思路

设A转账的钱为d_A,则B收到的钱$d_B = d_A \cdot \prod_i w_i$,其中$w_i$为$A$到$B$的路径需扣除的手续费的百分比.因$d_B = 100$,为使$d_A$最小,应使$\prod_i w_i$最大.

在Dijkstra算法中将用边权之和更新最短距离改为用边权之积更新最长距离,这等价于对边权之积取log,因$w_i \in (0, 1)$,则取log后边权都为负,问题转化为在负权图上求最长路,进而每个边权乘-1后转化为在正权图上求最短路.

代码

```cpp
const int MAXN = 2005;
int n, m, S, T;
double graph[MAXN][MAXN];
double dis[MAXN];
bool vis[MAXN];

void dijkstra() {
  dis[S] = 1;  // 乘法的单位元为1
  for (int i = 1; i <= n; i++) {
    int tmp = -1;
    for (int j = 1; j <= n; j++)
      if (!vis[j] && (tmp == -1 || dis[tmp] < dis[j])) tmp = j;
    vis[tmp] = true;

    for (int j = 1; j <= n; j++) dis[j] = max(dis[j], dis[tmp] * graph[tmp][j]);
  }
}

int main() {
  cin >> n >> m;
  while (m--) {
    int a, b, c; cin >> a >> b >> c;
    double z = (100.0 - c) / 100;
    graph[a][b] = graph[b][a] = max(graph[a][b], z);
  }
  cin >> S >> T;

  dijkstra();
  cout << fixed << setprecision(8) << 100 / dis[T];
}
```

14.14.5 最优乘车

题意

某旅游胜地设有单程巴士线路,每条单程巴士线路从某巴士站出发,途径若干个巴士站后到达终点站,巴士站编号$1 \sim n$.一旅客想到n号巴士站,但若从1号巴士站无直达n号巴士站的巴士,则他需先乘某路巴士到某巴士站,再换乘其他巴士.求一个最优乘车方案,使得他换成次数最少.

第一行输入整数m, n $(1 \leq m \leq 100, 2 \leq n \leq 500)$,分别表示巴士线路数、巴士站数.接下来$m$行每行描述一条巴士线路,其中第$i$描述第$i$条巴士线路依次途径的巴士站编号.

若他可从1号巴士站到达n号巴士站,输出最小换乘次数;否则输出"NO".

思路

用BFS求起点到每个节点需乘车的最小次数,起点和终点不重合时,$\text{换乘次数} = \text{乘车次数} - 1$.

代码

```cpp
const int MAXN = 505;
int m, n;  // 线路数、车站数
bool graph[MAXN][MAXN];
int dis[MAXN];  // dis[u]表示从节点1到节点u需乘车的最小次数
int stop[MAXN];

void bfs() {  // 求起点到每个节点需乘车的最小次数
  qi que;
  que.push(1);
  dis[1] = 0;

  while (que.size()) {
    int tmp = que.front(); que.pop();
    for (int i = 1; i <= n; i++) {
      if (graph[tmp][i] && dis[i] > dis[tmp] + 1) {
        dis[i] = dis[tmp] + 1;
        que.push(i);
      }
    }
  }
}

int main() {
  memset(dis, INF, so(dis));

  cin >> m >> n;
  string line; getline(cin, line);
  while (m--) {
    getline(cin, line);
    stringstream ss(line);
    int idx = 0, p;
    while (ss >> p) stop[idx++] = p;

    for (int j = 0; j < idx; j++)  // 每个公交站向其能到达的公交站连边
      for (int k = j + 1; k < idx; k++) graph[stop[j]][stop[k]] = true;
  }

  bfs();

  if (dis[n] == INF) cout << "NO";
  else cout << max(dis[n] - 1, 0);  // 注意换乘次数要-1,特判起点和终点重合的情况
}
```

14.14.6 昂贵的聘礼

题意

探险家到某部落中与酋长的女儿相爱,他向酋长求亲.酋长要他用10000金币作为聘礼才答应把女儿嫁给他.探险家拿不出这么多金币,请求酋长降低要求.酋长说:"若你能弄来大祭司的皮袄,我只要8000金币.若你能弄来他的水晶球,我只要5000金币".探险家到大祭司处向他要求皮袄或水晶球,大祭司要求他用金币购买,或替他弄来其他物品以降低价格.探险家找到其他人,其他人也提出了类似的要求,即要么用金币换,要么弄来其他物品来降低价格.探险家没必要用多样东西去换一样东西,因为这不会得到更低的价格,即若同时给酋长弄来了大祭司的皮袄和水晶球,也只能选取一个物品来降低价格.

部落中等级森严,地位等级差距超过一定限制的两人不能直接接触,包括交易.探险家是外来人,不受此约束,但若他与某个地位较低的人进行交易,则地位较高的人不会再与他交易,因为他们认为这是间接接触.

为探险家设计一个方案,使得他可以花最少的金币娶到酋长的女儿.为简化,将所有物品从1开始编号,酋长的允诺也视为一个物品,编号为1.每个物品有价格p,主人的等级l,以及一系列的替代品t_i和它们对应的优惠v_i.若两人地位等级相差超过m,则不能进行间接交易.

第一行输入两整数m, n $(1 \leq m \leq n \leq 100)$,表示地位等级差距限制和物品总数.接下来$n$行依次描述第$1 \sim n$个物品,每行输入三个非负整数$p, l, x$ $(1 \leq p \leq 1e5, 1 \leq l \leq n, 0 \leq x < n)$,分别表示该物品的价格、主人的地位和代替品数.接下来x行每行输入两整数t, v,分别表示代替品的编号和优惠价格.

输出探险家娶到酋长的女儿所需的最少金币数.

思路

显然有解,因可直接花10000金币.

忽略地位等级差距限制,考虑建图转化为最短路问题.以初始时的状态为虚拟源点S(节点0),它向所有物品连边.1号节点为终点,转化为从S到节点1的最短路.显然任一合法方案与图中的从S到节点1的路径一一对应.

考虑地位等级差距限制.注意到m最大为100,可枚举每个等级区间,每次只走等级在区间内的节点.朴素Dijkstra算法时间复杂度$O(n^2)$,总时间复杂度$O(n^2m)$.

代码

```cpp
const int MAXN = 105;
int n, m;  // 物品数、地位等级差距限制
int level[MAXN];  // 每个物品的等级
int w[MAXN][MAXN];  // w[u][v]表示节点u到节点v的边的边权
int dis[MAXN];
bool vis[MAXN];

int dijkstra(int down, int up) {  // 等级范围[down,up]
  memset(dis, INF, so(dis));
  memset(vis, false, so(vis));
  dis[0] = 0;

  for (int i = 1; i <= n; i++) {
    int tmp = -1;
    for (int j = 0; j <= n; j++)  // 注意包括虚拟源点,j从0开始循环
      if (!vis[j] && (tmp == -1 || dis[tmp] > dis[j])) tmp = j;

    vis[tmp] = true;
    for (int j = 1; j <= n; j++) {
      if (level[j] >= down && level[j] <= up)
        dis[j] = min(dis[j], dis[tmp] + w[tmp][j]);
    }
  }

  return dis[1];
}

int main() {
  memset(w, INF, so(w));

  cin >> m >> n;
  for (int i = 1; i <= n; i++) w[i][i] = 0;  // 初始化
```

```
33
34    for (int i = 1; i <= n; i++) {
35      int price, cnt; cin >> price >> level[i] >> cnt;
36      w[0][i] = min(price, w[0][i]);  // 虚拟节点向每个物品连边,重边保留长度最短的
37      while (cnt--) {
38        int id, cost; cin >> id >> cost;
39        w[id][i] = min(w[id][i], cost);  // 重边保留长度最短的
40      }
41    }
42
43    int ans = INF;
44    for (int i = level[1] - m; i <= level[1]; i++)  // 枚举等级的最小值
45      ans = min(ans, dijkstra(i, i + m));
46    cout << ans;
47  }
```

14.14.7 Easy Glide

原题指路:https://codeforces.com/gym/103687/problem/G

题意

在二维平面上,玩家可以v_1 m/s的速度行走.当玩家到达滑行点时,可以v_2 m/s $(v_2 > v_1)$的速度滑行3 s.给定起点S、终点T和途中的n个滑行点$p_1, \cdots, p_n$的坐标,求S到T的最短时间.

第一行输入一个整数n $(1 \le n \le 1000)$.接下来n行每行输入两整数x_i, y_i $(-1e6 \le x_i, y_i \le 1e6)$,表示滑行点$p_i$ $(1 \le i \le n)$的坐标.接下来一行输入四个整数S_x, S_y, T_x, T_y $(-1e6 \le S_x, S_y, T_x, T_y \le 1e6)$,分别表示起点$S$和终点$T$的坐标.最后一行输入两整数$v_1, v_2$ $(1 \le v_1, v_2 \le 1e6)$,分别表示行走速度和滑行速度.

输出S到T的最短时间,误差不超过$1e-6$.

思路

起点向每个滑行点和终点连单向边,边权为S以速度v_1走到该节点所需的最短时间.每个滑行点向其他滑行点和终点连单向边,边权为先以速度v_2滑行3 s,再以速度v_1走到该节点所需的最短时间.用朴素Dijkstra算法求S到T的最短路.时间复杂度$O(n^2)$.

代码

```
1   const int MAXN = 1005;
2   int n;  // 节点数
3   pii points[MAXN];  // 起点、滑行点、终点
4   int v1, v2;
5   double w[MAXN][MAXN];  // 两点间的权值
6   double dis[MAXN];
7   bool vis[MAXN];
8
9   double get_dis1(pii& a, pii& b) {  // 以速度v1行走的最短时间
10    return hypot(a.first - b.first, a.second - b.second) / v1;
11  }
12
13  double get_dis2(pii& a, pii& b) {  // 先以速度v2行走,再以速度v1行走的最短时间
14    double d = hypot(a.first - b.first, a.second - b.second);
15    double maxdis = v2 * 3;
16    if (cmp(maxdis, d) >= 0) return d / v2;
```

```
  else return 3 + (d - maxdis) / v1;
}

double dijkstra() {
  memset(dis, 0x42, so(dis));
  dis[0] = 0;

  for (int i = 1; i <= n; i++) {
    int tmp = -1;
    for (int j = 0; j <= n; j++)
      if (!vis[j] && (tmp == -1 || dis[tmp] > dis[j])) tmp = j;
    vis[tmp] = true;

    for (int j = 1; j <= n; j++) dis[j] = min(dis[j], dis[tmp] + w[tmp][j]);
  }

  return dis[n];
}

void solve() {
  memset(w, INFF, so(w));

  cin >> n;
  for (int i = 1; i <= n; i++) cin >> points[i].first >> points[i].second;
  n++;
  cin >> points[0].first >> points[0].second >> points[n].first >> points[n].second;
  cin >> v1 >> v2;

  // 建图
  for (int j = 1; j <= n; j++)  // 起点向其他滑行点和终点连边
    w[0][j] = get_dis1(points[0], points[j]);

  for (int i = 1; i <= n; i++) {  // 滑行点向其他滑行点和终点连边
    for (int j = 1; j <= n; j++) {
      if (i == j) continue;

      w[i][j] = get_dis2(points[i], points[j]);
    }
  }

  cout << fixed << setprecision(12) << dijkstra();
}

int main() {
  solve();
}
```

14.14.8 佳佳的魔法药水

题意

得到一种药水有两种方法:用配方配制或去商店购买.问:(1)至少花多少钱可以配置出目标药水?(2)共有多少种花费最少的方案,两种配置方案中有任一步骤不同则视为不同.初始时无药水.

输入第一行包含一个正整数$1 \leq n \leq 1000$,表示涉及到的药水的总数,编号$0 \sim n-1$,其中0号药水是目标药水.第二行包含n个整数,分别表示$0 \sim n$号药水在商店的价格.第三行起每行包含三个整数a, b, c $(0 \leq a, b, c \leq n-1)$,表示$1$份$a$号药水和$1$份$b$号药水混合可得到$1$份$c$号药水,数据保证配方无二义性.

输出两个用空格分隔的整数,分别表示得到0号药水的最少花费和花费最少的方案数,数据保证方案数不超过$2^{63}-1$.

思路

$cost[i]$表示获得i号药水的最少花费,$state[i]$表示i号药水是否已确定最小花费,$ans[i]$表示i号药水当前花费最少的方案数,$mix[i][j]=k$表示i、j号药水混合得到k号药水.

每次贪心一个花费最小的且未确定最小花费的药水,枚举能与该药水合成新药水的药水,更新合成的药水的最小花费.

节点的下标一般从1开始.

代码

```cpp
const int MAXN = 1005;
int n;  // 药水数
int cost[MAXN];  // cost[i]表示i号药水的最小花费
ll ans[MAXN];  // ans[i]表示i号药水当前花费最少的方案数
bool state[MAXN];  // state[i]表示i号药水是否已确定最小花费
int mix[MAXN][MAXN];  // mix[i][j]=k表示i、j号药水混合得到k号药水

int main() {
  cin >> n;
  for (int i = 1; i <= n; i++) {
    cin >> cost[i];
    ans[i] = 1;  // 直接购买
  }

  int a, b, c;
  while (cin >> a >> b >> c) mix[a + 1][b + 1] = mix[b + 1][a + 1] = c + 1;  // 建双向边,下标从1开始

  for (int i = 1; i <= n; i++) {
    int mincost = INF;  // 当前的最小花费
    int v = -1;  // 与i号药水匹配的药水
    for (int j = 1; j <= n; j++)  // 用未确定最小花费的药水更新与i号药水匹配的药水的最小花费
      if (!state[j] && cost[j] < mincost) v = j, mincost = cost[j];

    state[v] = true;  // 确定最小花费

    for (int j = 1; j <= n; j++) {
      if (state[j] && mix[v][j]) {  // 用已确定最小花费的药水更新合成出的新药水的最小花费和方案数
        // 注意下面两if不能倒过来,否则第二个if执行后第一个if也会执行
        if (cost[v] + cost[j] == cost[mix[v][j]])  // 花费相同,贡献方案数
          ans[mix[v][j]] += (ll)ans[v] * ans[j];
        if (cost[v] + cost[j] < cost[mix[v][j]])  // 找到更小的花费,更新最小花费和方案数
          cost[mix[v][j]] = cost[v] + cost[j], ans[mix[v][j]] = (ll)ans[v] * ans[j];
      }
    }
```

```
  }

  cout << cost[1] << ' ' << ans[1];  // 下标从1开始
}
```

14.15 单源最短路的应用

14.15.1 新年好

题意

某市有n个车站,有m条双向的公路连接其中某些车站,每两个车站间至多用一条公路连接,从任一车站出发都可经过一条或多条公路到达其他车站,不同的路径花费可能不同,一条路径上的花费定义为通过路径上所有公路所需的时间之和.某人需从自己家(1号车站)出发,以任意顺序拜访每个亲戚,求所需的最短时间.

第一行输入整数n, m $(1 \leq n \leq 5e4, 1 \leq m \leq 1e5)$.第二行输入五个整数$a, b, c, d, e$ $(1 < a, b, c, d, e \leq n)$,分别表示某人的五个亲戚所在车站的编号.接下来$m$行每行输入三个整数$x, y, t$ $(1 \leq x, y \leq n, 1 \leq t \leq 100)$,表示一条公路连接的两车站的编号和通过所需的时间.

输出某人拜访完所有亲戚所需的最短时间,数据保证有解.

思路

暴力做法:枚举拜访顺序,共5!种,再按顺序依次求相邻两节点间的最短路,它们之和的最小值即为答案.最短路可用堆优化的Dijktsra算法或SPFA算法求得,时间复杂度分别为$O(m \log n)$和$O(m)$,对每个排列都暴力求会超时.

考虑调整顺序,先预处理出以节点$1, a, b, c, d, e$分别为起点的到达其他节点的最短路,DFS时直接查表,若用SPFA求最短路,总时间复杂度$O(6m + 5!)$.

代码

```
const int MAXN = 5e4 + 5, MAXM = 2e5 + 5;
int n, m;
int relatives[MAXN];  // 亲戚所在的车站
vii edges[MAXN];
int dis[6][MAXN];  // dis[u][v]表示节点u与v间的最短路
bool vis[MAXN];

void dijkstra(int start, int dis[]) {  // 求起点start到其他节点的最短路
  memset(dis, INF, MAXN * 4);
  memset(vis, false, so(vis));
  dis[start] = 0;
  pque<pii, vector<pii>, greater<pii>> heap;  // 节点离起点的距离和节点的编号
  heap.push({ 0,start });

  while (heap.size()) {
    auto tmp = heap.top(); heap.pop();
    int u = tmp.second;
    if (vis[u]) continue;

    vis[u] = true;
    for (auto item : edges[u]) {
      int v = item.first, w = item.second;
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
```

```
        heap.push({ dis[v],v });
      }
    }
  }
}

int dfs(int pos, int start, int distance) {  // 准备拜访第pos个亲戚,起点为start,路径和为distance
  if (pos > 5) return distance;

  int res = INF;
  for (int i = 1; i <= 5; i++) {
    if (!vis[i]) {
      int nxt = relatives[i];
      vis[i] = true;
      res = min(res, dfs(pos + 1, i, distance + dis[start][nxt]));
      vis[i] = false;
    }
  }
  return res;
}

int main() {
  cin >> n >> m;
  relatives[0] = 1;  // 起点
  for (int i = 1; i <= 5; i++) cin >> relatives[i];

  while (m--) {
    int a, b, c; cin >> a >> b >> c;
    edges[a].push_back({ b,c }), edges[b].push_back({ a,c });
  }

  for (int i = 0; i < 6; i++)  // 预处理出节点1,a,b,c,d,e到其他节点的最短路
    dijkstra(relatives[i], dis[i]);

  memset(vis, false, so(vis));  // DFS前注意清空vis[]
  cout << dfs(1, 0, 0);  // 准备拜访第1个亲戚,从起点0出发,当前路径和为0
}
```

14.15.2 通信线路

题意

有n座通信基站,p条双向电缆,其中第i条电缆连接基站a_i和b_i.特别地,1号基站是通信公司的总站,n号基站在一个农场中.农场主希望升级通信线路,其中升级第i条电缆的花费为l_i.现有优惠活动,农场主可指定一条从1号基站到n号基站的路径,并指定路径上不超过k条电缆,由电话公司免费升级,农场主只需支付该路径上的剩余电缆中升级价格最贵的电缆的花费.求农产主至少花多少钱可完成升级.

第一行输入整数n,p,k $(0 \le k < n \le 1000, 1 \le p \le 1e4)$.接下来$p$行每行输入三个整数$a_i,b_i,l_i$ $(1 \le a_i,b_i \le n, 1 \le l_i \le 1e6)$.

若1号基站与n号基站连通,输出农场主升级电缆的最小花费;否则输出-1.

思路

一条路径的权值定义为该路径的边中权值第$(k+1)$大的边的权值,若路径包含不超过k条边,则定义其权值为0.

要使第$(k+1)$大的边权最小,考虑二分.二分边权x,将图中边权$>x$的边权视为1,边权$\leq x$的边权视为0,此时节点1到n的最短路的含义为:从节点1到n至少经过几条权值$>x$的边,这可用双端队列BFS实现.设至少经过y条权值$>x$的边,则二分的check为$y\geq k$.若边权x是答案,则至少存在一条路径使得x是该路径的边权的第$(k+1)$大数,即路径上恰有k条边的边权$>x$.考察这样的性质是否有二段性:①对$\forall x'>x$,路径上满足边权$>x$的边的数量$\leq k$,进而x的右边(含x)可能是答案;②对$\forall x'<x$,若存在一条路径使得路径上恰有k条边的边权$>x'$,则x'可能是答案,与x是答案的最小值矛盾.故x的左边(不含x)不可能是答案.综上,该性质有二段性,可二分.二分x的区间为$[0,1e6+1]$(即l_i往左和往右各扩大1单位)的原因:①$0$可能是答案;②若二分的上限为$1e6$,则二分结果$x=1e6$时无法确定节点1与节点n是否连通.若二分的上限为$1e6+1$,则二分结果$x=1e6$时必有解,$x=1e6+1$时无解.

代码

```cpp
const int MAXN = 1005, MAXM = 2e4 + 5;
int n, m, k;  // n个节点、m条边、可选k条免费
vii edges[MAXN];
int dis[MAXN];
bool vis[MAXN];

bool check(int mid) {
  memset(dis, INF, so(dis));
  memset(vis, false, so(vis));

  deque<int> que;
  que.push_back(1);
  dis[1] = 0;

  while (que.size()) {
    int u = que.front(); que.pop_front();
    if (vis[u]) continue;

    vis[u] = true;
    for (auto item : edges[u]) {
      int v = item.first;
      int w = item.second > mid;  // 边权>x的边权视为1,边权≤x的边权视为0
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        if (!w) que.push_front(v);
        else que.push_back(v);
      }
    }
  }

  return dis[n] <= k;
}

int main() {
  cin >> n >> m >> k;
  while (m--) {
    int a, b, c; cin >> a >> b >> c;
    edges[a].push_back({ b,c }), edges[b].push_back({ a,c });
  }

```

```
  int l = 0, r = 1e6 + 1;
  while (l < r) {
    int mid = l + r >> 1;
    if (check(mid)) r = mid;
    else l = mid + 1;
  }

  cout << (r == 1e6 + 1 ? -1 : r);
}
```

14.15.3 道路与航线

题意

农夫想把奶牛送到编号$1 \sim t$的t个城市,这些城市间通过编号$1 \sim r$的r条道路和编号$1 \sim p$的p条航线相连,每条道路i或航线i连接城市a_i和b_i,花费为c_i,其中道路的花费$c_i \in [0, 1e4]$,航线的花费$c_i \in [-1e4, 1e4]$.道路是双向的,且来回的花费相等.航线是单向的,只能从a_i到b_i.现出台了政策:若有一条航线可从a_i到b_i,则保证不能通过一些道路和航线从b_i返回a_i.农夫需从中心城市s出发,将奶牛送到每个城市,求他的最小花费.

第一行输入四个整数t, r, p, s $(1 \le t \le 2.5e4, 1 \le r, p \le 5e4)$.接下来$r$行每行输入三个整数$a_i, b_i, c_i$ $(1 \le a_i, b_i, s \le t)$描述一条道路.接下来$p$行每行输入三个整数$a_i, b_i, c_i$ $(1 \le a_i, b_i, s \le t)$描述一条道路.

输出t行,其中第i $(1 \le i \le t)$行表示城市s到城市i的最小花费,若不存在,输出"NO PATH".

思路

道路的特点:双向,边权非负.航线的特点:单向,边权可正可负,无环.这样的图是很多团(只由道路相连的连通块),航线连接团与团间,这样的图是拓扑图.团内部的最短路用堆优化的Dijkstra算法求出,时间复杂度$O(m \log n)$,团与团之间的最短路用拓扑排序求出,时间复杂度$O(n + m)$.具体地,先用Dijkstra算法求拓扑序为1的团的最短路,再用Dijkstra算法求拓扑序为2的团的最短路,$\cdots$.

找团的方法:可用DFS或BFS或并查集找到连通块.以DFS为例,给连通块中的每个点u编号$id[u]$,表示节点u属于哪个连通块;用一个vector<int> block[MAXN]存每个连通块中有哪些点.

输入所有航线,同时统计每个连通块的入度.再按拓扑序处理每个连通块.具体地,先将所有入度为0的连通块的编号加入队列中,每次从队首取出连通块的编号$blockid$,将该连通块内的所有点$block[blockid]$加入Dijkstra算法堆中,每次取出堆顶元素u,遍历u的所有邻点v.①若$id[u] = id[v]$,则节点u和v在同一连通块中,若此时$dis[v]$能被更新,则将节点v插入堆中;②若$id[u] \ne id[v]$,则找到了一条团之间的有向边,将连通块$id[v]$的入度-1,若减至0,将其插入拓扑排序的队列中.

时间复杂度的瓶颈为Dijkstra算法.设第i个连通块内的节点数和边数分别为n_i, m_i,则时间复杂度$m_1 \log n_1 + m_2 \log n_2 + \cdots \le (m_1 + m_2 + \cdots) \log n = m \log n$.

代码

```
const int MAXN = 2.5e4 + 5, MAXM = 1.5e5 + 5;
int n, mr, mp, S;  // 城市数、道路数、航线数、起点
vii edges[MAXN];
int id[MAXN];  // id[u]表示节点u所处的连通块的编号
int blockcnt;  // 连通块个数
vi blocks[MAXN];  // blocks[idx]表示编号为idx的连通块中的节点
int dis[MAXN];  // dis[u]表示节点u到起点的距离
bool vis[MAXN];
int in[MAXN];  // in[idx]表示编号为idx的连通块的入度
qi que;  // 拓扑排序的队列

```

```cpp
void dfs(int u, int blockid) {  // 当前节点、所在连通块的编号
  id[u] = blockid;
  blocks[blockid].push_back(u);

  for (auto item : edges[u]) {
    int v = item.first;
    if (!id[v]) dfs(v, blockid);
  }
}

void dijkstra(int blockid) {  // 求编号为blockid的连通块内部的点的最短路
  pque<pii, vii, greater<pii>> heap;  // 节点离起点的距离、节点的编号
  for (auto u : blocks[blockid]) heap.push({ dis[u],u });  // 连通块内所有点入堆

  while (heap.size()) {
    auto tmp = heap.top(); heap.pop();
    int distance = tmp.first, u = tmp.second;
    if (vis[u]) continue;

    vis[u] = true;
    for (auto item : edges[u]) {
      int v = item.first, w = item.second;
      if (id[v] != id[u] && --in[id[v]] == 0)  // 当前边是团与团间的航线
        que.push(id[v]);  // 入度减为0时加入拓扑排序的队列

      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        if (id[v] == id[u]) heap.push({ dis[v],v });  // 若u和v在同一连通块中,则v入堆
      }
    }
  }
}

void toposort() {
  memset(dis, INF, so(dis));
  dis[S] = 0;

  for (int i = 1; i <= blockcnt; i++)  // 将所有入度为0的点入队
    if (!in[i]) que.push(i);

  while (que.size()) {
    int tmp = que.front(); que.pop();
    dijkstra(tmp);  // 求每个连通块内部的点的最短路
  }
}

int main() {
  cin >> n >> mr >> mp >> S;
  while (mr--) {
    int a, b, c; cin >> a >> b >> c;
    edges[a].push_back({ b,c }), edges[b].push_back({ a,c });
  }

  for (int i = 1; i <= n; i++)  // 找到每个连通块
    if (!id[i]) dfs(i, ++blockcnt);

  while (mp--) {
    int a, b, c; cin >> a >> b >> c;
```

```
    in[id[b]]++;  // 更新连通块的入度
    edges[a].push_back({ b,c });
  }

  toposort();

  for (int i = 1; i <= n; i++) {
    if (dis[i] > INF / 2) cout << "NO PATH" << endl;  // 有负权边,故不连通为INF/2
    else cout << dis[i] << endl;
  }
}
```

一部分DP问题可视为拓扑图上的最短路或最长路问题.若DP间的状态转移存在环,但不存在负环,可转化为最短路或用Gauss消元求解.

14.15.4 最优贸易

题意

C国有编号$1 \sim n$的n个城市和m条道路,每条道路连接n个城市中的某两个城市,任意两城市间至多有一条道路直接相连.这m条道路中有一部分你是单向道路,有一部分是双向道路,其中双向道路在统计条数时也计为1条.C国地域辽阔,同种商品在不同城市可能价格不同,但同种商品在同一城市的买入价和卖出价相等.

一商人希望在旅游时利用商品在不同城市的差价赚钱.他从1号城市出发,最终到达n号城市.旅游过程中,他可经过一个城市任意次(含0次),他会选择一个经过的城市买入水晶球,并在后面经过的城市卖出水晶球,这个贸易至多进行一次,若赚不到差价,则他无需进行贸易.现给定n个城市的水晶球的价格和m条道路的信息,求他至多能赚取多少差价.

第一行输入整数n, m $(1 \leq n \leq 1e5, 1 \leq m \leq 5e5)$.第二行输入$n$个范围为$[1, 100]$的整数,依次表示$n$个城市水晶球的价格.接下来$m$行每行输入三个整数$x, y, z$ $(1 \leq x, y \leq n, z \in \{1, 2\})$描述一条道路,$z = 1$表示这条道路是城市$x$到$y$间的单向通路;$z = 2$表示这条道路是城市$x$与$y$间的双向道路.

思路

将从城市1到城市n的路径按买入和卖出的分解城市为$1, 2, \cdots, k, \cdots, n$分类,其中分界点为$k$表示在城市$k$之前买入水晶球,在城市$k$之后卖出水晶球,在城市$k$处可买入也可卖出.这样的分类方式有重复,但不影响最大值.先求从城市1到城市k的买入价的最小值$minprice[k]$和从城市k到城市n的卖出价的最大值$maxprice[k]$,因从城市1到城市k的路径与从城市k到城市n的路径独立,则赚取的最大差价为$maxprice[k] - minprice[k]$.

设城市k的水晶球价格为$w[k]$,且可从城市$u_1, u_2, \cdots, u_t$走到城市k,则$minprice[k] = \min\{minprice[u_1], \cdots, minprice[u_t], w[k]\}$.注意到DP的状态依赖可能存在环,考虑转化为最短路问题.考虑能否用Dijkstra算法求最短路,这取决于每次取出堆顶元素时能否断定其不能被其他点更新.因DP的状态依赖可能存在环,则一个点可能会入堆多次,故每次取出堆顶元素时无法保证其不能被其他点更新,故只能用Bellman-Ford算法或SPFA算法求最短路.同理$maxprice[]$可将整个图反向,再跑SPFA求得.

代码

```
const int MAXN = 1e5 + 5, MAXM = 2e6 + 5;
int n, m;  // 节点数、边数
int price[MAXN];  // price[i]表示城市i的水晶球的价格
vi edges[MAXN], redges[MAXN];  // 正向边和反向边
int minprice[MAXN];  // minprice[k]表示城市1到城市k的路径中买入水晶球价格的最小值
int maxprice[MAXN];  // maxprice[k]表示城市k到城市n的路径中买入水晶球价格的最大值
```

```cpp
bool vis[MAXN];

// dis[]为minprice[](flag为true时)或maxprice[](flag为false时),起点为start,用到正向边或反向边
void spfa(int* dis, int start, vi* edges, bool flag) {
  memset(vis, 0, so(vis));
  if (flag)  // 求minprice[]时将dis[]初始化为INF
    memset(dis, INF, so(minprice));  // 注意此处不能so(dis),因为它是指针的size

  qi que;
  que.push(start);
  vis[start] = true;
  dis[start] = price[start];

  while (que.size()) {
    int u = que.front(); que.pop();
    vis[u] = false;

    for (auto v : edges[u]) {
      if (flag && dis[v] > min(dis[u], price[v]) || !flag && dis[v] < max(dis[u], price[v])) {
        if (flag) dis[v] = min(dis[u], price[v]);
        else dis[v] = max(dis[u], price[v]);

        if (!vis[v]) {
          vis[v] = true;
          que.push(v);
        }
      }
    }
  }
}

int main() {
  cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> price[i];
  while (m--) {
    int a, b, c; cin >> a >> b >> c;
    edges[a].push_back(b), redges[b].push_back(a);  // 单向边
    if (c == 2) edges[b].push_back(a), redges[a].push_back(b);  // 双向边
  }

  spfa(minprice, 1, edges, true);  // 正向图上求minprice[]
  spfa(maxprice, n, redges, false);  // 反向图上求maxprice[]

  int ans = 0;  // 最坏可不进行贸易,赚取的差价为0
  for (int i = 1; i <= n; i++)  // 枚举买卖城市的分界点
    ans = max(ans, maxprice[i] - minprice[i]);
  cout << ans;
}
```

14.15.5 选择最佳线路

题意

城市中有编号$1 \sim n$的n个车站和m条公交线路,每条公交线路都是单向的,两车站间可能存在多条公交线路.某人想拜访朋友,他的朋友家在s号车站附近.某人从自己家附近的车站出发,可在任意车站换乘其他公交线路.求他到达s号车站的最短时间.

有多组测试数据.每组测试数据第一行输入三个整数n, m, s $(1 \leq n \leq 1000, 1 \leq m \leq 2e4, 1 \leq s \leq n)$.接下来$m$行每行输入三个整数$p, q, t$,表示存在一条从车站$p$到车站$q$的公交线路,用时为$t$.接下来一行输入一个整数$w$ $(0 < w < n)$,表示某人家附近有w个车站,他可从这些车站中任选一个作起始站.接下来一行输入w个整数,分别表示某人家附近的车站的编号.

对每组测试数据,若某人能到达s号车站,输出花费的最小时间;否则输出-1.

思路

代码

```
1
```

14.15.6 拯救大兵瑞恩

题意

有一外形是长方形的迷宫,其南北方向划分为n行,东西方向划分为m列,即整个迷宫划分为$n \times m$个单元.南北或东西方向相邻的两单元间可能互通,也可能有一扇锁着的门,也可能是一道不可逾越的墙.门可从两个方向穿过,即可视为一条无向边.迷宫中有些单元存放着钥匙,同个单元可能存放着多把钥匙.所有门被分为p类,打开同一类的门的钥匙相同,打开不同类的门的钥匙不同.某人从迷宫的$(1, 1)$单元出发,想到达(n, m)单元,他从一个单元移动到相邻单元的时间为1,忽略其他时间.求他到达(n, m)单元的最短时间.

第一行输入三个整数n, m, p $(1 \leq n, m, p \leq 10)$.第二行输入一个整数k $(1 \leq k \leq 150)$,表示迷宫中门和墙的总数.接下来k行每行输入五个整数x_1, y_1, x_2, y_2, g_i $(|x_1 - x_2| + |y_1 - y_2| = 1, 0 \leq g_i \leq p)$:当$g_i \geq 1$时,表示$(x_1, y_1)$单元与$(x_2, y_2)$单元间有一扇$g_i$类的门;当$g_i = 0$时,表示$(x_1, y_1)$单元与$(x_2, y_2)$单元间有一道不可逾越的墙.接下来一行输入一个整数$s$,表示迷宫中存放的钥匙的总数.接下来$s$行每行输入三个整数$x, y, q$,表示$(x, y)$单元有一把能开启$q$类门的钥匙.

输出某人到达(n, m)单元的最短时间,若他无法到达(n, m)单元,输出-1.

思路

代码

```
1
```

14.15.7 最短路计数

题意

有一个包含编号$1 \sim n$的n个节点和m条边的无向图,边权都为1.分别求从节点1到其他节点的最短路的条数.

第一行输入两个整数n, m $(1 \leq n \leq 1\mathrm{e}5, 1 \leq m \leq 2\mathrm{e}5)$.接下来$m$行每行输入两个整数$x, y$,表示节点$x$与节点$y$间存在无向边,注意可能存在重边和自环.

输出n行,每行输出一个非负整数,其中第i $(1 \leq i \leq n)$行输出从节点1到节点i的最短路的条数,答案对100003取模;若从节点1出发无法到达节点i,输出0.

思路

代码

```
1
```

14.15.8 观光

题意

某国有很多公交线路,公交车会从一个城市开往另一个城市,沿途可能在若干个(零个或多个)城市停靠.公交线路是单向的,两城市间可能存在多条公交线路.旅行社计划从城市s出发,终点为f城市.因旅客的景点偏好不同,旅行社将为旅客提供多种不同的线路,旅客可选择的行进路线有限制:要么所选路线为s到f的最短路,要么所选路线的总路程只比最短路多一个单位.现给定公交线路图,求旅行社至多可为旅客提供多少种不同的满足限制的线路.

有t组测试数据.每组测试数据第一行输入两个整数n, m $(2 \leq n \leq 1000, 1 \leq m \leq 1\mathrm{e}4)$,分别表示城市数和公交线路数.接下来$m$行每行输入三个整数$a, b, l$ $(1 \leq a, b \leq n, 1 \leq l \leq 1000)$,表示存在一条从城市$a$到城市$b$的长度为$l$的公交线路.接下来一行包含两个相异的整数$s, f$ $(1 \leq s, f \leq n)$,数据保证s与f间至少存在一条公交线路.

对每组测试数据,输出旅行社至多可为旅客提供多少种不同的满足限制的线路,数据保证答案不超过$1\mathrm{e}9$.

思路

代码

```
1
```

14.15.9 得到要求路径的最小带权子图

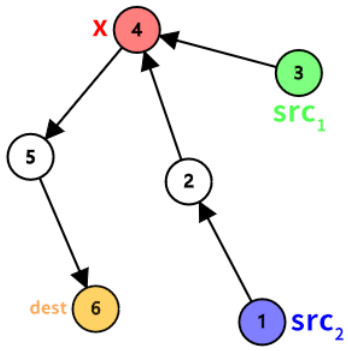
原题指路: https://leetcode.cn/problems/minimum-weighted-subgraph-with-the-required-paths/

题意

给定一张包含编号 $0 \sim (n-1)$ 的 n $(3 \leq n \leq 1e5)$ 个节点和 m $(0 \leq m \leq 1e5)$ 条边的有向图, 边权 $w_i \in [1, 1e5]$. 给定三个相异的整数 $src_1, src_2, dest$ $(0 \leq src_1, src_2, dest \leq n-1)$, 求该有向图的一个边权之和最小的子图, 使得从节点 src_1 和节点 src_2 出发, 都可到达 $dest$. 若有解, 输出子图边权和的最小值; 否则输出 -1 .

思路

[**定理1**] 边权之和最小的子图无环.



下面节点下标从 1 开始. 取 $src_1 = 3, src_2 = 1, dest = 6$, 则边权最小的子图有上图所示的形式.

[**定理2**] 从节点 src_1 到节点 $dest$ 的路径与从节点 src_2 到 $dest$ 的路径有交点 x , 且交点之后的路径重合.

先对原图的邻接表, 分别以 src_1 和 src_2 为起点分别求最短路 $dis_1[], dis_2[]$, 后对原图的逆邻接表, 以 $dest$ 为起点求最短路 $dis_3[]$, 枚举路径的交点 x , 更新答案为 $dis_1[x] + dis_2[x] + dis_3[x]$.

总时间复杂度 $O(3 \cdot n \log n) = O(n \log n)$.

代码

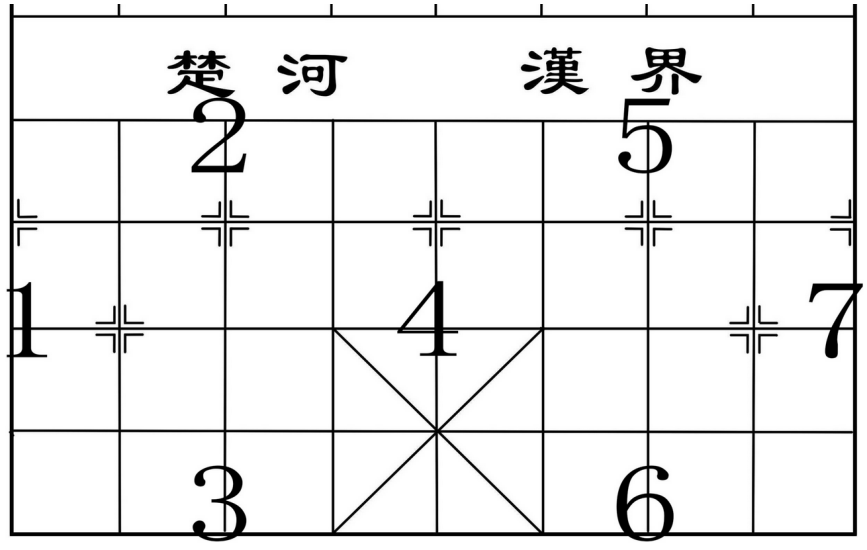
```
class Solution {
public:
    typedef long long ll;
    typedef pair<int, int> pii;

    struct DijkstraWithHeap {
        typedef pair<ll, int> pli;

        const ll INFF = 1e10;

        int n;
        vector<vector<pii>> edges;
        priority_queue<pli, vector<pli>, greater<pli>> heap;
        vector<bool> state;

        DijkstraWithHeap(int _n, const vector<vector<pii>>& _edges) :n(_n), edges(_edges)
{}

        void init() {
            state = vector<bool>(n + 1, false);
            heap = priority_queue<pli, vector<pli>, greater<pli>>();
```

```
        }

        vector<ll> dijkstra(int s) {
            init();

            vector<ll> dis(n + 1, INFF);
            dis[s] = 0;
            heap.push({ 0, s });

            while (heap.size()) {
                auto it = heap.top(); heap.pop();
                int u = it.second;
                if (state[u]) continue;

                state[u] = true;
                for (auto it : edges[u]) {
                    int v = it.first, w = it.second;
                    if (dis[v] > dis[u] + w) {
                        dis[v] = dis[u] + w;
                        heap.push({ dis[v], v });
                    }
                }
            }
            return dis;
        }
    };

    long long minimumWeight(int n, vector<vector<int>>& Edges, int src1, int src2, int dest) {
        vector<vector<pii>> edges(n + 1), iedges(n + 1);  // 邻接表、逆邻接表
        for (auto it : Edges) {
            auto u = it[0] + 1, v = it[1] + 1, w = it[2];
            edges[u].push_back({ v, w }), iedges[v].push_back({ u, w });
        }
        src1++, src2++, dest++;

        DijkstraWithHeap solver1(n, edges), solver2(n, iedges);
        auto dis1 = solver1.dijkstra(src1), dis2 = solver1.dijkstra(src2);
        auto dis3 = solver2.dijkstra(dest);

        const ll INFF = 1e10;
        ll ans = INFF;
        for (int x = 1; x <= n; x++)  // 枚举路径的交点
            ans = min(ans, dis1[x] + dis2[x] + dis3[x]);
        return ans == INFF ? -1 : ans;
    }
};
```

14.16 多源汇最短路的应用

14.16.1 342 and Xiangqi

题意



如上图为中国象棋中一方的象所能到达的位置.

有t $(1 \leq t \leq 1e5)$组测试数据.每组测试数据输入四个整数 a_1, a_2, b_1, b_2 $(1 \leq a_1, a_2, b_1, b_2 \leq 7, a_1 \neq a_2, b_1 \neq b_2)$,分别表示初始时一方的象的位置和目标位置.求将两只象从初始位置移动到目标位置的最少步数,两只象视为相同.

思路

任一有交叉的路径(即移动过程中一只象可能移动到与另一只象相同的位置)可调整为不交叉的路径.

[**证**] 因两只象视为相同,则在路径即将发生交叉时,可视为要移动的象A的灵魂传给了在其下一步到达的位置上的象B,

随后象B代替A到达目标位置,此时路径即可不交叉.

路径不交叉,则两只象的移动可视为独立,建图后用Floyd求出任意两个象可到达的位置间的最短路即可.时间复杂度 $O(7^3)$.

代码

```cpp
const int MAXN = 10;
int n = 7;  // 节点数
int dis[MAXN][MAXN];  // dis[u][v]表示节点u到节点v的最短路

void floyd() {
  for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++)
        dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
    }
  }
}

void init() {
  // 初始化
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      if (i == j) dis[i][j] = 0;
```

```
      else dis[i][j] = INF;
    }
  }

  // 建图
  dis[1][2] = dis[1][3] = 1;
  dis[2][1] = dis[2][4] = 1;
  dis[3][1] = dis[3][4] = 1;
  dis[4][2] = dis[4][3] = dis[4][5] = dis[4][6] = 1;
  dis[5][4] = dis[5][7] = 1;
  dis[6][4] = dis[6][7] = 1;
  dis[7][5] = dis[7][6] = 1;

  floyd();  // 预处理出任意两节点间的最短路
}

void solve() {
  init();

  CaseT{
    int a1, a2, b1, b2; cin >> a1 >> a2 >> b1 >> b2;
    int ans = dis[a1][b1] + dis[a2][b2];
    ans = min(ans, dis[a1][b2] + dis[a2][b1]);
    cout << ans << endl;
  }
}

int main() {
  solve();
}
```

14.17 最小生成树的应用

14.17.1 Nucleic Acid Test

题意

给定一个包含n个节点和m条边的无向图,其中k $(1 \le k \le n)$个节点处有核酸点.选择一个核酸点出发,以速度v遍历所有节点,并最后在核酸点结束,要求相邻两次核酸间隔不超过时间t,求满足要求的最小速度(整数).

第一行输入三个整数n, m, k $\left(2 \le n \le 300, 0 \le m \le \dfrac{n(n-1)}{2}, 1 \le k \le n\right)$.第二行输入一个整数$t$ $(0 \le t \le 1\text{e}9)$.接下来m行每行输入三个整数a, b, c $(1 \le a, b \le n, 1 \le c \le 1\text{e}9)$,表示节点$a$与$b$间存在长度为$c$的无向边.数据保证无重边.最后一行输入$k$个相异的整数$s_1, \cdots, s_k$,表示$k$个核酸点所在的节点编号.

若能从一个核算点出发,遍历所有节点,并最后在核酸点结束,输出最小速度(整数);否则输出-1.

思路I

n最大300,显然可先用Floyd算法求出任意两点间的最短路.设非核酸点x到核酸点A和B的距离分别为d_A和d_B,不妨设$d_A > d_B$.若从A出发经x到B,经过的路程为$d_A + d_B$,造成的影响是x被访问.注意到为访问x,只需从B出发,先走到x,再返回B,经过的路程为$2d_B < d_A + d_B$,这表明:非核酸点对总路程的贡献是它到最近核酸点的距离的两倍.将核酸点和非核酸点分开考虑,则非核酸点与距其最近的核酸点可能将图分为若干个连通块,此时显然核酸点间通过最小生成树连通答案最优,且其对答案的影响取决于生成树中的最长边.

注意$t = 0$或图不连通时无解.

代码I

```
const int MAXN = 305, MAXM = MAXN * MAXN;
int n, m, k;  // 节点数、边数、核酸点数
int t;  // 最大核酸间隔
int nucleic[MAXN];  // 核酸点编号
bool is_nucleic[MAXN];  // 记录每个节点是否是核酸点

namespace Floyd {
  int n;  // 节点数
  ll d[MAXN][MAXN];  // d[u][v]表示节点u与v间的最短路

  void init() {  // 初始化d[][]
    for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++) d[i][j] = i == j ? 0 : INFF;
  }

  void floyd() {
    for (int k = 1; k <= n; k++) {
      for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++) d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
    }
  }
}

namespace Kruskal {
  int n, m;  // 节点数、边数
  struct Edge {
    int u, v;
    ll w;

    bool operator<(const Edge& B) { return w < B.w; }
  }edges[MAXM];
  int fa[MAXN];  // 并查集的fa[]数组

  void init() {  // 初始化fa[]
    for (int i = 1; i <= n; i++) fa[i] = i;
  }

  int find(int x) { return x == fa[x] ? x : fa[x] = find(fa[x]); }

  ll kruskal() {  // 返回最小生成树的最长边,图不连通时返回INFF
    sort(edges, edges + m);  // 按边权升序排列

    ll res = 0;  // 最小生成树的最长边
    int cnt = 0;  // 当前连的边数
    for (int i = 0; i < m; i++) {
      auto [u, v, w] = edges[i];
      u = find(u), v = find(v);
      if (u != v) {
        fa[u] = v;
        res = max(res, w);
        cnt++;
      }
```

```
    }

    if (cnt < n - 1) return INFF;  // 图不连通
    else return res;
  }
}

void build() {  // 对核酸点建图
  Kruskal::m = 0;
  for (int i = 1; i <= k; i++) {
    for (int j = i + 1; j <= k; j++)
      Kruskal::edges[Kruskal::m++] = { nucleic[i],nucleic[j],Floyd::d[nucleic[i]]
[nucleic[j]] };
  }
}

void solve() {
  cin >> n >> m >> k >> t;

  if (!t) {
    cout << -1;
    return;
  }

  Floyd::n = n;
  Floyd::init();

  while (m--) {
    int a, b, c; cin >> a >> b >> c;
    Floyd::d[a][b] = Floyd::d[b][a] = min(Floyd::d[a][b], (ll)c);
  }

  for (int i = 1; i <= k; i++) {
    cin >> nucleic[i];
    is_nucleic[nucleic[i]] = true;
  }

  Floyd::floyd();

  ll maxlength = 0;  // 每个节点到其最近的核酸点的最短距离的最大值
  for (int i = 1; i <= n; i++) {
    ll minlength = INFF;  // 每个节点到其最近的核酸点的最短距离
    if (is_nucleic[i]) {  // 核酸点
      for (int j = 1; j <= k; j++) {
        if (nucleic[j] == i) continue;

        minlength = min(minlength, Floyd::d[i][nucleic[j]]);
      }

      if (k == 1) minlength = 0;  // 特判只有一个核酸点的情况
    }
    else {  // 非核酸点
      for (int j = 1; j <= k; j++)  // 找到距该非核酸点最近的核酸点
        minlength = min(minlength, Floyd::d[i][nucleic[j]] << 1);
    }
    maxlength = max(maxlength, minlength);
  }

```

```
  Kruskal::n = n;
  Kruskal::init();  // 注意并查集要对n个节点都初始化
  Kruskal::n = k;  // 实际图中只有k个节点
  build();
  maxlength = max(maxlength, Kruskal::kruskal());

  if (maxlength == INFF) cout << -1;
  else cout << (maxlength + t - 1) / t;
}

int main() {
  solve();
}
```

思路II

二分速度v,每次只对距离不超过vt的两节点连边,检查图的连通性即可.

代码II

```
const int MAXN = 305, MAXM = MAXN * MAXN;
int n, m, k;  // 节点数、边数、核酸点数
int t;  // 最大核酸间隔
bool is_nucleic[MAXN];  // 记录每个节点是否是核酸点

namespace Floyd {
  int n;  // 节点数
  ll d[MAXN][MAXN];  // d[u][v]表示节点u与v间的最短路

  void init() {  // 初始化d[][]
    for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++) d[i][j] = i == j ? 0 : INFF;
  }

  void floyd() {
    for (int k = 1; k <= n; k++) {
      for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++) d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
    }
  }
}

namespace DSU {
  int n;  // 元素个数
  int fa[MAXN];  // fa[]数组
  int siz[MAXN];  // 集合大小

  void init() {  // 初始化fa[]、siz[]
    for (int i = 1; i <= n; i++) fa[i] = i, siz[i] = 1;
  }

  int find(int x) { return x == fa[x] ? x : fa[x] = find(fa[x]); }

  bool merge(int x, int y) {  // 返回合并是否成功,即初始时是否不在同一集合中
    x = find(x), y = find(y);
    if (x != y) {
```

```
      if (siz[x] > siz[y]) swap(x, y);  // 保证x所在的集合小

      fa[x] = y;
      siz[y] += siz[x];
      return true;
    }
    else return false;
  }
}

bool vis[MAXN];  // 记录每个集合是否被遍历过

bool check(ll v) {
  DSU::init();
  for (int i = 1; i <= n; i++) vis[i] = false;

  // 核酸点间连边
  for (int i = 1; i <= n; i++) {
    if (!is_nucleic[i]) continue;

    for (int j = 1; j <= n; j++) {
      if (!is_nucleic[j]) continue;

      // if (v * t >= Floyd::d[i][j]) DSU::merge(i, j);  // 注意此处乘法会爆ll
      if ((double)Floyd::d[i][j] / v <= (double)t) DSU::merge(i, j);
    }
  }

  // 非核酸点向核酸点连边
  for (int i = 1; i <= n; i++) {
    if (is_nucleic[i]) continue;

    for (int j = 1; j <= n; j++) {
      if (!is_nucleic[j]) continue;

      // if (v * t >= 2 * Floyd::d[i][j]) DSU::merge(i, j);  // 注意此处乘法会爆ll
      if ((double)2 * Floyd::d[i][j] / v <= (double)t) DSU::merge(i, j);
    }
  }

  int cnt = 0;  // 连通块个数
  for (int i = 1; i <= n; i++) {
    int j = DSU::find(i);
    if (!vis[j]) {
      cnt++;
      vis[j] = true;

      if (cnt >= 2) return false;  // 图不连通
    }
  }
  return true;
}

void solve() {
  cin >> n >> m >> k >> t;

  if (!t) {
    cout << -1;
```

```
    return;
  }

  Floyd::n = n;
  Floyd::init();

  while (m--) {
    int a, b, c; cin >> a >> b >> c;
    Floyd::d[a][b] = Floyd::d[b][a] = min(Floyd::d[a][b], (ll)c);
  }

  for (int i = 1; i <= k; i++) {
    int x; cin >> x;
    is_nucleic[x] = true;
  }

  Floyd::floyd();

  // 检查图的连通性
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      if (Floyd::d[i][j] >= INFF) {  // 图不连通
        cout << -1;
        return;
      }
    }
  }

  DSU::n = n;

  ll l = 1, r = 1e18;  // 注意l最小取1
  while (l < r) {
    ll mid = l + r >> 1;
    if (check(mid)) r = mid;
    else l = mid + 1;
  }
  cout << l;
}

int main() {
  solve();
}
```

14.17.2 Avoid Rainbow Cycles

原题指路: https://codeforces.com/problemset/problem/1408/E

题意

给定m个集合$A_1, \cdots, A_m$, 集合中的元素为整数, 且在$[1, n]$范围内. 给定两个正整数序列$a = [a_1, \cdots, a_m]$和$b = [b_1, \cdots, b_n]$. 每次操作可删除集合A_i中的一个元素j, 花费为$(a_i + b_j)$. 现做若干次(可能为0次)操作后建立一张包含编号$1 \sim n$的n个节点的无向图, 具体地, 对每个集合A_i, 若$\exists x, y \in A_i \ s.t. \ x < y$, 则加一条边$(x, y)$, 其颜色为$i$. 称图中的一个环是坏的, 如果其上的所有边异色. 注意环可以是二元环, 即两个节点间有异色的重边. 问使得图中无坏的环所需的最小花费.

第一行输入两个整数m, n $(1 \le m, n \le 1e5)$. 第二行输入m个整数$a_1, \cdots, a_m$ $(1 \le a_i \le 1e9)$. 第三行输入n个整数$b_1, \cdots, b_n$ $(1 \le b_i \le 1e9)$. 接下来m行每行输入若干个整数, 其中第i $(1 \le i \le m)$行先输入一个整数s_i $(1 \le s_i \le n)$, 表示集合A_i中的元素个数. 接下来输入s_i个相异的、在$[1, n]$范围内的整数, 表示A_i中的元素个数. 数据保证所有集合的元素个数之和不超过$2e5$.

思路

直接建图有$O(nm)$条边, 是$1e10$的级别. 注意到是否成环只与节点间的连通性有关, 考虑建立虚节点来减少图的边数. 具体地, 用m个节点分别代表m个集合, 对$\forall x \in A_i$. 对A_i对应的节点p_i和$x, y \in A_i$, 在原图中节点x与节点y直接连通转化为节点x先与p_i连通, p_i再与y连通, 此时图中有$O(n + m)$条边. 边权上, x与p_i连一条边权为$(a_i + x)$的边, 即删除元素x的花费.

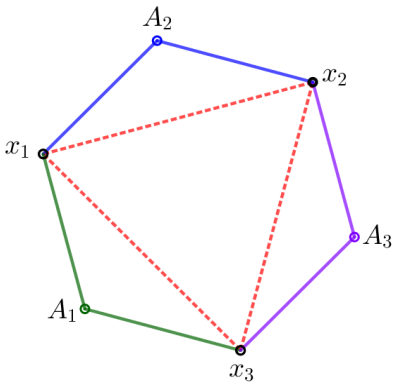
[**定理**] 原图中的坏的环与新图中的环一一对应.

[**证**] 显然原图中的坏的环对应于新图中的环, 下证新图中的环对应于原图中的环.

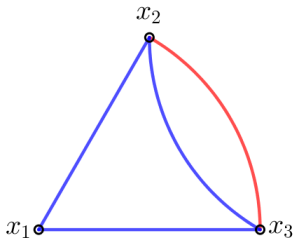
设第i个集合对应的节点为A_i, 元素$1 \sim n$对应的节点为x_i,

则新图中的环$A_1 \to x_1 \to A_2 \to x_2 \to \cdots \to A_k \to x_k \to A_1$对应于原图中坏的环$x_1 \to \cdots \to x_k \to x_1$.

$k = 3$时的情况如下图所示, 其中新图的环(绿色、蓝色、紫色)与原图中坏的环(红色)一一对应.



如下图, 考察是否会将原图中的环$x_1 \to x_2 \to x_3 \to x_1$误判为一个坏的环.



注意到存在边$(x_1, x_2), (x_2, x_3)$表明x_1, x_2, x_3是某集合A_i中的元素, 而集合元素有互异性, 故$x_2 \neq x_3$, 进而也存在边(x_2, x_3), 进而与新图中的环一一对应的是原图中的坏的二元环$x_2 \to x_3 \to x_2$.

删除集合中的元素等价于删除新图中的边, 问题转化为删除若干边使得新图中无环, 且删除的边的边权之和最小. 注意到删除的边权和最小等价于在无环的前提下新图剩下的边权和最大, 问题转化为求新图的最大生成森林.

代码

```cpp
struct Kruskal {
  const ll INFF = 0x3f3f3f3f3f3f3f3f;

  int n, m;  // 节点数、边数
  struct Edge {
```

```
    int u, v, w;

    bool operator<(const Edge& p) const {
      return w > p.w;
    }
  };
  vector<Edge> edges;
  vector<int> fa;

  Kruskal(int  _n) :n(_n), m(0) {
    fa.resize(n + 1);
    iota(all(fa), 0);
  }

  int find(int x) {
    return x == fa[x] ? x : fa[x] = find(fa[x]);
  }

  ll kruskal() {  // 求MST,返回MST的边权和
    sort(all(edges));

    ll res = 0;  // MST的边权和
    int cnt = 0;  // 当前连的边数
    for (int i = 0; i < m; i++) {
      auto [u, v, w] = edges[i];

      u = find(u), v = find(v);
      if (u != v) {
        fa[u] = v;
        res += w;
        cnt++;
      }
    }
    // return cnt < n - 1 ? INFF : res;
    return res;  // 求生成森林无需检查连通性
  }
};

void solve() {
  int m, n; cin >> m >> n;
  vector<int> a(m + 1), b(n + 1);
  for (int i = 1; i <= m; i++) cin >> a[i];
  for (int i = 1; i <= n; i++) cin >> b[i];

  Kruskal solver(n + m);
  ll sum = 0;  // 边权之和
  for (int i = 1; i <= m; i++) {  // 集合
    int s; cin >> s;
    while (s--) {
      int j; cin >> j;

      // 元素对应的节点向集合对应的节点连边
      int w = a[i] + b[j];
      solver.edges.push_back({ j + m, i, w });
      solver.m++;
      sum += w;
    }
  }
```

```
  cout << sum - solver.kruskal() << endl;
}

int main() {
  solve();
}
```

14.17.3 Road Reform

原题指路: https://codeforces.com/problemset/problem/1468/J

题意 (2 s)

有 t $(1 \le t \le 1000)$ 组测试数据. 每组测试数据给定一张包含编号 $1 \sim n$ 的 n $(2 \le n \le 2e5)$ 个节点和编号 $1 \sim m$ 的 m $\left(n-1 \le m \le \min\left\{\frac{n(n-1)}{2}, 2e5\right\}\right)$ 条边的无向连通图, 其中边权 $w_i \in [1, 1e9]$ $(1 \le i \le n)$. 现保留该无向图的一个子图, 使得该子图是一棵树. 对保留的树, 有操作: 选择一条边, 将其边权 $+1$ 或 -1 . 给定一个整数 k $(1 \le k \le 1e9)$, 求使得保留的树的最大边权恰为 k 的最小操作次数. 数据保证所有测试数据的 n 之和、 m 之和都不超过 $2e5$.

思路

先不考虑边权 $w_i > k$ 的边, 若只连边权 $w_i \le k$ 的边即可使得该图连通, 则该图存在不含边权 $w_i > k$ 的边的生成树. 但要求保留的边的最大值恰为 k , 则上述方案可能非最优, 进而应恰保留一条边权最接近 k 的边, 即 $ans = \min\limits_{1 \le i \le m} |w_i - k|$.

若只连边权 $w_i \le k$ 的边不足以使图连通, 则还需加入边权 $w_i > k$ 的边. 将 i 号边的边权改为 $\max\{w_i - k, 0\}$, 求该图的MST, 则 ans 为MST的权值.

总时间复杂度 $O(m \log m)$.

代码

```
const ll INFF = 0x3f3f3f3f3f3f3f3f;

struct DSU {
    int n;
    vector<int> fa;

    DSU(int _n) :n(_n) {
        fa.resize(n + 1);
        iota(all(fa), 0);
    }

    int find(int x) {
        return x == fa[x] ? x : fa[x] = find(fa[x]);
    }

    bool merge(int x, int y) {
        if ((x = find(x)) == (y = find(y))) return false;

        fa[x] = y;
        return true;
    }

```

```
    int count() {
        int res = 0;
        for (int i = 1; i <= n; i++)
            res += fa[i] == i;
        return res;
    }
};

struct Kruskal {
    int n;
    vector<tuple<int, int, int>> edges;
    vector<int> fa;

    Kruskal(int _n, const vector<tuple<int, int, int>>& _edges) :n(_n), edges(_edges) {
        fa.resize(n + 1);
        iota(all(fa), 0);
        sort(all(edges));
    }

    int find(int x) {
        return x == fa[x] ? x : fa[x] = find(fa[x]);
    }

    ll kruskal() {
        ll res = 0;
        int cnt = 0;  // 当前连的边数

        for (auto [w, u, v] : edges) {
            if ((u = find(u)) != (v = find(v))) {
                fa[u] = v;
                res += w, cnt++;
            }
        }
        return cnt == n - 1 ? res : INFF;
    }
};

void solve() {
    int n, m, k; cin >> n >> m >> k;
    vector<tuple<int, int, int>> edges;  // w, u, v
    for (int i = 0; i < m; i++) {
        int u, v, w; cin >> u >> v >> w;
        edges.push_back({ w, u, v });
    }
    sort(all(edges));

    // 先连边权 <= k 的边
    int idx;  // 当前用到的边的编号
    DSU dsu(n);
    for (idx = 0; idx < m; idx++) {
        int w = get<0>(edges[idx]), u = get<1>(edges[idx]), v = get<2>(edges[idx]);
        if (w > k) break;

        dsu.merge(u, v);
    }

    if (dsu.count() == 1) {  // 仅连边权 <= k 的边即可使图连通
        int ans = INF;
```

```
        for (int i = 0; i < m; i++)
            ans = min(ans, abs(get<0>(edges[i]) - k));
        cout << ans << endl;
        return;
    }

    for (int i = 0; i < m; i++) {
        int& w = get<0>(edges[i]);
        w = max(w - k, 0);
    }

    Kruskal solver(n, edges);
    cout << solver.kruskal() << endl;
}

int main() {
    CaseT
    solve();
}
```

14.18 LCA

在有根树中,每个节点到根节点的简单路径上的节点称为其祖先(自己也称为自己的祖先).对两个节点,它们可能有若干个公共祖先,其中里根节点最远的公共祖先称为两节点的最近公共祖先,简称LCA(Lowest Common Ancestor).

节点u与v的LCA的求法:

(1)[**向上标记法**] 先标记节点u到根节点的路径上的所有节点,再从节点v开始向上跳,直至跳到第一个有标记的节点.最坏情况是一条链,时间复杂度$O(n)$,不常用.

(2)[**倍增法**] BFS预处理出每个节点的深度 $depth[]$ 和数组 $fa[][]$,其中 $fa[i][j]$ 表示从节点 i 开始向上跳 2^j 步到达的节点,$0 \le j \le \lfloor \log_2 n \rfloor$,不存在的节点定义为 $\varnothing$. 预处理时间复杂度$O(n \log n)$,查询一次时间复杂度$O(\log n)$,常用又好写.

$fa[][]$可递推求出:①$j = 0$时,$fa[i][j] = father[i]$;②$j > 0$时,跳两次2^{j-1}步,即$fa[fa[i][j-1]][j-1]$.

若节点i向上跳2^j步会跳出根节点,则规定$fa[i][j] = depth[i] = 0$.此时因树的节点u都满足$depth[u] \ge 1$,故循环条件不成立.

求LCA步骤:

①不妨设$depth[u] > depth[v]$.先将较深的节点u向上跳至与v同一深度,即使得$depth[u] = depth[v]$.设$tmp = depth[u] - depth[v]$,从高到低考察其二进制数位,$u$向上跳$tmp$的二进制为一的数位对应的$fa[][]$即可.实现时无需考察二进制数位,只需检查是否满足$depth[fa[u][k]] > depth[v]$,若满足则向上跳,时间复杂度$O(\log n)$.

②若此时$u = v$则退出;否则两节点同时以类似①的方式向上跳,直至到达它们的LCA的下一层,即跳至它们是LCA的父亲节点为止,此时LCA即$fa[u][0] = fa[v][0]$,时间复杂度$O(\log n)$.

应跳到LCA的下一层而不是直接跳到LCA的原因:若$fa[u][k] = fa[v][k]$,只能说明它们跳到的点是一个公共祖先,但未必是LCA.

(3)[**Tarjan离线求LCA算法**] 考虑优化向上标记法.在DFS过程中将所有节点分为三类:①已搜过且回溯过的节点,这样的节点的所有子树都已搜过,标记为2;②正在搜索的分支,标记为1;③还未搜索到的点,标记为0.

考察询问中哪些询问是求标记为1的点x与标记为2的点y的LCA,注意到它们的LCA都是标记1的链上的节点,即标记为2的子树的根节点,故可用并查集将标记为2的子树的点合并到其根节点上.每个节点只会被遍历一次、合并一次,每个询问只会被查询一次,并查集的合并和查询都是$O(1)$,故总时间复杂度$O(n+m)$,其中n为节点数,m为询问数.

(4)[**RMQ求LCA**] 遍历过程中记录DFS序列(每个点入栈出栈都记录).$\mathrm{LCA}(u,v)$是DFS序列中u与v间深度最小的点,转化为离线求区间最小值问题,不常用.

14.18.1 祖孙询问

题意

给定一棵包含n个节点的有根无向树,节点编号互不相同,但未必是$1\sim n$.有m个询问,每次询问给定一对节点的编号x,y,表示询问节点x与y的祖孙关系.

第一行输入一个整数n $(1\le n\le 4\mathrm{e}4)$.接下来n行每行输入两个整数a,b $(1\le a,b\le 4\mathrm{e}4)$,表示节点$a$与节点$b$间有无向边,若$b=-1$表示节点$a$为根.接下来一行输入一个整数$m$ $(1\le m\le 4\mathrm{e}4)$表示询问次数.接下来m行每行输入两个相异的整数x,y $(1\le x,y\le 4\mathrm{e}4)$,表示询问节点$x$与节点$y$的祖孙关系.

对每个询问,若x是y的祖先,则输出1;若y是x的祖先,则输出2;否则输出0.

思路

倍增求LCA.

代码

```cpp
const int MAXN = 4e4 + 5;

struct RMQLCA {
    const int MAXJ;
    int n;
    vector<vector<int>> edges;
    vector<int> depth;
    vector<vector<int>> fa;

    RMQLCA(int _n, const vector<vector<int>>& _edges)
        :n(_n), edges(_edges), MAXJ(log2(_n) + 1) {
        depth = vector<int>(n + 1, INF);
        fa = vector<vector<int>>(n + 1, vector<int>(MAXJ + 1));
    }

    void bfs(int root) {  // 预处理depth[], fa[][]: 根节点为root
        depth[0] = 0, depth[root] = 1;

        queue<int> que;
        que.push(root);
        while (que.size()) {
            auto u = que.front(); que.pop();
            for (auto v : edges[u]) {
                if (depth[v] > depth[u] + 1) {
                    depth[v] = depth[u] + 1;
                    que.push(v);

                    fa[v][0] = u;
                    for (int k = 1; k <= MAXJ; k++)
                        fa[v][k] = fa[fa[v][k - 1]][k - 1];
                }
            }
        }
    }
```

```
35
36      int lca(int u, int v) {
37          if (depth[u] < depth[v]) swap(u, v);  // 保证节点u深度大
38
39          // 将节点u与节点v跳到同一深度
40          for (int k = MAXJ; k >= 0; k--)
41              if (depth[fa[u][k]] >= depth[v]) u = fa[u][k];
42
43          if (u == v) return u;  // u与v原本在一条链上
44
45          // u与v同时跳到LCA的下一层
46          for (int k = MAXJ; k >= 0; k--) {
47              if (fa[u][k] != fa[v][k])
48                  u = fa[u][k], v = fa[v][k];
49          }
50          return fa[u][0];  // 节点u向上跳1步到LCA
51      }
52  };
53
54  void solve() {
55      int n; cin >> n;
56      vector<vector<int>> edges(MAXN);
57      int root;
58      for (int i = 0; i < n; i++) {
59          int u, v; cin >> u >> v;
60
61          if (~v) edges[u].push_back(v), edges[v].push_back(u);
62          else root = u;
63      }
64
65      RMQLCA solver(MAXN, edges);
66    solver.bfs(root);
67
68    CaseT {
69      int u, v; cin >> u >> v;
70
71          int lca = solver.lca(u, v);
72      if (lca == u) cout << 1 << endl;
73      else if (lca == v) cout << 2 << endl;
74      else cout << 0 << endl;
75    }
76  }
77
78  int main() {
79      solve();
80  }
```

14.18.2 距离

题意

给定一棵包含编号$1 \sim n$的n个节点的无向树,多次询问两点间的最短距离.

第一行输入两个整数n, m $(2 \leq n \leq 1e4, 1 \leq m \leq 2e4)$,分别表示节点数和询问数.接下来$(n-1)$行每行输入三个整数$x, y, w$ $(1 \leq x, y \leq n, 0 \leq w \leq 100)$,表示节点$x$与$y$间有一条长度为$w$的无向边.接下来$m$行每行输入两个整数$x, y$ $(1 \leq x, y \leq n)$,表示询问节点x与y的最短距离.

思路

Tarjan离线求LCA算法.

节点x与节点y的最短距离为$depth[x] + depth[y] - 2 \cdot depth[\mathrm{LCA}(x, y)]$.

代码

```cpp
const int MAXN = 1e4 + 5, MAXM = MAXN << 1;
int n, m;  // 节点数、询问数
int head[MAXN], edge[MAXM], w[MAXM], nxt[MAXM], idx;
int dis[MAXN];  // 节点到根节点的距离
int fa[MAXN];  // 并查集的fa[]数组
int state[MAXN];  // tarjan算法中记录每个节点的状态,1表示正在搜的分支,2表示已搜过且回溯过的节点,0表示还未搜的节点
vii querys[MAXN];  // querys[a]={b, i},表示第i个询问为求dis(a,b)
int ans[MAXM];  // 每个询问的答案

void add(int a, int b, int c) {
  edge[idx] = b, w[idx] = c, nxt[idx] = head[a], head[a] = idx++;
}

void dfs(int u, int fa) {  // 预处理出dis[]数组:u为当前节点,其前驱节点为fa
  for (int i = head[u]; ~i; i = nxt[i]) {
    int j = edge[i];
    if (j == fa) continue;

    dis[j] = dis[u] + w[i];
    dfs(j, u);
  }
}

int find(int x) { return x == fa[x] ? x : fa[x] = find(fa[x]); }

void tarjan(int u) {  // 当前节点
  state[u] = 1;  // 正在搜该节点的子树
  for (int i = head[u]; ~i; i = nxt[i]) {
    int j = edge[i];
    if (!state[j]) {
      tarjan(j);
      fa[j] = u;
    }
  }

  for (auto [v, id] : querys[u]) {
    if (state[v] == 2) {
      int lca = find(v);
      ans[id] = dis[u] + dis[v] - dis[lca] * 2;
    }
  }
```

```
42
43    state[u] = 2;
44  }
45
46  void solve() {
47    memset(head, -1, so(head));
48
49    cin >> n >> m;
50
51    for (int i = 1; i <= n; i++) fa[i] = i;  // 初始化并查集的fa[]数组
52
53    for (int i = 0; i < n - 1; i++) {
54      int a, b, c; cin >> a >> b >> c;
55      add(a, b, c), add(b, a, c);
56    }
57
58    for (int i = 0; i < m; i++) {
59      int a, b; cin >> a >> b;
60      if (a != b) {
61        querys[a].push_back({ b,i });
62        querys[b].push_back({ a,i });
63      }
64    }
65
66    dfs(1, -1);  // 从根节点开始搜,根节点无前驱节点
67    tarjan(1);
68
69    for (int i = 0; i < m; i++) cout << ans[i] << endl;
70  }
71
72  int main() {
73    solve();
74  }
```

14.18.3 严格次小生成树

题意

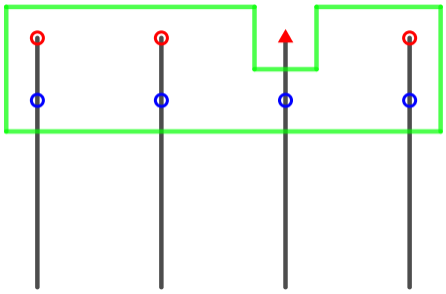
给定一张包含n个点m条边的无向图,求其严格次小生成树.设最小生成树的边权之和为sum,则严格次小生成树是边权之和$>$ sum的生成树中边权之和最小的.

第一行输入两个整数n, m $(1 \leq n \leq 1e5, 1 \leq m \leq 3e5)$.接下来$m$行每行输入三个整数$x, y, z$ $(1 \leq x, y \leq n)$,表示节点x与y间存在一条权值为z的边.

输出严格次小生成树的边权和.数据保证严格次小生成树存在.

思路

在倍增求LCA的基础上,预处理出$dis1[i][j]$和$dis2[i][j]$分别表示节点i向上跳2^j步的路径上的最大(下图中的红色)、次大(下图中的蓝色)边权.



所有边权的最大值时所有$dis1[][]$和$dis2[][]$取max,不妨设最大值为上图中红色三角形的值,则所有边权的次大值是绿色框中的值的max.与递推求$fa[][]$类似,递推时同时维护$dis1[][]$和$dis2[][]$即可.

代码

```cpp
const int MAXN = 1e5 + 5, MAXM = 3e5 + 5, MAXJ = 16;  // 16=floor(log2(MAXN))
int n, m;  // 节点数、边数
struct Edge {
  int u, v, w;
  bool used;  // 表示每条边是否在MST中

  bool operator<(const Edge& t)const { return w < t.w; }
}edges[MAXM];
int dsu[MAXN];  // 并查集的fa[]数组
int head[MAXN], edge[MAXM], w[MAXM], nxt[MAXM], idx;
int depth[MAXN];  // 节点的深度
int fa[MAXN][MAXJ + 1];  // fa[i][j]表示节点i向上跳2^j步到达的节点
// dis1[i][j]和dis2[i][j]分别表示节点i向上跳2^j步的路径上的最大、次大边权
int dis1[MAXN][MAXJ + 1], dis2[MAXN][MAXJ + 1];

int find(int x) { return x == dsu[x] ? x : dsu[x] = find(dsu[x]); }

ll kruskal() {
  for (int i = 1; i <= n; i++) dsu[i] = i;  // 初始化dsu[]

  sort(edges, edges + m);
  ll res = 0;
  for (int i = 0; i < m; i++) {
    int u = find(edges[i].u), v = find(edges[i].v), w = edges[i].w;
    if (u != v) {
      dsu[u] = v;
      res += w;
      edges[i].used = true;  // 记录该边在MST中
    }
  }
  return res;
}

void add(int a, int b, int c) {
  edge[idx] = b, w[idx] = c, nxt[idx] = head[a], head[a] = idx++;
}

void build() {  // 建出MST的图
  memset(head, -1, so(head));

  for (int i = 0; i < m; i++) {
    if (edges[i].used) {
      int u = edges[i].u, v = edges[i].v, w = edges[i].w;
```

```
      add(u, v, w), add(v, u, w);
    }
  }
}

void bfs() {  // 预处理depth[]、fa[][]、dis1[][]、dis2[][]
  memset(depth, INF, so(depth));
  depth[0] = 0, depth[1] = 1;

  qi que;
  que.push(1);
  while (que.size()) {
    auto tmp = que.front(); que.pop();
    for (int i = head[tmp]; ~i; i = nxt[i]) {
      int j = edge[i];
      if (depth[j] > depth[tmp] + 1) {
        depth[j] = depth[tmp] + 1;
        que.push(j);
        fa[j][0] = tmp;  // 向上跳1步
        dis1[j][0] = w[i], dis2[j][0] = -INF;  // 无次大值,初始化为-INF
        for (int k = 1; k <= MAXJ; k++) {
          int mid = fa[j][k - 1];  // 跳到中间的节点
          fa[j][k] = fa[mid][k - 1];  // 中间这个点再跳2^{j-1}步
          int distance[4]  // 每条路径上的最大值和次大值
            = { dis1[j][k - 1],dis2[j][k - 1],dis1[mid][k - 1],dis2[mid][k - 1] };
          dis1[j][k] = dis2[j][k] = -INF;
          for (int u = 0; u < 4; u++) {
            int d = distance[u];
            if (d > dis1[j][k]) dis2[j][k] = dis1[j][k], dis1[j][k] = d;  // 更新最大值、次大值
            else if (d != dis1[j][k] && d > dis2[j][k]) dis2[j][k] = d;  // 更新次大值
          }
        }
      }
    }
  }
}

int LCA(int a, int b, int w) {  // 求节点a和b的LCA,返回w-边权最大值或w-边权次大值
  if (depth[a] < depth[b]) swap(a, b);  // 保证a在下

  static int distance[MAXN << 1];  // 存最大值和次大值
  int cnt = 0;  // distance[]当前用到的下标
  for (int k = MAXJ; k >= 0; k--) {  // a跳到b的同层
    if (depth[fa[a][k]] >= depth[b]) {
      distance[cnt++] = dis1[a][k];
      distance[cnt++] = dis2[a][k];
      a = fa[a][k];
    }
  }

  if (a != b) {  // 若还不是LCA
    for (int k = MAXJ; k >= 0; k--) {  // 跳到LCA的下一层
      if (fa[a][k] != fa[b][k]) {
        distance[cnt++] = dis1[a][k];
        distance[cnt++] = dis2[a][k];
        distance[cnt++] = dis1[b][k];
        distance[cnt++] = dis2[b][k];
```

```
        a = fa[a][k], b = fa[b][k];
      }
    }

    // 加上跳到LCA的边权
    distance[cnt++] = dis1[a][0];
    distance[cnt++] = dis1[b][0];
  }

  int dist1 = -INF, dist2 = -INF;  // 所有边权的最大值、次大值
  for (int i = 0; i < cnt; i++) {
    int d = distance[i];
    if (d > dist1) dist2 = dist1, dist1 = d;  // 更新最大值、次大值
    else if (d != dist1 && d > dist2) dist2 = d;  // 更新次大值
  }

  if (w > dist1) return w - dist1;
  if (w > dist2) return w - dist2;
  return INF;  // 这条边不应用
}

void solve() {
  cin >> n >> m;
  for (int i = 0; i < m; i++) {
    int a, b, c; cin >> a >> b >> c;
    edges[i] = { a,b,c };
  }

  ll sum = kruskal();  // MST的边权和
  build();
  bfs();

  ll ans = INFF;
  for (int i = 0; i < m; i++) {
    if (!edges[i].used) {  // 只考虑不在MST中的边
      int u = edges[i].u, v = edges[i].v, w = edges[i].w;
      ans = min(ans, sum + LCA(u, v, w));
    }
  }
  cout << ans;
}

int main() {
  solve();
}
```

14.18.4 Fools and Roads

原题指路:https://codeforces.com/problemset/problem/191/C

题意 (2 s)

给定一棵包含编号$1 \sim n$的n个节点和$(n-1)$条边的树,初始时每条边的边权都为0.现有m $(0 \leq m \leq 1e5)$次操作,每次操作给定两个节点u, v $(1 \leq u, v \leq n)$,表示给节点u到节点v的简单路径上的边的边权都$+1$.操作后,按输入的边的顺序输出每条边的边权.

思路

将边的边权绑定在边的深度较大的节点上,树上差分即可.

代码

```cpp
const int MAXN = 1e5 + 5, MAXJ = 16;  // MAXJ=floor(log(MAXN))
namespace RMQLCA {
  pair<int, int> edge[MAXN];
  vector<int> edges[MAXN];
  int depth[MAXN];  // 节点深度
  int fa[MAXN][MAXJ + 1];  // fa[i][j]表示节点i向上跳2^j步到达的节点
  int diff[MAXN];  // 差分数组,DFS后diff[u]表示以节点u为根节点的子树中的点权和

  void bfs() {  // 预处理depth[]和fa[][]
    memset(depth, INF, sizeof(depth));
    depth[0] = 0, depth[1] = 1;

    queue<int> que;
    que.push(1);
    while (que.size()) {
      auto u = que.front(); que.pop();
      for (auto v : edges[u]) {
        if (depth[v] > depth[u] + 1) {
          depth[v] = depth[u] + 1;
          que.push(v);
          fa[v][0] = u;
          for (int k = 1; k <= MAXJ; k++)
            fa[v][k] = fa[fa[v][k - 1]][k - 1];
        }
      }
    }
  }

  int lca(int u, int v) {
    if (depth[u] < depth[v]) swap(u, v);  // 保证节点u深度大

    for (int k = MAXJ; k >= 0; k--)  // 节点u跳到与节点v同一深度
      if (depth[fa[u][k]] >= depth[v]) u = fa[u][k];

    if (u == v) return u;

    for (int k = MAXJ; k >= 0; k--)  // 节点u与节点v同时向上跳至LCA的下一层
      if (fa[u][k] != fa[v][k]) u = fa[u][k], v = fa[v][k];
    return fa[u][0];  // 注意返回父亲节点
  }

  void modify(int u, int v) {  // 节点u到节点v的简单路径上的边权都+1
    diff[u]++, diff[v]++, diff[lca(u, v)] -= 2;
  }

```

```
  void dfs(int u, int pre) {  // 求以u为根节点的子树的点权和
    for (auto v : edges[u]) {
      if (v != pre) {
        dfs(v, u);
        diff[u] += diff[v];
      }
    }
  }
}
using namespace RMQLCA;

void solve() {
  int n; cin >> n;
  for (int i = 1; i < n; i++) {
    int u, v; cin >> u >> v;
    edge[i] = { u,v };
    edges[u].push_back(v), edges[v].push_back(u);
  }

  bfs();

  CaseT{
    int u, v; cin >> u >> v;
    modify(u, v);
  }

  dfs(1, 0);
  for (int i = 1; i < n; i++) {
    auto [u, v] = edge[i];
    cout << diff[depth[u] > depth[v] ? u : v] << " \n"[i == n - 1];
  }
}

int main() {
  solve();
}
```

14.18.5 闇の連鎖

题意

有一张包含编号$1 \sim n$的n个节点的无向图Dark,其中包含两种边,一种是主要边,一种是附加边.Dark有$(n-1)$条主要边,其任意两节点间都存在一条只由主要边构成的路径.此外Dark还有m条附加边.现要将Dark拆成不连通的两部分.初始时Dark的附加边都处于无敌状态,只能选取一条主要边切断.切断一条主要边后,所有主要边变为无敌状态,而附加边可被切断.求将Dark拆成不连通的两部分的方案数.注意若第一步切断一条主要边之后已将Dark拆成不连通的两部分,也要再切断一条附加边才算完成.

第一行输入两个整数n, m $(1 \leq n \leq 1e5, 1 \leq m \leq 2e5)$.接下来$(n-1)$行每行输入两个整数$a, b$ $(1 \leq a, b \leq n)$,表示节点a与b间存在主要边.接下来m行每行以相同形式描述无向边.

输出将Dark拆成不连通的两部分的方案数,数据保证答案不超过int范围.

思路

不考虑附加边时,Dark是一棵树,则主要边为树边,附加边为非树边.

对非树边与树边构成的环中的每条树边,若要使得切断该树边后图不连通,还应砍掉非树边.枚举每条非树边,给与其成环的树边权值$+1$,表示若要使得切断该树边后图不连通,还应在切断一条非树边.遍历一遍树边,设当前树边上的权值为c,则:①若$c=0$,则切断该树边后无需再切断非树边即可使图不连通,$ans+=m$;②若$c=1$,则切断该树边后还需切断1条非树边才能使图不连通,$ans++$;③若$c>1$,则切断该树边后还需切断c条非树边才能使图不连通,而每一次至多只能切断1条非树边,故该情况对答案无贡献.注意根节点无需计算.

考虑如何快速给树上两点间的路径上的边加上一个数.考虑树上差分.若要将节点x与y上的树边的权值$+=c$,可$diff[x]+=c,diff[y]+=c,diff[\mathrm{LCA}(x,y)]-=2c$,则边$<father[u],u>$的边权为以$u$为根节点的子树的点权之和.

代码

```cpp
const int MAXN = 1e5 + 5, MAXM = MAXN << 1, MAXJ = 16;  // 16=floor(log2(MAXN))
int n, m;  // 节点数、边数
int head[MAXN], edge[MAXM], nxt[MAXM], idx;
int diff[MAXN];  // 差分数组
int depth[MAXN];  // 节点的深度
int fa[MAXN][MAXJ + 1];  // fa[i][j]表示节点i向上跳2^j步到达的节点
int ans;

void add(int a, int b) {
  edge[idx] = b, nxt[idx] = head[a], head[a] = idx++;
}

void bfs() {  // 预处理depth[]、fa[][]
  memset(depth, INF, so(depth));
  depth[0] = 0, depth[1] = 1;

  qi que;
  que.push(1);
  while (que.size()) {
    auto tmp = que.front(); que.pop();
    for (int i = head[tmp]; ~i; i = nxt[i]) {
      int j = edge[i];
      if (depth[j] > depth[tmp] + 1) {
        depth[j] = depth[tmp] + 1;
        que.push(j);
        fa[j][0] = tmp;
        for (int k = 1; k <= MAXJ; k++)
          fa[j][k] = fa[fa[j][k - 1]][k - 1];
      }
    }
  }
}

int LCA(int a, int b) {  // 求节点a和b的LCA
  if (depth[a] < depth[b]) swap(a, b);

  for (int k = MAXJ; k >= 0; k--)
    if (depth[fa[a][k]] >= depth[b]) a = fa[a][k];

  if (a == b) return a;
```

```
41
42    for (int k = MAXJ; k >= 0; k--)
43      if (fa[a][k] != fa[b][k]) a = fa[a][k], b = fa[b][k];
44    return fa[a][0];
45  }
46
47  int dfs(int u, int pre) {  // 求以u为根节点的子树的点权和
48    int res = diff[u];
49    for (int i = head[u]; ~i; i = nxt[i]) {
50      int v = edge[i];
51      if (v != pre) {
52        int c = dfs(v, u);
53        if (!c) ans += m;
54        else if (c == 1) ans++;
55        res += c;
56      }
57    }
58    return res;
59  }
60
61  void solve() {
62    memset(head, -1, so(head));
63
64    cin >> n >> m;
65    for (int i = 0; i < n - 1; i++) {
66      int a, b; cin >> a >> b;
67      add(a, b), add(b, a);
68    }
69
70    bfs();
71
72    for (int i = 0; i < m; i++) {
73      int a, b; cin >> a >> b;
74      diff[a]++, diff[b]++, diff[LCA(a, b)] -= 2;  // 树上差分
75    }
76
77    dfs(1, -1);  // 从根节点开始搜,根节点无前驱节点
78    cout << ans;
79  }
80
81  int main() {
82    solve();
83  }
```

14.19 有向图的强连通分量

有向图的连通分量(Strongly Connected Components,SCC):设u、v是连通分量中的两节点,则可从u走到v,也可从v走到u.

有向图的强连通分量:极大连通分量,即所含节点数最多的连通分量,亦即再加一个点都不能构成连通分量.

缩点:将所有连通分量分别缩为一个点.

强连通分量用于将一个有向图通过缩点转化为DAG,这样求最短路或最长路时可按拓扑序逆推,时间复杂度$O(n+m)$.

在DFS生成树中将边分为四类:①树枝边:父节点指向其子节点,是特殊的前向边;②前向边:祖先节点指向其子孙节点;③后向边:子孙节点指向其祖先节点;④横插边:当前节点指向已搜过的节点.

若节点x在某个SCC中,有如下两种情况:①存在一条后向边指向x的祖先节点;②可从x先到达一条横插边,再到达祖先节点.故一个节点在某个SCC中时,可从它走到它的一个祖先.

按DFS序给每个节点编号.①对树枝边(x,y),时间戳$x<y$;②对前向边(x,y),时间戳$x<y$;③对后向边(x,y),时间戳$x>y$;④对横插边,时间戳$x>y$.

[**Tarjan算法求SCC**] 所求的是强连通分量中深度最小的点.

对每个节点定义两个时间戳:①$dfn[u]$,表示遍历到节点u的时间戳;②$low[u]$,表示以节点u为根节点的子树中的所有节点所能追溯到的时间戳最小的节点.

若节点u是其所在的强连通分量中深度最小的点,则$dfn[u]=low[u]$,即u无法再追溯到更前的点,则u是其所在的强连通分量的深度最小的节点.

用一个栈$stk[]$存当前正在遍历的强连通分量中非最高点的点,用一个bool数组$in_stk[]$记录每个点是否在栈中.每次遍历当前节点u的所有出边,若边的另一个端点j还未被搜过(即$dfn[j]$未更新),则继续搜该节点,并用$low[j]$更新$low[u]$;否则,若j在栈中,则用$dfn[j]$更新$low[u]$.遍历完u的所有出边后,若$dfn[u]=low[u]$,则u是其所在的强连通分量的最高点,更新强连通分量的个数,并将该强连通分量的其他节点出栈,并记录它们所在的强连通分量的编号$id[]$.这样每个节点只会被遍历一次,时间复杂度$O(n+m)$.

[**缩点**] 遍历所有节点i,遍历其所有邻点j,若i与j不在同一SCC中,则加一条边$id[i]\to id[j]$,其中$id[i]$表示节点i所在的SCC的编号.这样建立的图是DAG,且按$id[]$递减的顺序是拓扑序.

14.19.1 受欢迎的牛

题意

有n $(1\le n\le 1e4)$头牛,编号$1\sim n$.给定m $(1\le m\le 5e4)$对整数(A,B)(给出的数据中可能有重复),表示牛A喜欢牛B.喜欢具有传递性,若A喜欢B,B喜欢C,则A喜欢C.求有多少头牛被除自己外的所有牛喜欢.

思路

朴素做法:对每个节点做一遍DFS或BFS,则每个节点的时间复杂度$O(n+m)$,总时间复杂度$O(n^2+nm)$,会T.

若图为DAG,则只需考察出度为0的点,其他节点都可以走到它,即除自己外其他牛都喜欢它,且该节点不能到达其他节点,即它不喜欢其他牛.若图非DAG,用缩点将其转化为DAG,找到强连通分量后,该强连通分量对答案的贡献即其所含的节点数,时间复杂度$O(n+m)$.

若存在多于一个节点的出度为0,则不存在一头牛被除自己外的所有牛喜欢.

无需将缩点后的图建出来,只需枚举每条边的两个端点,若两端点不在同一SCC中,则更新出度.

代码

```cpp
const int MAXN = 1e4 + 5;
namespace TarjanSCC {
  vector<int> edges[MAXN];
  int dfn[MAXN], low[MAXN], tim;  // 节点的DFS序、所能追溯到的最小时间戳、时间戳
  stack<int> stk;
  bool state[MAXN];  // 记录节点是否在栈中
  int id[MAXN], siz[MAXN], cnt;  // 节点所在的SCC的编号、SCC的大小、SCC的个数
  int out[MAXN];  // 节点的出度

  void tarjan(int u) {
    dfn[u] = low[u] = ++tim;
    stk.push(u);
    state[u] = true;

    for (auto v : edges[u]) {
      if (!dfn[v]) {
        tarjan(v);
        low[u] = min(low[u], low[v]);
      }
      else if (state[v]) low[u] = min(low[u], dfn[v]);
    }

    if (dfn[u] == low[u]) {  // 该节点是所在SCC的深度最小的点
      cnt++;
      int cur;
      do {  // 记录该SCC中的所有节点的信息
        cur = stk.top(); stk.pop();
        state[cur] = false;
        id[cur] = cnt;
        siz[cnt]++;
      } while (cur != u);
    }
  }
}
using namespace TarjanSCC;

void solve() {
  int n, m; cin >> n >> m;
  while (m--) {
    int u, v; cin >> u >> v;
    edges[u].push_back(v);
  }

  for (int i = 1; i <= n; i++)
    if (!dfn[i]) tarjan(i);

  for (int u = 1; u <= n; u++) {
    for (auto v : edges[u]) {
      int a = id[u], b = id[v];
      if (a != b) out[a]++;  // 节点u与v不在同一SCC中,则该边连接两SCC
    }
  }

  int zeros = 0;  // 出度为0的节点的个数
  int ans = 0;
```

```
  for (int i = 1; i <= cnt; i++) {
    if (!out[i]) {
      ans += siz[i];
      if (++zeros > 1) {  // 不止1个出度为0的节点
        ans = 0;
        break;
      }
    }
  }
  cout << ans << endl;
}

int main() {
  solve();
}
```

14.19.2 学校网络

题意

一些学校连在一个计算机网络上,学校间存在软件支援协议,每个学校有其支援的学校名单(学校A支援学校B不代表学校B支援学校A).某校获得新软件时,无论是直接获得还是通过网络获得,都立即将该软件通过网络传送到其支援的学校,因此若想要一个新软件被所有学校使用,只需提供给一些学校即可.(1)求至少需将新软件直接提供给几个学校才能使所有学校都能使用该软件?(2)至少需添加几条支援关系才能使得将一个新软件提供给任一学校,其他所有学校能通过该网络获得该软件?

输入第一行包含一个整数n $(2 \leq n \leq 100)$,表示学校数.下面n行每行包含若干个整数,其中第$(i+1)$行表示学校i支援的学校名单,每行最后有一个0表示名单结束,只有一个0表示该学校无需支援其他学校.

思路

数据范围较小,可用Floyd算法求传递闭包来求解.下面讨论用SCC的做法.

(2)即求至少需要加几条边才能使得图中任意两节点都连通,亦即将整个图变为SCC至少要加几条边.

(1)考虑图为DAG的情况,若不然通过缩点转化为DAG.

设有p个起点(入度为0的点),q个终点(出度为0的点),则至少需给p个起点发软件才能保证每个节点都能收到软件.

[**证**] ①p个入度为0的点只能直接发软件,故$ans \geq p$.

②$ans = p$时,因图是DAG,则每个节点都能逆拓扑序走回某个起点,则每个起点能沿拓扑序将软件发给各个节点.

(2)特判只有一个SCC的情况,答案为0;否则至少需加$max(p, q)$条边.证明:不妨设$p \leq q$.

①若$p = 1$,即图中只有1个起点,为使得它能到达q个终点,则起点向各个终点分别连一条边即可.

②若$p > 1$,则$q > 1$,此时$\exists$互不相同的起点p_1, p_2和终点q_1, q_2 $s.t.$ p_1可以到达q_1, p_2可以到达q_2.

[**证**] 若不然,则所有起点都只能到达q_1,与q_2可从某个起点到达矛盾.

此时连边$q_1 \to p_2$,并将q_1从终点集Q中删除得到新终点集Q',将p_2从起点集P中删除得到新起点集P',则$|P'| = |P| - 1, |Q'| = |Q| - 1$,即进行一次这样的操作可以使得起点集的元素个数$-1$.

进行$p-1$次操作后只有一个起点,化为①的情况,此时有$[q-(p-1)]$个终点.

由①知:至少加$(q-p+1)$条边,则总共需加$q-p+1+(p-1) = q$条边.

代码

```cpp
struct TarjanSCC {
  int n;  // 节点数
  vector<vector<pair<int, int>>> edges;
  vector<int> dfn, low;  // 节点的DFS序、所能追溯到的最小时间戳
  int tim;  // 时间戳
  stack<int> stk;
  vector<bool> state;  // 记录节点是否在栈中
  int sccCnt;  // SCC的个数
  vector<int> id;  // 节点所在的SCC的编号
  vector<int> siz;  // SCC的大小
  vector<int> in, out;  // 缩点后SCC的入度、出度

  TarjanSCC(int _n, const vector<vector<pair<int, int>>>& _edges) :n(_n), edges(_edges) {
    tim = sccCnt = 0;
    dfn.resize(n + 1), low.resize(n + 1), state.resize(n + 1);
    state.resize(n + 1), id.resize(n + 1), siz.resize(n + 1);
    in.resize(n + 1), out.resize(n + 1);

    for (int u = 1; u <= n; u++)
      if (!dfn[u]) tarjan(u);

    for (int u = 1; u <= n; u++) {
      for (auto [v, _] : edges[u]) {
        int a = id[u], b = id[v];
        if (a != b) out[a]++, in[b]++;
      }
    }
  }

  void tarjan(int u) {
    dfn[u] = low[u] = ++tim;
    stk.push(u);
    state[u] = true;

    for (auto [v, _] : edges[u]) {
      if (!dfn[v]) {
        tarjan(v);
        low[u] = min(low[u], low[v]);
      }
      else if (state[v]) low[u] = min(low[u], dfn[v]);
    }

    if (dfn[u] == low[u]) {  // 该节点是其所在的SCC的顶点
      sccCnt++;
      int cur;
      do {
        cur = stk.top(); stk.pop();
        state[cur] = false;
        id[cur] = sccCnt;
        siz[sccCnt]++;
      } while (cur != u);
    }
  }
};

```

```cpp
void solve() {
  int n; cin >> n;
  vector<vector<pair<int, int>>> edges(n + 1);
  for (int i = 1; i <= n; i++) {
    int x;
    while (cin >> x, x) edges[i].push_back({ x,1 });
  }

  TarjanSCC solver(n, edges);

  int ans1 = 0, ans2 = 0;
  for (int u = 1; u <= solver.sccCnt; u++) {
    if (!solver.in[u]) ans1++;
    if (!solver.out[u]) ans2++;
  }
  cout << ans1 << endl;
  cout << (solver.sccCnt == 1 ? 0 : max(ans1, ans2)) << endl;
}

int main() {
  solve();
}
```

14.19.3 最大半连通子图

题意 (2 s)

称一个有向图$G=(V,E)$是半连通的,如果对$\forall u,v\in V$,满足存在一条从节点u到节点v的有向路径或存在一条从节点v到节点u的有向路径.

对图$G=(V,E)$,若E'是E中所有与V'有关的边,则称$G'=(V',E')$是G的一个导出子图.

若G'是G的导出子图,且G'半连通,则称G'是G的半连通子图.称G的所有半连通子图中包含节点数最多的为G的最大半连通子图.

给定一个有向图G,求G的最大半连通子图包含的节点数K和不同的最大半连通子图的个数C,后者输出对x取模.两最大半连通子图不同当且仅当点集不同.

第一行输入三个整数n,m,x $(1\le n\le 1e5,1\le m\le 1e6,1\le x\le 1e8)$,分别表示有向图$G$的节点数、边数、模数.接下来$m$行每行输入两个整数$u,v$ $(1\le u,v\le n)$,表示存在一条从节点u到节点v的有向边.数据保证无重边.

思路

注意到一个SCC中的所有节点都能互相到达,故所有的SCC都选.

缩点为DAG后,可以选择一条不分叉的链,使得链上的节点间能到达,问题转化为求DAG的最长链包含的节点数和最长链的方案数.

$f[u]$表示到节点u的最长路,即包含的节点数的最大值.根据节点u的入边分类,状态转移方程$f[u]=\max\limits_{edge<v,u>} f[v]+siz[u]$,其中$siz[u]$表示节点$u$所在的SCC的大小.

$g[u]$表示使得$f[u]$取得最大值的方案数.

①若$f[v]+siz[u]>f[u]$,则更新$f[u]=f[v]+siz[u],g[u]=g[v]$.

②若$f[v]+siz[u]=f[u]$,则更新$g[u]+=g[v]$.

若题目中不保证无重边,则需给边判重,否则会多统计答案.可用哈希表判重,边$<u,v>$的哈希值定义为$1e6\cdot u+v$.

因需建出缩点后的DAG,故总边数最多为$2m$.

代码

```cpp
const int MAXN = 1e5 + 5;
int MOD;
namespace TarjanSCC {
  int n;
  vector<int> edges[MAXN];  // 原图邻接表
  vector<int> edgesDAG[MAXN];  // 缩点后的图的邻接表
  int dfn[MAXN], low[MAXN], tim;  // 节点的DFS序、所能追溯到的最小时间戳、时间戳
  stack<int> stk;
  bool state[MAXN];  // 记录节点是否在栈中
  int id[MAXN], siz[MAXN], cnt;  // 节点所在的SCC的编号、SCC的大小、SCC的个数
  int out[MAXN], in[MAXN];  // 节点的出度、入度

  void tarjan(int u) {
    dfn[u] = low[u] = ++tim;
    stk.push(u);
    state[u] = true;

    for (auto v : edges[u]) {
      if (!dfn[v]) {
        tarjan(v);
        low[u] = min(low[u], low[v]);
      }
      else if (state[v]) low[u] = min(low[u], dfn[v]);
    }

    if (dfn[u] == low[u]) {  // 该节点是所在SCC的深度最小的点
      cnt++;
      int cur;
      do {  // 记录该SCC中的所有节点的信息
        cur = stk.top(); stk.pop();
        state[cur] = false;
        id[cur] = cnt;
        siz[cnt]++;
      } while (cur != u);
    }
  }

  void build() {  // 建出缩点后的DAG
    for (int i = 1; i <= n; i++)
      if (!dfn[i]) tarjan(i);

    // 判重后加边
    unordered_map<ll, int> mp;
    for (int u = 1; u <= n; u++) {
      for (auto v : edges[u]) {
        int a = id[u], b = id[v];
        ll hash = (ll)a * 1e5 + b;
```

```
        if (a != b && !mp.count(hash)) {
          edgesDAG[a].push_back(b);
          mp[hash] = 1;
        }
      }
    }
  }
}
using namespace TarjanSCC;

int f[MAXN];  // f[u]表示到节点u的最长路,即包含的节点数的最大值
int g[MAXN];  // g[u]表示使得f[u]取得最大值的方案数

void solve() {
  int m; cin >> n >> m >> MOD;
  while (m--) {
    int u, v; cin >> u >> v;
    edges[u].push_back(v);
  }

  build();

  for (int u = cnt; u; u--) {  // 按拓扑序DP
    if (!f[u]) f[u] = siz[u], g[u] = 1;  // 起点

    for (auto v : edgesDAG[u]) {
      if (f[v] < f[u] + siz[v]) f[v] = f[u] + siz[v], g[v] = g[u];
      else if (f[v] == f[u] + siz[v]) g[v] = (g[v] + g[u]) % MOD;
    }
  }

  int maxf = 0, sum = 0;  // f[]的最大值和取得最大值的方案数
  for (int i = 1; i <= cnt; i++) {
    if (f[i] > maxf) maxf = f[i], sum = g[i];
    else if (f[i] == maxf) sum = (sum + g[i]) % MOD;
  }
  cout << maxf << endl << sum << endl;
}

int main() {
  solve();
}
```

14.19.4 银河

题意

用一个正整数表示恒星的亮度,数值越大越量,亮度最暗为1.现有编号$1 \sim n$的n颗恒星和m对亮度关系,求这n颗恒星的亮度值和的最小值.

第一行输入两个整数n, m $(1 \leq n, m \leq 1e5)$.之后m行每行输入三个整数T, A, B $(1 \leq T \leq 5, 1 \leq A, B \leq n)$,表示恒星$A$与恒星$B$的亮度关系,格式如下:

①$T = 1$时,恒星A的亮度与恒星B相等.

②$T = 2$时,恒星A的亮度$<$恒星B的亮度.

③$T=3$时,恒星A的亮度$\geq$恒星B的亮度.

④$T=4$时,恒星A的亮度$>$恒星B的亮度.

⑤$T=5$时,恒星A的亮度$\leq$恒星B的亮度.

若有解,输出这n颗恒星的亮度值之和的最小值;否则输出-1.

思路

本题也可用差分约束解决.下面讨论SCC的做法.

由差分约束知:求最小值用最长路,存在正环时无解.

隐含条件$x_i \geq 1\ \ (1 \leq i \leq n)$.

本题中所有边权都非负,为判断图中是否存在正环,因每个环在同一SCC中,若一个SCC中存在一条边的权值>0,则存在正环.故有解时每个SCC中的边权都为0.

对一个SCC,若有关系$A_1 \geq A_2 \geq A_3 \geq A_1$,则说明所有恒星的亮度都相等.

求缩点后的DAG上的最长路即可,每个节点对答案的贡献是它到起点的距离乘它所在的SCC的大小,最坏情况所有节点连成一条链且都在同一SCC中,此时答案为$O(n^2)$,会爆int.

时间复杂度稳定的$O(n+m)$,不会像差分约束中SPFA容易被卡.但Tarjan算法的空间更大.

代码

```cpp
const int MAXN = 1e5 + 5;
namespace TarjanSCC {
  int n;
  vector<pair<int, int>> edges[MAXN];  // 原图邻接表
  vector<pair<int, int>> edgesDAG[MAXN];  // 缩点后的图的邻接表
  int dfn[MAXN], low[MAXN], tim;  // 节点的DFS序、所能追溯到的最小时间戳、时间戳
  stack<int> stk;
  bool state[MAXN];  // 记录节点是否在栈中
  int id[MAXN], siz[MAXN], cnt;  // 节点所在的SCC的编号、SCC的大小、SCC的个数
  int out[MAXN], in[MAXN];  // 节点的出度、入度

  void tarjan(int u) {
    dfn[u] = low[u] = ++tim;
    stk.push(u);
    state[u] = true;

    for (auto [v, w] : edges[u]) {
      if (!dfn[v]) {
        tarjan(v);
        low[u] = min(low[u], low[v]);
      }
      else if (state[v]) low[u] = min(low[u], dfn[v]);
    }

    if (dfn[u] == low[u]) {  // 该节点是所在SCC的深度最小的点
      cnt++;
      int cur;
      do {  // 记录该SCC中的所有节点的信息
        cur = stk.top(); stk.pop();
        state[cur] = false;
```

```
        id[cur] = cnt;
        siz[cnt]++;
      } while (cur != u);
    }
  }

  bool build() {  // 建出缩点后的DAG,返回是否有解
    tarjan(0);  // 从超级源点开始搜即可

    for (int u = 0; u <= n; u++) {  // 注意从0开始枚举
      for (auto [v, w] : edges[u]) {
        int a = id[u], b = id[v];
        if (a != b) edgesDAG[a].push_back({ b,w });
        else if (w > 0) return false;  // SCC中有边权>0的边时无解
      }
    }
    return true;
  }
}
using namespace TarjanSCC;

int dis[MAXN];  // 节点到起点的最长路

void solve() {
  int m; cin >> n >> m;
  for (int i = 1; i <= n; i++) edges[0].push_back({ i,1 });  // 超级源点向各个节点连边
  while (m--) {
    int op, u, v; cin >> op >> u >> v;
    if (op == 1) edges[u].push_back({ v,0 }), edges[v].push_back({ u,0 });
    else if (op == 2) edges[u].push_back({ v,1 });
    else if (op == 3) edges[v].push_back({ u,0 });
    else if (op == 4) edges[v].push_back({ u,1 });
    else edges[u].push_back({ v,0 });
  }

  if (!build()) {
    cout << -1 << endl;
    return;
  }

  for (int u = cnt; u; u--) {  // 按拓扑序DP
    for (auto [v, w] : edgesDAG[u])
      dis[v] = max(dis[v], dis[u] + w);
  }

  ll ans = 0;
  for (int i = 1; i <= cnt; i++) ans += (ll)dis[i] * siz[i];
  cout << ans << endl;
}

int main() {
  solve();
}
```

14.19.5 Edge Reverse

原题指路:https://codeforces.com/contest/1777/problem/E

题意 (4 s)

给定一棵包含编号$1 \sim n$的n个节点和编号$1 \sim m$的m条边的有向树,其中$i\ (1 \le i \le m)$号边的权值为w_i.现需反转一些边的方向,使得至少存在一个可以到达其他所有节点的节点,反转的代价为反转的边的最大边权.若有解,输出最小代价;否则输出-1.

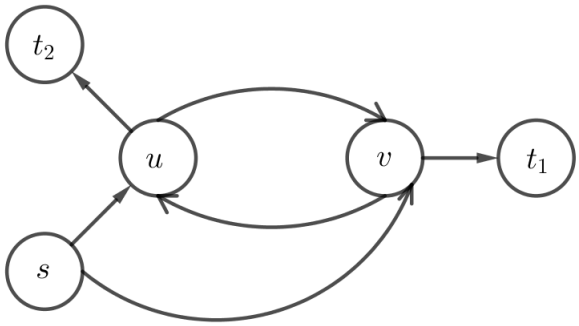
有$t\ (1 \le t \le 1e5)$组测试数据.每组测试数据第一行输入两个整数$n, m\ (2 \le n \le 2e5, 1 \le m \le 2e5)$.接下来$m$行每行输入三个整数$u, v, w\ (1 \le u, v \le n, u \ne v, 1 \le w \le 1e9)$.数据保证所有测试数据的$n$之和不超过$2e5$.数据给定的边构成一棵树.

思路

[**引理**] 反转一条边可视为将该边变为双向边.

[**证**] 考察反转边$u \to v$对图的中节点的可达性的影响.

若不然,设从节点s到节点t_1必须经过正向边$u \to v$,到节点t_2必须经过反向边$v \to u$,此时可表示为:



由上图:存在路径$s \to u \to t_2$,与s到t_2必须经过反向边$v \to u$矛盾.

因反转的代价为反转的边的最大边权,由**引理**:可反转的边越多,图越接近一个无向图,进而越存在一个可以到达其他所有节点的节点,即反转的代价具有二段性,考虑二分.

对二分的代价mid,将图中边权$\le mid$的边变为双向边后根据SCC缩点,显然存在一个可以到达其他所有节点的节点的充要条件是:缩点后在且只存在一个入度为0的SCC.

代码

```cpp
struct TarjanSCC {
  int n;  // 节点数
  vector<vector<pair<int, int>>> edges;
  vector<int> dfn, low;  // 节点的DFS序、所能追溯到的最小时间戳
  int tim;  // 时间戳
  stack<int> stk;
  vector<bool> state;  // 记录节点是否在栈中
  int sccCnt;  // SCC的个数
  vector<int> id;  // 节点所在的SCC的编号
  vector<int> siz;  // SCC的大小
  vector<int> in, out;  // 缩点后SCC的入度、出度

  TarjanSCC(int _n, const vector<vector<pair<int, int>>>& _edges) :n(_n), edges(_edges) {
    tim = sccCnt = 0;
    dfn.resize(n + 1), low.resize(n + 1), state.resize(n + 1);
```

```
    state.resize(n + 1), id.resize(n + 1), siz.resize(n + 1);
    in.resize(n + 1), out.resize(n + 1);

    for (int u = 1; u <= n; u++)
      if (!dfn[u]) tarjan(u);

    for (int u = 1; u <= n; u++) {
      for (auto [v, _] : edges[u]) {
        int a = id[u], b = id[v];
        if (a != b) out[a]++, in[b]++;
      }
    }
  }

  void tarjan(int u) {
    dfn[u] = low[u] = ++tim;
    stk.push(u);
    state[u] = true;

    for (auto [v, _] : edges[u]) {
      if (!dfn[v]) {
        tarjan(v);
        low[u] = min(low[u], low[v]);
      }
      else if (state[v]) low[u] = min(low[u], dfn[v]);
    }

    if (dfn[u] == low[u]) {  // 该节点是其所在的SCC的顶点
      sccCnt++;
      int cur;
      do {
        cur = stk.top(); stk.pop();
        state[cur] = false;
        id[cur] = sccCnt;
        siz[sccCnt]++;
      } while (cur != u);
    }
  }
};

void solve() {
  int n, m; cin >> n >> m;
  vector<vector<pair<int, int>>> edges(n + 1);
  int maxw = 0;
  while (m--) {
    int u, v, w; cin >> u >> v >> w;
    edges[u].push_back({ v,w });
    maxw = max(maxw, w);
  }

  auto check = [&](int mid)->bool {
    auto Edges = edges;
    for (int u = 1; u <= n; u++) {
      for (auto [v, w] : edges[u])
        if (w <= mid) Edges[v].push_back({ u,w });
    }

    TarjanSCC solver(n, Edges);
```

```
74
75      int cnt = 0;  // 入度为0的SCC数
76      for (int i = 1; i <= solver.sccCnt; i++)
77        cnt += !solver.in[i];
78      return cnt == 1;
79    };
80
81    int l = 0, r = maxw + 1;  // 注意二分的上界+1以判定无解的情况
82    while (l < r) {
83      int mid = l + r >> 1;
84      if (check(mid)) r = mid;
85      else l = mid + 1;
86    }
87    cout << (l == maxw + 1 ? -1 : l) << endl;
88  }
89
90  int main() {
91    CaseT
92      solve();
93  }
```

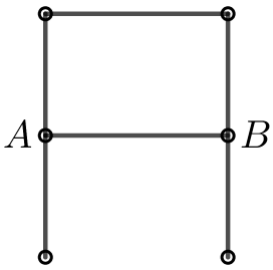
14.20 无向图的双连通分量

在无向图中,若删除某边可使得该图不连通,则称该边为桥.

在无向图中,若删除某节点和所有以它为端点之一的边可使得该图不连通,则称该节点为割点.

割点与桥无必然关系.

[**性质1**] 两割点间的边未必是桥.



[**证**] 如上图,节点A和节点B是割点,但边(A,B)不是桥.

[**性质2**] 桥的两端点未必是割点.

[**证**] 考察只有边(A,B)的图,删去任意节点都会使得图剩下一个节点,即一个连通块,故两个节点都不是割点.

无向图的双连通分量(又称重连通分量)分为两种:

①边双连通分量e-DCC,指无向图中极大的不含桥的连通块,与有向图的SCC类似.

[**性质1**] 删除e-DCC中的任一条边不改变其连通性.

[**性质2**] 一个连通块是e-DCC充要条件是:连通块中的任意两节点间存在两条无公共边的路径.

[**证**] (充) 若任意两节点间存在两条无公共边的路径,但它不是e-DCC,则至少存在一座桥,将其删去后图不连通.

该桥将该连通块分为两个部分,且一个部分的节点到另一个部分的节点的路径都经过桥,

进而连接两部分的路径都存在公共边,与任意两节点间存在两条无公共边的路径矛盾.

(必) 若一个连通块是e-DCC,且存在节点x和节点y使得它们之间有且只有两条有公共边的路径.

将公共边删去将使得该连通块不连通,即该边是桥,与e-DCC不包含桥矛盾.

②点双连通分量v-DCC,指无向图中极大的不包含割点的连通块.

[**性质**] 每个割点至少属于两个v-DCC,每个v-DCC至少包含一个割点.

e-DCC与v-DCC间无必然关系.

[**e-DCC的Tarjan算法**] 对每个节点定义两个时间戳:①$dfn[u]$,表示遍历到节点u的时间戳;②$low[u]$,表示以节点u为根节点的子树中的所有节点所能追溯到的时间戳最小的节点.与有向图的不同之处在于,无向图不存在横插边.

对边(x,y),若节点y无法追溯到节点x或其祖先节点,即$dfn[x]<low[y]$时,边(x,y)是桥.

求e-DCC的两种方法:①删去所有桥后每个连通块是一个e-DCC;②类似于Tarjan算法求SCC的方法,遍历时用一个栈存下当前的e-DCC中的所有节点.

注意搜索时不能往回搜,否则每个节点沿其反向边都能追溯到其前驱节点,进而图不存在桥.

[**v-DCC的Tarjan算法**] 对每个节点定义两个时间戳:①$dfn[u]$,表示遍历到节点u的时间戳;②$low[u]$,表示以节点u为根节点的子树中的所有节点所能追溯到的时间戳最小的节点.

对边(x,y),若节点y至多追溯到节点x,即$low[y]\geq dfn[x]$时:

①若节点x非根节点,则删去节点x后其子树与其父亲节点间不连通,故节点x是割点.

②若节点x是根节点,且其子树中至少存在两个节点$y_1,y_2\ s.t.\ low[y_1]\geq dfn[x],low[y_2]\geq dfn[x]$,则节点$x$是割点.

求v-DCC的方法:遍历时用一个栈存下当前v-DCC中的所有节点.设以x为根节点的子树可以分成的连通块数为cnt.对节点x和节点y,若$dfn[x]\leq low[y]$,且x非根节点或$cnt\geq 2$,则节点x是割点,此时将栈中元素弹出直至弹出节点y,弹出的节点连同节点x构成一个v-DCC.注意特判孤立点自身构成的v-DCC.

v-DCC的缩点步骤:①每个割点单独作为一个节点;②每个v-DCC向其所包含的每个割点连边.缩点后的图中边数不超过原图的边数,节点数等于连通分量数加割点数≤ 2倍原图的节点数.

14.20.1 冗余路径

题意

给定一个包含编号$1\sim n$的n $(1\leq n\leq 5000)$个节点和m $(n-1\leq m\leq 1e4)$条边的连通图.现需添加若干条边,使得每一对节点间都存在至少两条相互分离的路径,两条路径相互分离当其仅当它们无公共边,但两条相互分离的路径上可能有相同的节点.初始时每对节点间至少存在一条路径.对已存在两条不同路径的节点,也可在其间添加边作为另一条不同的路径.问至少需要添加多少条边.

思路

即给定一个无向连通图,问至少加几条边可将其变为一个e-DCC.

注意到最优策略中只会在不同的e-DCC间加边,求出所有e-DCC后将每个e-DCC缩点,缩点后的图是一棵树,其中所有边都是桥.

所有度数为1的节点都至少需加一条边,否则删去与其相连的边将使得图不连通.

设度数为1的节点有cnt个,则$ans \geq \left\lceil \frac{cnt}{2} \right\rceil$.可以证明$ans$可以取得最小值$\left\lceil \frac{cnt}{2} \right\rceil$.

代码

```cpp
struct TarjanEDCC {
    int n, m;
    vector<int> head, edge, nxt;
    int idx;
    vector<int> dfn;  // 节点的DFS序
    vector<int> low;  // 节点能追溯到的最小时间戳
    int tim;  // 时间戳
    stack<int> stk;
    vector<int> id;  // 节点所在的e-DCC的编号
    int edccCnt;  // e-DCC的个数
    vector<bool> isBridge;
    vector<int> d;  // 缩点后节点的度数

    TarjanEDCC(int _n, int _m) :n(_n), m(_m * 2), idx(0), tim(0), edccCnt(0) {
        head = vector<int>(n + 5, -1);
        edge.resize(m + 5), nxt.resize(m + 5);
        dfn.resize(n + 5), low.resize(n + 5);
        id.resize(n + 5), isBridge.resize(n + 5);
        d.resize(n + 5);
    }

    void add(int u, int v) {
        edge[idx] = v, nxt[idx] = head[u], head[u] = idx++;
    }

    void addEdge(int u, int v) {
        add(u, v), add(v, u);
    }

    void tarjan(int u, int pre) {  // 当前节点、前驱边
        dfn[u] = low[u] = ++tim;
        stk.push(u);

        for (int i = head[u]; ~i; i = nxt[i]) {
            int v = edge[i];
            if (!dfn[v]) {
                tarjan(v, i);
                low[u] = min(low[u], low[v]);
                if (dfn[u] < low[v])
                    isBridge[i] = isBridge[i ^ 1] = true;
            }
            else if (i != (pre ^ 1))  // 非反向边
                low[u] = min(low[u], dfn[v]);
        }
```

```
45
46          if (dfn[u] == low[u]) {  // u是所在的e-DCC的深度最小的节点
47              edccCnt++;
48              int v;
49              do {
50                  v = stk.top(); stk.pop();
51                  id[v] = edccCnt;
52              } while (v != u);
53          }
54      }
55
56      void work() {
57          for (int i = 0; i < idx; i++)
58              if (isBridge[i]) d[id[edge[i]]]++;
59
60          int cnt = 0;
61          for (int i = 1; i <= edccCnt; i++)
62              cnt += d[i] == 1;
63          cout << (cnt + 1) / 2 << endl;
64      }
65  };
66
67  void solve() {
68      int n, m; cin >> n >> m;
69
70      TarjanEDCC solver(5000, 1e4);
71      for (int i = 0; i < m; i++) {
72          int u, v; cin >> u >> v;
73          solver.addEdge(u, v);
74      }
75
76      solver.tarjan(1, -1);
77      solver.work();
78  }
79
80  int main() {
81      solve();
82  }
```

14.20.2 电力

题意

给定一个包含编号$0 \sim (n-1)$的n个节点和m条边的无向图,求删除该图的一个节点后最多得到多少连通块.

有多组测试数据.每组测试数据第一行输入两个整数n, m $(1 \leq n \leq 1e4, 0 \leq m \leq 1.5e4)$.接下来$m$行每行输入两个整数$u, v$ $(0 \leq u, v \leq n-1)$,表示节点u与节点v间存在无向边.数据保证无重边.最后一行输入0 0表示输入结束.

思路

先求连通块个数cnt,枚举删除的节点所在的连通块,再枚举连通块中删除的节点,设删去该节点后得到ans个连通块,问题转化为求$(ans + cnt - 1)$的最大值.

若$dfn[x] \leq low[y]$,则删除节点x将使得节点y独立为一个连通块,此时:①若x非根节点,则原树被分为三个连通块;②若x为根节点,则原树被分为两个连通块.

代码

```cpp
const int MAXN = 1e4 + 5, MAXM  = 3e4 + 5;
namespace TarjanVDCC {
  int n;
  int head[MAXN], edge[MAXM], nxt[MAXM], idx;
  int dfn[MAXN], low[MAXN], tim;  // 节点的DFS序、所能追溯到的最小时间戳、时间戳
  int id[MAXN], edccCnt;  // 节点所在的e-DCC的编号、e-DCC的个数
  int root;
  int ans;  // 删除割点后得到的连通块的个数的最大值

  void add(int a, int b) {
    edge[idx] = b, nxt[idx] = head[a], head[a] = idx++;
  }

  void tarjan(int u) {
    dfn[u] = low[u] = ++tim;

    int cnt = 0;  // 以u为根节点的子树可以分成的连通块数
    for (int i = head[u]; ~i; i = nxt[i]) {
      int v = edge[i];
      if (!dfn[v]) {
        tarjan(v);
        low[u] = min(low[u], low[v]);
        if (low[v] >= dfn[u]) cnt++;
      }
      else low[u] = min(low[u], dfn[v]);
    }

    cnt += u != root;  // u非根节点时连通块个数多1个
    ans = max(ans, cnt);
  }
}
using namespace TarjanVDCC;

void init() {
  memset(head, -1, sizeof(head));
  memset(dfn, 0, sizeof(dfn));
  idx = tim = ans = 0;
}

void solve() {
  int m;
  while (cin >> n >> m, n || m) {
    init();

    while (m--) {
      int u, v; cin >> u >> v;
      add(u, v), add(v, u);
    }

    int cnt = 0;  // 连通块个数
    for (root = 0; root < n; root++)
      if (!dfn[root]) cnt++, tarjan(root);
    cout << ans + cnt - 1 << endl;
  }
}
```

```
56
57 int main() {
58   solve();
59 }
```

14.20.3 割点(割顶)

原题指路:https://www.luogu.com.cn/problem/P3388

题意

给定一张包含编号$1 \sim n$的n $(1 \leq n \leq 2e4)$个节点和m $(1 \leq m \leq 1e5)$条边的无向图,求图的割点.

代码

```cpp
const int MAXN = 2e4 + 5, MAXM  = 2e5 + 5;
namespace TarjanVDCC {
  int n;
  vector<int> edges[MAXN];
  int dfn[MAXN], low[MAXN], tim;  // 节点的DFS序、所能追溯到的最小时间戳、时间戳
  int id[MAXN], edccCnt;  // 节点所在的e-DCC的编号、e-DCC的个数
  int ans;  // 割点数
  bool isCut[MAXN];  // 记录节点是否是割点

  void tarjan(int u, int fa) {  // 当前节点、前驱节点
    dfn[u] = low[u] = ++tim;

    int son = 0;  // 节点u的子树数
    for (auto v : edges[u]) {
      if (!dfn[v]) {
        son++;
        tarjan(v, u);
        low[u] = min(low[u], low[v]);
        if (low[v] >= dfn[u] && u != fa && !isCut[u])
          isCut[u] = true, ans++;
      }
      else if (v != fa) low[u] = min(low[u], dfn[v]);
    }

    if (u == fa && son >= 2 && !isCut[u])
      isCut[u] = true, ans++;
  }
}
using namespace TarjanVDCC;

void solve() {
  int m; cin >> n >> m;
  while (m--) {
    int u, v; cin >> u >> v;
    edges[u].push_back(v), edges[v].push_back(u);
  }

  for (int u = 1; u <= n; u++)
    if (!dfn[u]) tarjan(u, u);

```

```
  cout << ans << endl;
  for (int u = 1; u <= n; u++)
    if (isCut[u]) cout << u << ' ';
  cout << endl;
}

int main() {
  solve();
}
```

14.20.4 矿场搭建

题意

煤矿工地可看作一张由隧道连接挖煤点的无向图.现需在某些挖煤点设置出口,使得每个挖煤点坍塌后,其他挖煤点的工人都可以到达出口.求所需的出口数的最小值和取得最小值的方案数。

由多组测试数据.每组测试数据第一行输入一个整数m $(1 \leq m \leq 500)$,表示隧道数.接下来m行每行输入两个整数u, v,表示挖煤点u与挖煤点v间有隧道相连.每组测试数据的挖煤点编号$1 \sim Max$ $(1 \leq Max \leq 1000)$,其中Max表示由隧道连接的挖煤点中的最大编号,可能存在未被隧道连接的挖煤点.最后一行输入一个0,表示输入结束.

对每组测试数据,输出一行,其中第i $(i \geq 1)$行以"Case i: "开始,然后输出两个整数,分别表示所需的出口数的最小值和取得最小值的方案数.数据保证答案$< 2^{64}$.

思路

[**性质1**] 出口数≥ 2.

[**性质2**] 每个连通块的方案独立,可分别考虑每个连通块后将方案数相乘.

[**性质3**] 若连通块中不存在割点,则只需设置两个出口即可满足要求.设连通块大小为cnt,则方案数为C_{cnt}^{2}.

考察原图按v-DCC缩点后的图.

因缩点后的图是一棵树,若以割点为根节点,则根节点处的挖煤点坍塌后会将树分为至少两个连通块.原树中度数为1的节点是树的叶子节点,在每个叶子节点处设置一个出口即可满足要求.

(1)若图中某个v-DCC的度数为0,则它是一个孤立点,需设置一个出口.

(2)若图中某个v-DCC的度数为1,则其中只包含一个割点.

①若割点处的挖煤点坍塌,则它连接的v-DCC中都需在任一非割点处设置一个出口.设一个v-DCC的大小为cnt,则方案数为$(cnt-1)$.

下面证明设置1个出口可满足要求.因缩点后的图是一棵树,若以割点为根节点,则根节点处的挖煤点坍塌后会将树分为至少两个连通块.原树中度数为1的节点是树的叶子节点,在某个叶子节点处设置一个出口即可满足要求.

其余情况中割点处的挖煤点坍塌同理.

②若非割点处的挖煤点坍塌,因该v-DCC的度数为1,且它所包含的一个割点还与另一个v-DCC相连,故一个v-DCC中的非割点处坍塌时,其他节点可到达另一个v-DCC的叶子节点的出口.

③若图中某个v-DCC的度数> 1,则它所包含的一个割点至少还与另两个v-DCC相连,故一个v-DCC和中的非割点处坍塌时,其他节点可到达其他v-DCC的叶子节点的出口,故度数> 1的v-DCC中无需设置出口.

代码

```cpp
const int MAXN = 2e4 + 5, MAXM  = 2e5 + 5;
namespace TarjanVDCC {
  int n;
  vector<int> edges[MAXN];
  int dfn[MAXN], low[MAXN], tim;  // 节点的DFS序、所能追溯到的最小时间戳、时间戳
  int root;
  stack<int> stk;
  int edccCnt;  // e-DCC的个数
  vector<int> edcc[MAXN];  // 每个e-DCC中的节点
  bool isCut[MAXN];  // 记录节点是否是割点

  void tarjan(int u) {  // 当前节点
    dfn[u] = low[u] = ++tim;
    stk.push(u);

    if (u == root && !edges[u].size()) {  // 节点u是孤立点
      edcc[++edccCnt].push_back(u);
      return;
    }

    int cnt = 0;  // 以u为根节点的子树可以分成的连通块数
    for (auto v : edges[u]) {
      if (!dfn[v]) {
        tarjan(v);
        low[u] = min(low[u], low[v]);

        if (dfn[u] <= low[v]) {
          cnt++;
          if (u != root || cnt > 1) isCut[u] = true;  // 节点u是割点

          edccCnt++;
          int x;
          do {
            x = stk.top(); stk.pop();
            edcc[edccCnt].push_back(x);
          } while (x != v);  // 注意是v不是u
          edcc[edccCnt].push_back(u);  // 节点u也属于该e-DCC
        }
      }
      else low[u] = min(low[u], dfn[v]);
    }
  }
}
using namespace TarjanVDCC;

int C(int n) {  // 组合数C(n,2)
  return n * (n - 1) / 2;
}

void init() {
  for (int i = 1; i <= n; i++) edges[i].clear();
  for (int i = 1; i <= edccCnt; i++) edcc[i].clear();
  memset(dfn, 0, sizeof(dfn));
  memset(isCut, false, sizeof(isCut));
  n = tim = edccCnt = 0;
```

```
}

void solve() {
  int Case = 1, m;
  while (cin >> m, m) {
    init();

    while (m--) {
      int u, v; cin >> u >> v;
      n = max({ n,u,v });
      edges[u].push_back(v), edges[v].push_back(u);
    }

    for (root = 1; root <= n; root++)
      if (!dfn[root]) tarjan(root);

    int exit = 0;  // 出口数
    unsigned long long ans = 1;  // 方案数
    for (int i = 1; i <= edccCnt; i++) {
      int cut = 0;  // 割点数
      for (auto u : edcc[i]) cut += isCut[u];

      if (!cut) {  // 该连通块中无割点
        if (edcc[i].size() > 1) exit += 2, ans *= C(edcc[i].size());
        else exit++;  // 注意特判孤立点构成一个v-DCC的情况,防止方案数乘0
      }
      else if (cut == 1) exit++, ans *= edcc[i].size() - 1;
    }
    cout << "Case " << Case++ << ": " << exit << ' ' << ans << endl;
  }
}

int main() {
  solve();
}
```

14.21 Euler回路与Euler路径

Euler路径:图中的一条经过所有边恰一次的路径,即一笔画问题.

Euler回路:图中的一个回路,使得从一个节点出发,经过该回路可回到该节点,且经过图中的每条边恰一次.

(1)在无向图中,若所有边都连通,

①若Euler路径的起点与终点不同,则它们的度数为奇数,其余路径上的节点的度数都为偶数.

②若Euler路径的起点与终点不同,则所有节点的度数都为偶数.

综上,无向图存在Euler路径的充要条件是:图中奇数度的节点有0个或2个.

因Euler回路可视为起点与终点相同的Euler路径,故无向图存在Euler回路的充要条件是:不存在奇数度的节点.

[**必.证**]

从某个奇数度的节点开始DFS,除起点和终点外,因其余节点的度数都为偶数,故当前遍历到的非起点和终点的节点存在出边.

遍历到终点时回溯,将路径上经过的节点为走过的环走一遍,则最终路径是从起点到终点的简单路径加上一些环.

不存在奇数度的节点时,搜索的路径是起点的回路加上一些环.

显然环可与相连的路径一笔画,环可与环按"8"字型一笔画.

DFS过程中当节点出栈时记录路径即可,最终得到的是Euler路径的倒序.

(2)在有向图中,若所有边都连通,

①若Euler路径的起点与终点不同,则起点的出度比入度多1,终点的入度比出度多1,其余路径上的节点的入度等于出度.

②若Euler路径的起点与终点相同,则所有节点的入度等于出度.

综上,有向图存在Euler路径的充要条件是:要么所有节点的入度等于出度,要么除两个节点外其他节点的入度等于出度,且剩余的两节点中,一个节点的出度比入度多1,一个节点的入度比出度多1.

因Euler回路可视为起点与终点相同的Euler路径,故有向图存在Euler回路的充要条件是:所有节点的入度等于出度.

Euler路径和Euler回路的DFS用边判重,若一个节点有多个自环,则用邻接表存图时时间复杂度为$O(nm)$.考虑优化,经过一条边后将其删去,这样每条边只会经过一次,时间复杂度$O(n+m)$.对有向图,直接删去该边即可;对无向图,删除该边外,还需将其反向边也删去.

14.21.1 铲雪车

题意

城市的所有道路都是双向车道,道路的两个方向都需要铲雪.城市只有一辆铲雪车,铲雪车能将其开过的车道的雪铲干净,它需从停放的地方出发.问最少需多少时间能铲完城市道路上的所有雪.

第一行输入两个整数x,y $(-1e6 \le x,y \le 1e6)$,表示铲雪车停放位置的坐标,单位为米.下面最多4000行,每行输入一条双向道路的起止点坐标,均为范围$[-1e6,1e6]$内的整数.数据保证铲雪车从起点出发可到达任意道路.铲雪车可在任意交叉路口、任意街道的末尾任意转向,包括转U型弯.铲雪车铲雪时前进速度为$20\ \mathrm{km/h}$,不铲雪时前进速度为$50\ \mathrm{km/h}$.

输出铲掉所有街道的雪并范围出发点所需的最短时间,精确到分钟,四舍五入到整数.输出格式为"hours:minutes",其中$minutes$不足两位数时需补前导零.

思路

铲雪车从起点出发可到达任意道路说明铲雪车初始时在某条道路上.

因所有道路都为双向道理,则所有节点的入度和出度都为偶数,进而存在Euler回路.

显然沿着Euler回路铲雪是最优的,且与铲雪车的初始位置无关,$ans=\dfrac{2\cdot sum}{20\ \mathrm{km/h}}$,其中$sum$为所有边长度之和.

代码

```
void solve() {
  double x1, y1, x2, y2; cin >> x1 >> y1;

  double sum = 0;
  while (cin >> x1 >> y1 >> x2 >> y2) {
    double dx = x1 - x2, dy = y1 - y2;
    sum += 2 * hypot(dx, dy);
  }

  int minutes = round(sum / 1000 / 20 * 60), hours = minutes / 60;
```

```
  minutes %= 60;
  printf("%d:%02d\n", hours, minutes);
}

int main() {
  solve();
}
```

14.21.2 Euler回路

题意

给定一张包含编号$1 \sim n$的n个节点和编号$1 \sim m$的m条边的图,求它的任一条Euler回路.

第一行输入一个整数t $(1 \leq t \leq 2)$,$t = 1$表示所给的图为无向图,$t = 2$表示所给的图为有向图.第二行输入两个整数n, m $(1 \leq n \leq 1e5, 0 \leq m \leq 2e5)$.接下来$m$行每行输入两个整数,其中第$i$ $(1 \leq i \leq m)$行输入的两个整数u_i, v_i $(1 \leq u_i, v_i \leq n)$当$t = 1$时表示存在有一条从节点u_i到节点v_i的无向边,当$t = 2$时表示存在一条从节点u_i到节点v_i的有向边.图中可能存在重边和自环.

若不存在Euler回路,输出"NO";否则第一行输出"YES",第二行输出任一条Euler回路.

①若$t = 1$,输出m个整数$p_1, \cdots, p_m$.令$e = |p_i|$表示经过的第i条边的编号,p_i为正数时表示从节点v_e走到节点u_e;否则表示从节点u_e走到节点v_e.

②若$t = 2$,输出m个整数$p_1, \cdots, p_m$,其中p_i表示经过的第i条边的编号.

思路

(1)对无向图,先检查:

①所有节点的度数都为偶数.

②所有边都连通.

(2)对有向图,先检查:

①所有节点的入度等于出度.

②所有边都连通.

注意本题不保证图连通,故要找一个包含边的起点开始DFS.

本题数据范围较大,为防止TLE,遍历一条边后将其删去.

代码

```
const int MAXN = 1e5 + 5, MAXM = 4e5 + 5;
int type;
int n, m;
int head[MAXN], edge[MAXM], nxt[MAXM], idx;
bool used[MAXM];  // 记录每条边是否被遍历过
int ans[MAXM], cnt;
int in[MAXN], out[MAXN];  // 节点的入度、出度

void add(int a, int b) {
```

```
  edge[idx] = b, nxt[idx] = head[a], head[a] = idx++;
}

void dfs(int u) {
  for (auto& i = head[u]; ~i;) {  // 让i始终存当前可用的第一条边
    if (used[i]) {
      i = nxt[i];  // 删边
      continue;
    }

    used[i] = true;
    if (type == 1) used[i ^ 1] = true;  // 无向图删除反向边

    int t;
    if (type == 1) {
      t = i / 2 + 1;
      if (i & 1) t = -t;  // 反向走
    }
    else t = i + 1;

    int j = edge[i];
    i = nxt[i];  // 删边
    dfs(j);

    ans[++cnt] = t;  // 节点出栈时记录路径
  }
}

void solve() {
  memset(head, -1, sizeof(head));

  cin >> type >> n >> m;
  for (int i = 0; i < m; i++) {
    int a, b; cin >> a >> b;
    add(a, b);
    if (type == 1) add(b, a);
    in[b]++, out[a]++;
  }

  // 特判无解的情况
  if (type == 1) {
    for (int i = 1; i <= n; i++) {
      if (in[i] + out[i] & 1) {
        cout << "NO" << endl;
        return;
      }
    }
  }
  else {
    for (int i = 1; i <= n; i++) {
      if (in[i] != out[i]) {
        cout << "NO" << endl;
        return;
      }
    }
  }

  for (int i = 1; i <= n; i++) {  // 找一个有边的节点开始搜
```

```
    if (~head[i]) {
      dfs(i);
      break;
    }
  }

  if (cnt < m) {  // 图中的边不连通
    cout << "NO" << endl;
    return;
  }

  cout << "YES" << endl;
  for (int i = cnt; i; i--) cout << ans[i] << " \n"[i == 1];  // 倒序输出
}

int main() {
  solve();
}
```

14.21.3 骑马修栅栏

题意

给定一张包含不超过500个节点和m条边的无向图,节点编号$1 \sim 500$.求该图的一条Euler路径,输出路径上的节点编号.将输出的路径视为一个500进制数,存在多组解时,输出500进制表示法中最小的.数据保证有解.

第一行输入一个整数m $(1 \leq m \leq 1024)$.接下来m行每行输入两个整数i, j $(1 \leq i, j \leq 500)$,表示该边连接节点i和节点j.

思路

DFS记录答案时得到的是Euler路径的逆序,但字典序最大的逆序未必是字典序最小的.

DFS时先走连接节点编号最小的出边即可.为避免对重边和自环排序,可用邻接矩阵存图.

代码

```
const int MAXN = 505, MAXM = 1035;
int n = 500;
int graph[MAXN][MAXN];
int d[MAXN];
int ans[MAXM], cnt;

void dfs(int u) {
  for (int i = 1; i <= n; i++) {
    if (graph[u][i]) {
      graph[u][i]--, graph[i][u]--;  // 删边
      dfs(i);
    }
  }

  ans[++cnt] = u;  // 节点出栈时记录路径
}

void solve() {
```

```
  int m; cin >> m;
  while (m--) {
    int a, b; cin >> a >> b;
    graph[a][b]++, graph[b][a]++;
    d[a]++, d[b]++;
  }

  int start = 1;
  while (!d[start]) start++;  // 找到第一个非零度数的节点,防止所有节点的度数都为偶数时无输出
  for (int i = 1; i <= n; i++) {  // 找到第一个度数为奇数的点
    if (d[i] & 1) {
      start = i;
      break;
    }
  }
  dfs(start);

  for (int i = cnt; i; i--) cout << ans[i] << endl;
}

int main() {
  solve();
}
```

14.21.4 单词游戏

题意

有n个盘子,每个盘子上写着一个只包含小写英文字母的单词.现需给所有盘子排序,使得对相邻两盘子,前一个盘子上的单词的尾字母等于后一个盘子上的单词的首字母.

有t组测试数据.每组测试数据第一行输入一个整数n $(1 \leq n \leq 1e5)$.接下来n行每行输入一个长度不超过1000的、只包含小写 英文字母的单词,一个单词可能出现多次.

若有解,输出"Ordering is possible.";否则输出"The door cannot be opened.".

思路

每个单词的首字母代表的节点向尾字母代表的节点连边,问题转化为图中是否存在Euler路径.

代码

```
const int MAXN = 35;
int n;
int in[MAXN], out[MAXN];  // 出度、入度
bool state[MAXN];  // 每个节点是否出现过

namespace DSU {
  int fa[MAXN];

  int find(int x) {
    return x == fa[x] ? x : fa[x] = find(fa[x]);
  }
}
using namespace DSU;
```

```
14
15  void init() {
16    memset(state, false, sizeof(state));
17    memset(in, 0, sizeof(in));
18    memset(out, 0, sizeof(out));
19
20    iota(fa, fa + MAXN, 0);
21  }
22
23  void solve() {
24    init();
25
26    cin >> n;
27    while (n--) {
28      string s; cin >> s;
29      int len = s.length();
30      int a = s[0] - 'a', b = s[len - 1] - 'a';
31      state[a] = state[b] = true;
32      out[a]++, in[b]++;
33      fa[find(a)] = find(b);
34    }
35
36    int start = 0, end = 0;  // 可以是起点和终点的节点数
37    bool ok = true;
38    for (int i = 0; i < 26; i++) {
39      if (in[i] != out[i]) {
40        if (in[i] == out[i] + 1) end++;
41        else if (in[i] + 1 == out[i]) start++;
42        else {
43          ok = false;
44          break;
45        }
46      }
47    }
48
49    if (ok && !(!start && !end || start == 1 && end == 1)) ok = false;
50
51    int u = -1;  // 代表节点
52    for (int i = 0; i < 26; i++) {
53      if (state[i]) {
54        if (u == -1) u = find(i);
55        else if (u != find(i)) {  // 边不连通
56          ok = false;
57          break;
58        }
59      }
60    }
61
62    if (ok) cout << "Ordering is possible." << endl;
63    else cout << "The door cannot be opened." << endl;
64  }
65
66  int main() {
67    CaseT
68    solve();
69  }
```

14.22 Hamilton路径、Hamilton回路与竞赛图

[**Hamilton路径**] 经过且只经过图的所有节点一次的路径称为该图的一条**Hamilton路径**.

[**Hamilton回路**] 经过且只经过图的所有节点一次的回路称为该图的一条**Hamilton回路**.

[**Hamilton图,半Hamilton图**] 有Hamilton回路的图称为**Hamilton图**,有Hamilton路径而无Hamilton回路的图称为**半Hamilton图**.

[**竞赛图**] 给无向完全图中的每条边分配一个方向得到的有向完全图称为**竞赛图**.若将包含n个节点的竞赛图(n阶竞赛图)中的有向边$< u, v >$视为玩家u打败玩家v,则该竞赛图可视为n个玩家两两对战且无平局的比赛结果.

[**定理14.22.1**] 设$G =< V, E >$是Hamilton图,则对V的任一非空真子集V_1,有$p(G - V_1) \leq |V_1|$,其中$p(x)$表示节点x的连通分支数.

[**注**] 将G换为半Hamilton图,结论仍成立.

[**定理14.22.2**] (1)完全图K_{2k+1} $(k \geq 1)$中包含k条无重边的Hamilton回路,且这些回路覆盖K_{2k+1}中的所有边.

(2)完全图K_{2k} $(k \geq 2)$中包含$(k-1)$条无重边的Hamilton回路,从K_{2k}中删除这些回路包含的边所得的图中包含k条无公共点的边.

[**Hamilton路径、Hamilton回路的存在性的判定**]

[**定理14.22.3**] 若对包含n $(n \geq 2)$个节点的无向简单图G中任意两不相邻的节点u和v,都有$d(u) + d(v) \geq n - 1$,则G中存在Hamilton路径.

[**推论1**] 若对包含n $(n \geq 3)$个节点的无向简单图G中任意两不相邻的节点u和v,都有$d(u) + d(v) \geq n$,则G中存在Hamilton回路.

[**推论2**] 若对包含n $(n \geq 3)$个节点的无向简单图G中的任一顶点u,都有$d(u) \geq \dfrac{n}{2}$,则G中存在Hamilton回路.

[**Hamilton路径、Hamilton回路与竞赛图的关系**]

[**定理14.22.4**] n $(n \geq 2)$阶竞赛图有Hamilton路径.

[**证**] 考虑归纳.设当前图可分为两部分,每个部分包含若干节点且分别包含一条Hamilton路径.

下面构造整张图的Hamilton路径.设两部分的的Hamilton路径的起点分别为A和B.

用①类询问确定A与B间的有向边的方向,不妨设从A指向B.

将A作为整张图的Hamilton路径的第一个节点,删除节点A,重复上述过程即可.

[**推论**] 包含n $(n \geq 2)$阶竞赛图的子图的图有Hamilton路径.

[**注**] 构造Hamilton路径的过程类似于归并排序.

[**定理14.22.5**] 强连通的竞赛图有Hamilton回路.

[**推论**] 包含n $(n \geq 2)$阶竞赛图的子图的图有Hamilton回路.

[**定理14.22.6**] 将竞赛图按SCC缩点后所得的图是一条链.

14.22.1 Baby Ehab's Hyper Apartment

原题指路:https://codeforces.com/problemset/problem/1514/E

题意

这是一道交互题.

有一张包含编号$0\sim(n-1)$的n个节点的有向图.对每对节点$u,v\ \ (0\le u,v\le n-1,u\ne v)$,要么存在一条从节点$u$到节点$v$的有向边,要么存在一条节点$v$到节点$u$的有向边,但不会同时存在上述两条有向边.

现有如下两种询问:

①$1\ u\ v\ \ (0\le u,v\le n-1,u\ne v)$,表示询问节点$u$与节点$v$间的有向边是从$u$指向$v$还是从$v$指向$u$,该种询问至多问$9n$次.

询问将返回一个整数0或1,其中0表示有向边从v指向u,1表示有向边从u指向v.

②$2\ u\ k\ v_1\ \cdots\ v_k\ \ (0\le u,v_i\le n-1,u\ne v_i,0\le k<n)$,表示询问节点$u$是否有指向节点$v_1,\cdots,v_k$的有向边,该种询问至多问$2n$次.

询问将返回一个整数0或1,其中1表示至少存在一条从u指向$v_i\ \ (1\le i\le k)$的有向边,否则返回0.

进行完所有询问后,需回答对每对节点$u,v\ \ (0\le u,v\le n-1,u\ne v)$,是否存在从节点$u$到节点$v$的路径.

有$t\ \ (1\le t\le 30)$组测试数据.每组测试数据输入一个整数$n\ \ (4\le n\le 100)$.数据保证所有测试数据的n之和不超过500.

对每组测试数据,先输出一行包含一个整数3.接下来n行每行输出一个长度为n的$0/1$串,其中第$i\ \ (1\le i\le n)$个字符串的第$j\ \ (1\le j\le n)$个字符为'1'当且仅当存在从节点i到节点j的路径.特别地,第$i\ \ (1\le i\le n)$个字符串的第i个字符都为'1'.输出每组测试数据的答案后会返回一个整数,若为1则表示该测试数据答案正确,继续进行下一组测试数据,若为最后一个测试数据则退出;若为-1则表示该测试数据答案错误,需终止程序.

思路I

将竞赛图按SCC缩点后所得的图是一条链,求得每个SCC即可求出所有的答案.

考虑如何找到一条经过所有节点的路径.设当前路径为$u_1,\cdots,u_k$,现要加入节点v,则需找到一个下标$p\in[2,k]\ s.t.$存在边$<u_{p-1},v>$和边$<v,u_p>$,将v插入u_p之前即可.注意到若存在边$<u_{l-1},v>$和边$<v,u_r>$,则可在区间$[l,r]$中求得下标p,即答案具有二段性.考虑二分,用①类询问进行check,需要$n\log n\le 9n$个①类询问.

设上述过程得到的链为$S_1\to\cdots\to S_k$,其中S_i为一个节点集.从后往前缩点,对S_i中的节点u,若存在一条从节点u到$S_1,\cdots,S_{i-1}$中的节点的边,则S_i与S_{i-1}在同一SCC中,将它们合并即可.最坏情况n个节点连成一条链,合并时每个SCC合并到前一个SCC中,节点和SCC各n个,最坏需要$2n$个②类询问.

代码I

```
#undef endl
const int MAXN = 105;
set<int> scc[MAXN];  // 存每个SCC中的节点

int query1(int u, int v) {  // 询问节点u到节点v的有向边的方向,0表示v->u,1表示u->v
  cout << 1 << ' ' << u << ' ' << v << endl;

```

```
  int res; cin >> res;
  if (res == -1) exit(0);
  return res;
}

int query2(int u, vector<int>& nodes) {
  cout << 2 << ' ' << u << ' ' << nodes.size() << ' ';
  for (auto v : nodes) cout << v << ' ';
  cout << endl;

  int res; cin >> res;
  if (res == -1) exit(0);
  return res;
}

void merge(int x, int y) {  // 将scc[y]合并入到scc[x]中
  for (auto u : scc[y]) scc[x].insert(u);
  scc[y].clear();
}

void solve() {
  int n; cin >> n;

  vector<int> v(n + 1);  // 包含所有节点的有序路径
  int cnt = 0;  // 当前路径包含的节点数
  for (int u = 0; u < n; u++) {  // 当前节点
    // 二分出最靠右的l s.t. 存在边<v[l-1],u>和边<u,v[l]>
    int l = 1, r = cnt + 1;
    while (l < r) {
      int mid = l + r >> 1;
      if (query1(u, v[mid])) r = mid;
      else l = mid + 1;
    }

    // 将当前节点插入路径中
    for (int i = cnt; i >= l; i--) v[i + 1] = v[i];
    v[l] = u, cnt++;
  }

  for (int i = 1; i <= cnt; i++) scc[i].insert(v[i]);  // 初始时每个节点分别在一个SCC中

  vector<bool> state(n);  // 记录节点是否已合并入某个SCC中
  for (int cur = cnt; cur > 1; cur--) {  // cur表示当前需合并的SCC的编号
    for (auto u : scc[cur]) {
      if (!state[u]) {
        vector<int> nodes;
        for (int i = 1; i < cur; i++)
          for (auto v : scc[i]) nodes.push_back(v);
        if (query2(u, nodes)) {
          merge(cur - 1, cur);  // 将当前的SCC合并到前面的一个SCC中
          break;
        }
        state[u] = true;
      }
    }
  }

  vector<string> ans(n, string(n, '0'));
```

```
  for (int i = 1; i <= cnt; i++) {
    for (int j = i; j <= cnt; j++) {
      for (auto u : scc[i])
        for (auto v : scc[j]) ans[u][v] = '1';
    }
  }

  cout << 3 << endl;
  for (int i = 0; i < n; i++) cout << ans[i] << endl;

  for (int i = 1; i <= cnt; i++) scc[i].clear();  // 清空

  int res; cin >> res;  // 注意读走每组测试数据的反馈
  if (res == -1) exit(0);
}

int main() {
  CaseT
    solve();
}
```

思路II

注意到本题的目的等价于找到一些边,使得图中的所有路径都能只通过这些边遍历到.

竞赛图存在Hamilton路径,而构造Hamilton路径的过程类似于归并排序,故可以①类询问为比较规则,对图中的所有节点做归并排序,该过程需$n\log n\leq 9n$个①类询问.

实现时无需手写归并排序,可直接用STL中的stable_sort()函数,将①类询问作为比较规则即可,即开一个数组$path[]$记录图中的一条Hamilton路径,初始时$path_i=i$,对该数组stable_sort()即可.

注意到Hamilton路径上的节点的前向边对答案无贡献,故只需考察后向边.从后往前遍历Hamilton路径,考察途中遇到的后向边,该边对答案有贡献,当且仅当它能追溯到未遍历过的边,这是因为通过这些边可得到更多之前未遍历到的路径.

考虑如何得到图中的所有路径.双指针从后往前遍历Hamilton路径,指针p指向当前遍历到的所有边最远能追溯到的节点,指针q指向当前节点.在当前节点处,用②类询问确定是否存在从当前节点指向Hamilton路径的前p个节点的边,若存在,令$p--$,重复上述过程直至不存在从当前节点指向Hamilton路径的前p个节点的边.该过程至多需要$2n$次②类询问,因为每次询问后要么$p--$,要么$q--$,两指针移动距离之和$\leq 2n$.

实现时倒序枚举$path[]$,用②类询问维护指针p即可.

代码II

```
#undef endl
vector<int> path;  // 图中的一条Hamilton路径

bool query1(const int& u, const int& v) {  // 询问节点u到节点v的有向边的方向,0表示v->u,1表示u->v
  cout << 1 << ' ' << u << ' ' << v << endl;

  int res; cin >> res;
  if (res == -1) exit(0);
  return res;
}

```

```cpp
bool query2(int u, int len) {  // 询问是否存在从节点u指向节点path[0],...,path[len]的有向边
  if (len < 0) return false;

  cout << 2 << ' ' << u << ' ' << len + 1 << ' ';
  for (int i = 0; i <= len; i++) cout << path[i] << ' ';
  cout << endl;

  int res; cin >> res;
  if (res == -1) exit(0);
  return res;
}

void solve() {
  int n; cin >> n;

  path.clear();
  for (int i = 0; i < n; i++) path.push_back(i);
  stable_sort(all(path), query1);  // 以query1()为比较规则,对path[]做归并排序

  vector<string> ans(n, string(n, '1'));
  int ancestor = n - 2;  // 当前遍历到的所有边最远能追溯到的节点
  for (int i = n - 1; i >= 0; i--) {  // 倒序遍历Hamilton路径
    if (ancestor == i) {  // 节点i是当前最远能追溯到的节点
      ancestor--;

      for (int j = 0; j <= i; j++) {
        for (int k = i + 1; k < n; k++)
          ans[path[k]][path[j]] = '0';  // 之后的节点不能追溯到之前的节点
      }
    }

    while (query2(path[i], ancestor)) ancestor--;
  }

  cout << 3 << endl;
  for (int i = 0; i < n; i++) cout << ans[i] << endl;

  int res; cin >> res;  // 注意读走每组测试数据的反馈
  if (res == -1) exit(0);
}

int main() {
  CaseT
    solve();
}
```

14.22.2 Anya's Simultaneous Exhibition

原题指路:https://codeforces.com/contest/1779/problem/E

题意

这是一道交互题.

有编号$1 \sim n$的n个选手,满足如下两性质:①对任意一对选手,其中一个选手总能打败另一个选手,即无平局;②打败关系没有传递性,即存在a号选手打败b号选手,b号选手打败c号选手,c号选手打败a号选手的情况.现未知每一对选手中谁能打败谁.

现需举办$(n-1)$场一对一的比赛,每场比赛选取两个选手进行比赛,胜者留下,负者淘汰.所有比赛结束后只会剩下一个选手.称一个选手是好的,如果他可能最后留下.

现需找到所有好的人,为此举办至多$2n$场一对多的比赛,每场比赛选取一个选手P,让他与其他若干(至少一个)选手进行比赛,一场一对多的比赛结束后,将得到选手P的胜场数.在一对多的比赛中无淘汰机制.在询问过程中,不同场的一对多的比赛中玩家之间的胜负关系可能变化,但变化不与之前的比赛矛盾.

第一行输入一个整数n $(3 \leq n \leq 250)$.接下来可至多输入$2n$个询问,每个询问表示一场一对多的比赛.每个询问的格式为"? i $s_1 \cdots s_n$",其中i表示该场一对多的比赛中选出的选手P的编号,$s_1 \cdots s_n$是一个长度为n的$0/1$串,表示与i号选手比赛的其他选手,其中$s_j = 1$ $(1 \leq j \leq n)$表示i号选手在该场比赛中将与j号选手比赛,$s_j = 0$ $(1 \leq j \leq n)$表示i号选手在该场比赛中不与j号选手比赛,注意必须有$s_i = 0$.每个询问结束后将返回一个整数,表示i号选手的胜场数.

当得到确切的答案时,输出一行"! $c_1 \cdots c_n$",其中$c_1 \cdots c_n$是一个长度为n的$0/1$串,其中$c_i = 1$表示i号玩家是好的.

思路

对于一个给定的竞赛图,i号选手好的,当且仅当存在从i号选手到其他所有选手的路径.

[**引理I**] 若节点i的出度最大,则i号选手是好的.

[**证**] 考虑一个更强的命题:若节点i的出度最大,则从节点i出发,经1或2条边即可到达其他所有节点.

设S_1为从节点i出发,经1条边可到达的其他所有节点构成的集合,S_2为除节点i和S_1中的节点外其他所有节点构成的集合.

对$u \in S_2$,若从节点i出发无法到达节点u,因该图是竞赛图,则存在节点u到节点i的有向边,同理存在节点u到节点i的有向边.

这表明:节点u的出度$\geq |S_1| + 1$,而节点i的出度为$|S_1|$,与i的出度最大矛盾,故证.

[**引理II**] 存在一个整数w $s.t.$ 节点i是好的当且仅当它的出度$\geq w$.

[**证**] 设竞赛图按SCC缩点后得到的链为$S_1, \cdots, S_k$,其中S_1是拓扑序最大的SCC,S_k是拓扑序最小的SCC.

因该图是竞赛图,则对$\forall x \in S_i, y \in S_j$ $(i < j)$,都存在有向边$< x, y >$,进而x的出度$> y$的出度.

这表明:S_1中的所有节点的出度都$> S_i$ $(2 \leq i \leq k)$中的所有节点的出度,故S_1中的所有节点都是好的.

w取S_1中的节点的最小出度即可.

对每个玩家,举办一场他与其他所有玩家的一对多的比赛,则他的胜场数即他对应的节点的出度.所有玩家都进行一场一对多的比赛后,出度最大的节点是好的节点之一.

将所有节点按出度非升序排列,对每个玩家进行一场一对多的比赛,同时维护此时好的节点构成的集合S_1和w,其中w的初始值为前n场比赛中得到的节点的最大出度.对当前玩家,若他至少能打败一个S_1中的玩家,则他也是好的.

事实上只需n场甚至$(n-1)$场一对多的比赛即可确定好的玩家.以n场为例,由上述讨论知:好的玩家对应的节点是出度较大的几个节点.将所有节点按出度非降序排列后,枚举好的玩家数,即第一个SCC包含的节点数ans,则答案应满足出度前ans大的节点的出度之和为$\frac{ans(ans-1)}{2}+ans(n-ans)$,其中第一项是该SCC中所有有向边的贡献,第二项为该SCC中的ans个节点向其他$(n-ans)$个节点的有向边的贡献.

代码I

```cpp
#undef endl

void solve() {
  int n; cin >> n;

  auto query = [&](int i) {
    string s(n, '1');
    s[i - 1] = '0';
    cout << "? " << i << ' ' << s << endl;

    int res; cin >> res;
    return res;
  };

  vector<pair<int, int>> a(n + 1);  // first为节点出度,second为节点编号
  for (int i = 1; i <= n; a[i++].second = i) a[i].first = query(i);
  sort(a.rbegin(), a.rend() - 1);  // 节点按出度非升序排列

  int sum = 0;  // 好的节点的出度之和
  for (int i = 1; i <= n; i++) {
    if ((sum += a[i].first) == i * (i - 1) / 2 + i * (n - i)) {
      string ans(n, '0');
      for (int j = 1; j <= i; j++) ans[a[j].second - 1] = '1';
      cout << "! " << ans << endl;
      return;
    }
  }
}

int main() {
  solve();
}
```

代码II

```cpp
#undef endl

void solve() {
  int n; cin >> n;

  auto query = [&](int i) {
    string s(n, '1');
    s[i - 1] = '0';
    cout << "? " << i << ' ' << s << endl;

    int res; cin >> res;
    return res;
```

```cpp
  };

  vector<int> out(n + 1);  // 节点的出度
  for (int i = 1; i <= n; i++) out[i] = query(i);

  vector<int> ord(n + 1);
  iota(all(ord), 0);
  sort(ord.begin() + 1, ord.end(), [&](int i, int j) {
    return out[i] < out[j];
  });

  int ans = 0;  // 非好的节点数
  int sum = 0;  // 好的节点向非好的节点的有向边贡献的出度之和
  for (int i = 1; i < n; i++) {
    sum += out[ord[i]];
    if (sum == i * (i - 1) / 2) ans = i;
  }

  string c(n, '0');
  for (int i = ans + 1; i <= n; i++) c[ord[i] - 1] = '1';
  cout << "! " << c << endl;
}

int main() {
  solve();
}
```