图论

```cpp
/**
 * 链式前向星  basic
 * 拓扑序 topo
 * 树的直径:  DT
 * 基环树:  Base_Ring_Tree
 * 最短路: shortest_path
 *         https://www.acwing.com/blog/content/462/
 *
 * 最小生成树:  MST
 * 二分图:  bipartite_graph
 *
 *
*/

#include <iostream>
#include <cstring>
#include <vector>
#include <queue>
#include <algorithm>
#include <numeric>
#include <bitset>

const int N = 1e3;
const int INF = 0x3f3f3f3f;

using namespace std;
namespace golitter {
namespace basic {



// 链式前向星
int h[N]; // 链表头, 初始为-1 memset(h, -1, sizeof(h));
int e[N]; // 链表内容
int ne[N]; // 链表中指向下一个元素的指针
int w[N]; // 链表内容的权重
bool vis[N];
int idx; //
// <u   , -- c -- , v>  ( u --- w --> v
void add(int u, int v, int c) {
    e[idx] = v, w[idx] = c, ne[idx] = h[u], h[u] = idx++;
}
void init() {
    memset(h, -1, sizeof(h));
}
void dfs(int u) {
    vis[u] = 1;
    for (int i = h[u]; ~i; i = ne[i]) {
        if (!vis[e[i]]) {
```

```
            dfs(e[i]);
        }
    }
}


// 邻接表
vector<vector<int>> adj;
void add() {
    int u,v;
    adj[u].push_back(v);
}
void dfs(int u) {
    if(vis[u]) return ;
    vis[u] = true;
    for(int i = 0; i < adj[u].size(); ++i) {
        dfs(adj[u][i]);
    }
}

}}

namespace golitter {
namespace topo {

int e[N], ne[N], h[N],idx;
int in[N];
bool vis[N];
void inpfile();
void add(int u, int v) {
    e[idx] = v, ne[idx] = h[u], h[u] = idx++;
}
void solve() {
    memset(h, -1, sizeof(h));
    int n; cin>>n;
    for(int i = 1; i <= n; ++i) {
        int k;
        while(cin>>k, k != 0) {
            add(i,k);
            in[k]++;
        }
    }
    queue<int> q;
    for(int i = 1; i <= n; ++i) {
        if(!in[i]) q.push(i);
    }
    while(q.size()) {
        auto tmp = q.front(); q.pop();
        if(vis[tmp]) continue;
        cout<<tmp<<" ";
        vis[tmp] = 1;
        for(int i = h[tmp]; ~i; i = ne[i]) {
            int y = e[i];
            in[y]--;
            if(!in[y]) q.push(y);
        }
    }
}
```

```
}}

namespace golitter {
namespace DT {

const int INF = 0x3f3f3f3f;
const int N = 2e5 + 21;

/**
 * 树的直径：树中最远的两个节点之间的距离被称为树的直径 -- 算法竞赛进阶指南
 * 求树的直径通常有两种方法，一种是通过两次搜索（bfs和dfs均可），另一种就是通过树形dp来求了。
 * POJ 1985 板子 http://poj.org/problem?id=1985
 * https://blog.csdn.net/AC__dream/article/details/119101320
*/
int e[N], ne[N], w[N], h[N], idx;
bool vis[N];
int ans, d[N],n,m;
void add(int u, int v, int c) {
    e[idx] = v; w[idx] = c; ne[idx] = h[u]; h[u] = idx++;
}
// dp法
void tdp(int x) {
    vis[x] = 1;
    for(int i = h[x]; ~i; i = ne[i]) {
        int y = e[i];
        if(vis[y]) continue;
        tdp(y);
        ans = max(ans, d[x] + d[y] + w[i]);
        d[x] = max(d[x], d[y] + w[i]);
    }
}

// 两次遍历
void dfs(int x, int fa) {
    for(int i = h[x]; ~i; i = ne[i]) {
        int y = e[i];
        if(y == fa) continue; // 树是一种有向无环图，只要搜索过程中不返回父亲节点即可保证不会重复遍历同一个点。
        d[y] = d[x] + w[i]; // 更新每个点到起始点的距离（在树中任意两点的距离是唯一的）
        dfs(y, x); // 继续搜索下一个节点
    }
}
void solve() {
    memset(h, -1, sizeof(h));
    cin>>n>>m; // v edge
    int u,v,c; char ch;
    for(int i = 0; i < m; ++i) {
        cin>>u>>v>>c>>ch;
        add(u,v,c); add(v,u,c);
    }
    // tdp(1);
    // 第一次遍历
    dfs(1,0);
    int e = 0;
    for(int i = 1; i <= n; ++i) {
        if(d[i] > ans) {
            ans = d[i];
```

```
            e = i;
        }
    }
    // reset d
    memset(d, 0, sizeof(d));
    // 第二次遍历
    dfs(e,0);
    for(int i = 1; i <= n; ++i) {
        if(d[i] > ans) {
            ans = d[i];
        }
    }
    cout<<ans;
}

}}

namespace golitter {
namespace Base_Ring_Tree {
/**
 * 基环树 https://www.luogu.com.cn/blog/user52918/qian-tan-ji-huan-shu
 * 基环树就是有n个点n条边的图，由于比树只出现了一个环，那么就称之为基环树了。
*/
    // https://codeforces.com/contest/1872/problem/F
    // https://codeforces.com/contest/1867/problem/D

}}

namespace golitter {
namespace shortest_path {
/**
    1.dijkstra算法,最经典的单源最短路径算法

    2.bellman-ford算法,允许负权边的单源最短路径算法

    3.spfa,其实是bellman-ford+队列优化,其实和bfs的关系更密一点

    4.floyd算法,经典的多源最短路径算法
*/
// dijkstra 朴素
// 最短路
    // 朴素Dijjstra算法
    // 距离 dis[1] = 0, dis[others] = +INF
    // 已经确定最短路的点存入 s。迭代，找到不到s中的距离最近的点，存入 t
    // 用 t更新其他边的距离
typedef pair<int,int> PII;
const int N = 50010;
int n, m;
int g[N][N];
int dist[N];
int h[N], e[N], ne[N],idx,w[N]; // w -- 权重
bool st[N]; // 确保<v0,vi>是否已被确定最短路径 true 表示确定，false表示否定

void Dijkstra__plain() {
    // input n and v;
    cin>>n>>m;
    memset(g,0x3f,sizeof(g));
    while(m--) {
```

```
        int a, b, c;
        cin>>a>>b>>c;
        g[a][b] = min(g[a][b], c); // 可能有多个边，只要找最小的那个边就行

    }
    // start Dijkstra;
    memset(dist,0x3f, sizeof(dist));
    dist[1] = 0;
    for(int i = 0; i < n; ++i) {
        int t = -1;;
        for(int j = 1; j <= n; ++j) {
            if(!st[j] && (t == -1 || dist[t] > dist[j])) { // 若还未存入，或者存入的较大，
                t = j;
            }
        }
        st[t] = true;
        for(int j = 1; j <= n; ++j) {
            dist[j] = min(dist[j], dist[t] + g[t][j]);
        }
    }
    if(dist[n] == 0x3f3f3f3f) {
        ; // 无路径
    } else {
        ; // 有路径
    }
}
// dijkstra 堆优化
void add(int a, int b, int c) {
    e[idx] = b, ne[idx] = h[a], w[idx] = c, h[a] = idx++;
}
void Dijkstra__heap() { // 不需要对重边做处理
    // 手写堆，或者优先队列
    memset(h,-1,sizeof(h));

    // input

    memset(dist, 0x3f, sizeof(dist));
    dist[1] = 0;

    priority_queue<PII,vector<PII>,greater<PII>> heap; // 小顶堆
    heap.push({0,1});
    while(heap.size()) {
        auto t = heap.top();
        heap.pop();
        int ver = t.second, distance = t.first;
        if(st[ver]) {
            continue;
        }
        st[ver] = true;
        for(int i = h[ver]; i != -1; i = ne[i]) {
            int j = e[i];
            if(dist[j] > distance + w[i]) {
                dist[j] = distance + w[i];
                heap.push({dist[j],j});
            }
        }
    }
```

```
    if(dist[n] == 0x3f3f3f3f) {
        // wu
    } else {
        // you
    }
}

// bellman_Ford
// 有负权边
int backup[N];
const int M = 10021;
struct Edge {
    int a, b, w;
}edges[M];
int k;
int BellmanFord() {
    memset(dist, 0x3f, sizeof(dist)); dist[1]  = 0;
    for(int i = 0; i < k; ++i) { // 小于等于前i个边
        memcpy(backup, dist,sizeof(dist));
        for(int j = 0; j < m; ++j) {
            int a = edges[j].a, b = edges[j].b, w = edges[j].w;
            dist[b] = min(dist[b], backup[a] + w);
        }
    }
    if(dist[n] > 0x3f3f3f3f / 2 ) return -1; // 防止无穷到无穷
    return dist[n];
}
void Bellman_Ford() {
    // a -- w -- >b
    // 定义个结构体就可以喽。
    // 松弛操作
    // dis[b] <= dis[a] + w; 三角不等式
    // 有负权回路，不一定有最短路咯
    cin>>n>>m>>k;
    for(int i = 0; i < m; ++i) {
        int a, b, w;
        cin>>a>>b>>w;
        edges[i] = {a,b,w};
    }
    // start Bellman-ford;
    int t = BellmanFord();
    if(t == -1) cout<<"N";
    else cout<<t;

}

// SPFA
    // 随机数据可用，一般会被卡
int SPFA() {
    // 根据三角不等式
    // bfs 将变小的入队，更新以其为起始点的其他边
    memset(dist, 0x3f, sizeof(dist));
    queue<int> q;
    dist[1] = 0;
    q.push(1);
    st[1] = true; // 判断队列中是否重复
    while(q.size()) {
        int t = q.front();
```

```
        q.pop();
        st[t] = false;
        for(int i = h[t]; i != -1; i = ne[i]) {
            int j = e[i];
            if(dist[j] > dist[t] + w[i]) {
                dist[j] = dist[t] + w[i];
                if(!st[j]) {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }
    if(dist[n] == 0x3f3f3f3f) return -1;
    else dist[n];
}
// 判断是否有负环
int cnt[N];
bool Is_SPFA_ECli() { // 都遍历一遍
    // 根据三角不等式
    // bfs 将变小的入队，更新以其为起始点的其他边
   // memset(dis, 0x3f, sizeof(dis));
    queue<int> q;
    dist[1] = 0;
    q.push(1);
    st[1] = true; // 判断队列中是否重复
    while(q.size()) {
        int t = q.front();
        q.pop();
        st[t] = false;
        for(int i = h[t]; i != -1; i = ne[i]) {
            int j = e[i];
            if(dist[j] > dist[t] + w[i]) {
                dist[j] = dist[t] + w[i];
                cnt[j] = cnt[t] + 1;
                if(cnt[j] >= n) return true;
                if(!st[j]) {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }
    // if(dis[n] == 0x3f3f3f3f) return -1;
    // else dis[n];
    return false;
}

// 多源最短路 不能有负权回路
    // 动态规划
// floyd算法主要作用有：1.找最短路   2.求传递闭包   3.找最小环   4.求出恰好经过k条边的最短路
int d[N][N], n, m, Q;
void floyd() {
    cin>>n>>m>>Q;
    for(int i = 1; i <= n; ++i) {
        for(int j = 1; j <= n; ++j) {
            if(i == j) {
```

```
                d[i][j] = 0;
            } else {
                d[i][j] = INF;
            }
        }
    }
    while(m--) {
        int a, b, w;
        cin>>a>>b>>w;
        if(d[a][b] < INF / 2)
        d[a][b] = min(d[a][b], w);
    }

    // start 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]);
            }
        }
    }

    while(Q--) {
        int a, b;
        cin>>a>>b;
        cout<<d[a][b];
    }
}
    // 2. 求传递闭包
/**
 * 在交际网络中，给定若干个元素和若干对二元关系，且关系具有传递性。通过传递性推导出尽量多的元素
之间的关系的问题被成为传递闭包
*/
void floyd_transitive_closure() {
    // 多一层循环 k, 若 i -> k 可以，j -> i 可以， 则 j -> k 可以
    int f[N][N];
    for(int k = 1; k <= n; ++k) {
        for(int i = 1; i <= n; ++i) {
            for(int j = 1; j <= n; ++j) {
                f[i][j] = f[i][j] || (f[i][k] && f[k][j]);
            }
        }
    }
    // https://www.luogu.com.cn/problem/P2881
    bitset<N> bf[N];
    for(int k = 1; k <= n; ++k) {
        for(int j = 1; j <= n; ++j) {
            if(bf[j][k]) bf[j] |= bf[k]; // j 可以到k，则k可以到的点j都可以到，或一下
即可
        }
    }
}

// 差分约束：
/**
 *  ** url: https://zhuanlan.zhihu.com/p/104764488
 * 差分约束系统是下面这种形式的多元一次不等式组
```

```
 * 原理：图论的三角形不等式 dist[b] <= dist[a] + c，表示此时，a -- c --> b 已经是最短路
了。
 *
 * 超级源点建图即可。
*/

}}

namespace golitter {
namespace MST {

// 最小生成树：（无向图常见）在无向图中，连通而且不含有圈（环路）的图称为树
    // 无向图，特殊的有向图
// 正边 负边 都可以
// 记得 memset dist
/**
 * @brief
 *
 *      Prim 稠密图
 *          原理：最近的距离一定在MST上
 *      kruskal 稀疏图
 *          原理：最短的边一定在MST上
 */

typedef long long LL;
typedef pair<int,int> PII;

const int INF = 0x3f3f3f3f;
const int N = 5021;
int n,m, ph[N][N], dist[N]; bool st[N];
void test();
int prim() {
    memset(dist,0x3f, sizeof(dist));
    int res = 0;
    for(int i = 0; i < n; ++i) {
        int t = -1;
        for(int j = 1; j <= n; ++j) {
            if(!st[j] && (t == -1 || dist[t] > dist[j])) {
                t = j;
            }
        }
        if(i && dist[t] == 0x3f3f3f3f) return 0x3f3f3f3f;
        if(i) res += dist[t];
                                            // this is 点到集合的距离
        for(int j = 1; j <= n; ++j) dist[j] = min(dist[j], ph[t][j]);

        st[t] = true;

    }
    return res;
}

// kruskal :
    // 将边的权重按从小到大排序，从小到大依次枚举每条边，并查集，不连通加入集合中
int n,m;
int p[N];
struct Edge {
    int a, b, w;
```

```
    // 方便排序
    bool operator< (const Edge &W) const {
        return w < W.w;
    }
}edges[N];
// 并查集
int find(int x) {
    if(p[x] != x) p[x] = find(p[x]);
    return p[x];
}
void test();
int main()
{
    scanf("%d%d",&n,&m);
    for(int i = 0; i < m; ++i) {
        int a,b,w; scanf("%d%d%d",&a,&b,&w);
        edges[i] = {a,b,w};
    }
    // 排序
    sort(edges, edges+m);
    int res = 0, cnt = 0;
    // 并查集初始化
    for(int i = 1; i <= n; ++i) p[i] = i;
    // 枚举
    for(int i = 0; i < m; ++i) {
        int a = edges[i].a, b = edges[i].b, w = edges[i].w;
        a = find(a), b = find(b);
        // 连通
        if(a != b) {
            res += w; cnt++; p[a] = b;
        }
    }
    // 如果cnt从1开始则为cnt < n， 但是cnt从0开始， 所以是cnt < n-1;;
    if(cnt < n - 1) puts("orz");
    else cout<<res;
    return 0;
}


}}





namespace golitter {
namespace bipartite_graph {

// 二分图： 当且仅当图中不含奇数环；

    // 染色法判断二分图 https://www.acwing.com/problem/content/862/
int n, h[N],e[N],ne[N],idx,color[N];
void add(int a, int b) {
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}
bool dfs(int u, int c) { // 不矛盾 返回 true
    color[u] = c;
    for(int i = h[u]; i != -1; i = ne[i]) {
        int j = e[i];
```

```
        if(color[j] == -1) {
            // 矛盾
            if(!dfs(j,!c)) return false;
            // 相邻颜色相同
        } else if(color[j] == c) return false;
    }
    return true;
}
bool check() {
    memset(color,-1,sizeof color) ;
    bool fg = true;
    for(int i = 1; i <= n; ++i) {
        if(color[i] == -1) {
            // 判断 + 染色
            if(!dfs(i,0)) {
                fg = false;
                break;
            }
        }
    }
    return fg;
}
int n,m;// ver edge
    // 并查集判断二分图
struct ckst{
    vector<int> p;
    ckst(int n): p(n+1) {iota(p.begin(), p.end(),0); } // iota --> <numeric>
    int find(int x) {return p[x]==x?x:p[x]=find(p[x]); }
    void uni(int x, int y) {p[find(x)]=p[find(y)]; }
    bool same(int x, int y) {return p[x]==p[y]; }
};
// M is max edge number
const int M = 34343;
struct Edge{int u,v; }edge[M];
bool check(int n, int m) {
    ckst cs(n*2);
    for(int i = 1; i <= n; ++i) { // 合并两个边上的点
        int u = edge[i].u, v = edge[i].v;
        cs.uni(u,v+n), cs.uni(u+n,v);
    }
    for(int i = 1; i <= m; ++i) {
        // 如果相连通，返回false
        if(cs.same(i,n+i)) return false;
    }
    return true;
}

// 二分图的最大匹配

namespace 二分图匹配 {

namespace 匈牙利 {
        // 时间复杂度 O(n * m)
        // https://www.acwing.com/problem/content/description/863/
int e[N], ne[N], h[N], w[N], idx;
int match[N]; // 存储第二个集合中的每个点当前匹配的第一个集合中的点是哪个
bool vis[N]; // 表示第二个集合中的每个点是否已经被遍历过
void add(int u, int v) {
```

```
    e[idx] = v, ne[idx] = h[u], h[u] = idx++;
}
int n1,n2; // n1表示第一个集合中的点数，n2表示第二个集合中的点数
bool find(int x) {
    for(int i = h[x]; ~i; i = ne[i]) {
        int y = e[i];
        if(vis[y]) continue;
        vis[y] = true;
        if(match[y] == 0 || find(match[y])) {
            match[y] = x;
            return true;
        }
    }
    return false;
}
void inpfile();
void solve() {
    int m;
    memset(h, -1, sizeof(h));
    cin>>n1>>n2>>m;
    for(int i = 0; i < m; ++i) {
        int u,v; cin>>u>>v;
        add(u,v);
        // add(v,u); #  // 保存图，因为只从一遍找另一边，所以该无向图只需要存储一个方向
    }
    int res = 0;
    // 求最大匹配数，依次枚举第一个集合中的每个点能否匹配第二个集合中的点
    for(int i = 1; i <= n1; ++i) {
        memset(vis, 0, sizeof(vis));
        if(find(i)) res++;
    }
    cout<<res;
}

}

}


}}
```