图论

平面图性质

一、定义
G = <V,E>是一个无向图。
1.图G可嵌入平面：如果可以把图G的所有结点和边都画在平面上，同时除断点外连线之间没有交点，就称图G可嵌入平面。画出的无边相交的G’称G的平面嵌入。
2.可平面化：如果图G可以嵌入平面，就称图G可平面化。
3.G中边所包含的区域称作一个面。有界区域称为内部面，无界区域称为外部面，常记作R0，包围面的长度最短的闭链称为该面的边界，面R的边界的长度称为该面的度数，记作deg®。
4.**面的度数计算中，含有割边和桥的度数为2，其余为1。**

❶ 性质 1. $K_1, K_2, K_3, K_4, K_5 - e$ 均为极大可平面图.

❷ 性质 2. 极大平面图必是连通图.

❸ 性质 3. 当图阶数 $n \geq 3$ 时，有割点或者桥的平面图不是极大平面图.

二、定理
定理1：图G可嵌入球面当且仅当图G可嵌入平面。

定理2：G中各面的度数之和等于图G边数的两倍。

证明：设e为图G的两个面的公共边，再计算两个面的度数时候边数各提供1，当e不是公共边时候，也就是e为桥或者割边时候提供度数为2。因此，面的度数之和为边的两倍。

定理3：设R是图G的某个平面嵌入的一个内部面，则存在图G的一个平面嵌入使R为外部面。

定理4：设图G是简单的可平面图，如果G中任意两个不相邻的结点加边后所得到的为非可平面图。则称G是极大可平面图，极大可平面图的任何平面嵌入都称为极大平面图。极大平面图必是连通图

定理5：图G为n阶简单的连通的平面图，G为极大平面图当且仅当G的每一个面的度数为3。
定理说明结点数大于等于3的极大平面图的任何面都是由三角形组成。

结论1：K1,K2,K3,K4,K5-e(K5任意删去一条边)均为极大可平面图，他们的任何平面嵌入都是极大平面图；当阶数等于3时候，有割边或桥的平面图不可能是极大平面图。

结论2：无向完全图K5和无向完全二部图K3,3都是极小非可平面图（去掉一条边就成为可平面图）。

结论3：一个图是可平面图，那么他的子图也是可平面图；
一个图的子图是非可平面图，那么图本身也是非可平面图。

结论4：同一个图的平面嵌入中，外部面和内部面的度数可以不同。

❶ 性质 1. $K_1, K_2, K_3, K_4, K_5 - e$ 均为极大可平面图.

❷ 性质 2. 极大平面图必是连通图.

❸ 性质 3. 当图阶数 $n \geq 3$ 时，有割点或者桥的平面图不是极大平面图.

平面图性质—欧拉公式
一、定理
定理1：欧拉公式：设图G是有n个结点、m条边和r个面的连通平面图，则它们满足：**n - m + r = 2**

推论1：设图G是有n个结点、m条边的连通平面简单图，其中n≥3，则有：**m ≤ 3n - 6**
证明：由图G的面度数之和为边数的二倍，即2m。又因为G是平面简单图每一个面的度数至少为3，则2m≥3r，由欧拉公式有：m ≤ 3n - 6

推论2：设图G是有n个结点、m条边的连通平面简单图，其中n≥3且没有长度为3的圈，则有：**m ≤ 2n - 4**

证明：G没有长度为3的圈也就没有度为3的面，G的每一个面的度数至少为4。所以2m≥4r，由欧拉公式有：m ≤ 2n - 4
对于推论1和推论2我们可以用定理进行判定它不是平面图。

例1：证明K5和K3,3是非平面图。

证明：在K5中，m应该小于等于3n - 6，即m≤9。而完全图K5具有10条边。所以是非平面图。

在K3,3中，没有长度大于3的圈，根据推论2可知，m≤2n-4，也就是m≤8，而K3,3含有9条边，所以是非平面图。

推论3：

设 G 是连通的平面图，且每个面的度数至少为 $l\,(l \geq 3)$，则

$$m \leq \frac{l}{l-2}(n-2).$$

证明：同理，有2m≥r x l，根据欧拉公式化简得：
2m ≥ l(m - n + 2)

推论4：

设 G 是平面图，有 ω 个连通分支，n 个结点，m 条边，r 个面，则公式 $n-m+r=\omega+1$ 成立.

推论5：

设 G 是有 n 个结点、m 条边和 r 个面、ω 个连通分支的平面图，且 G 的各个面的度数至少为 l，$(l \geq 4)$，则

$$m \leq \frac{(n-\omega-1)l}{l-2}.$$

证明过程与推论3类似，用到推论4的结论。

推论6：

设 G 是任意平面简单图，则 $\delta(G) \leq 5$.

证明. 设 G 有 n 个顶点 m 条边. 若 $m \leq 6$，结论显然成立；若 $m > 6$，假设 G 的每个顶点的度数 > 6，则由推论 1，有 $6n \leq \sum d(v) = 2m \leq 2(3n-6) = 6n-12$，与定理矛盾，故 $\delta(G) \leq 5$.

极大平面图的判别定理:$n(n \geq 3)$ 阶连通的简单平面图 G. 则以下四个条件等价:

1. G 是极大平面图;
2. G 中每个面的度数都是3;
3. G 中有 m 条边 r 个面，则 $3r = 2m$;
4. 设 G 带有 n 个顶点，m 条边，r 个面则 $m = 3n - 6$;

lca 最近公共祖先

```cpp
std::vector<int> adj[MAXN];
int depth[MAXN], lg[MAXN], p[MAXN][30];
int lca(int x, int y) {
    if (depth[x] < depth[y])std::swap(x, y);
    while (depth[x] > depth[y])
        x = p[x][lg[depth[x] - depth[y]] - 1];
    if (x == y)return x;
    for (int k = lg[depth[x]] - 1;k >= 0;--k)
        if (p[x][k] != p[y][k])
            x = p[x][k], y = p[y][k];
    return p[x][0];
}

int get_dis(int u, int v) {
    int c = lca(u, v);
    return depth[u] + depth[v] - depth[c] * 2;
}

void dfs(int x, int par) {
    p[x][0] = par;
    depth[x] = depth[par] + 1;
    for (int i = 1;i <= lg[depth[x]];++i)
        p[x][i] = p[p[x][i - 1]][i - 1];

    for (int nxt : adj[x])if (nxt != par)dfs(nxt, x);
}

void init() {
    for (int i = 1;i <= n;++i)
        lg[i] = lg[i >> 1] + 1;
}
```

重链剖分

```
struct HPD_tree
{
    int tree_size;
    bool is_hpd_init = false;
    std::vector<std::vector<std::pair<int, i64>>> adj;
    std::vector<int> Fa, size, hson, top, rank, dfn, depth;
    HPD_tree(int n = 0) {
        tree_size = n;
        adj.resize(tree_size + 1);
    }
    void add_edge(int u, int v, i64 w = 1) {
        adj[u].push_back({ v,w });
        adj[v].push_back({ u,w });
    }
    void HPD_init() {
        is_hpd_init = true;
        Fa.assign(tree_size + 1, 0);
        size.assign(tree_size + 1, 0);
        hson.assign(tree_size + 1, 0);
        top.assign(tree_size + 1, 0);
        rank.assign(tree_size + 1, 0);
        dfn.assign(tree_size + 1, 0);
        depth.assign(tree_size + 1, 0);
        std::function<void(int, int, int)> dfs1 = [&](int u, int p, int d)->void {
            hson[u] = 0;
            size[hson[u]] = 0;
            size[u] = 1;
            depth[u] = d;
            for (auto [v, w] : adj[u])if (v != p) {
                dfs1(v, u, d + 1);
                size[u] += size[v];
                Fa[v] = u;
                if (size[v] > size[hson[u]]) {
                    hson[u] = v;
                }
            }
            };
        dfs1(1, 0, 0);
        int tot = 0;
        std::function<void(int, int, int)> dfs2 = [&](int u, int p, int t)->void {
```

```cpp
            top[u] = t;
            dfn[u] = ++tot;
            rank[tot] = u;
            if (hson[u]) {
                dfs2(hson[u], u, t);
                for (auto [v, w] : adj[u])if (v != p && v != hson[u]) {
                    dfs2(v, u, v);
                }
            }
            };
        dfs2(1, 0, 1);
    }
    int lca(int u, int v) {
        if (!is_hpd_init)HPD_init();
        while (top[u] != top[v]) {
            if (depth[top[u]] > depth[top[v]])
                u = Fa[top[u]];
            else
                v = Fa[top[v]];
        }
        return depth[u] > depth[v] ? v : u;
    }
    i64 dist(int u, int v) {
        int w = lca(u, v);
        return depth[u] - depth[w] + depth[v] - depth[w] + 1;
    }
    a3 get_diam() {
        i64 cur; int pos;
        std::function<void(int, int, i64)> dfs = [&](int u, int p, i64 d) {
            if (d > cur) {
                cur = d;
                pos = u;
            }
            for (auto [v, dis] : adj[u])if (v != p) {
                dfs(v, u, d + dis);
            }
            };
        cur = 0, pos = 1;
        dfs(pos, 0, cur);
        int u = pos;
        cur = 0;
        dfs(pos, 0, cur);
        int v = pos;
```

```
        return { u,v,cur };
    }
};
```

树的直径

```
int dpest, dpest_p;
int dfs2(int x, int p, int depth) {
    if (depth > dpest_p) {
        dpest_p = depth;
        dpest = x;
    }
    for (int nxt : adj[x])if (nxt != p) {
        dfs2(nxt, x, depth + 1);
    }
    return dpest;
}

signed main(){
    dpest = -1, dpest_p = 0;
    u = dfs2(r, 0, 0);
    dpest = -1, dpest_p = 0;
    v = dfs2(u, 0, 0);
}
```

割边

```cpp
struct CutEdge {
    int n, tot = -1;
    vector<pair<int, int>> edge;
    vector<vector<int>> map;
    vector<int> d, id, ans;

    CutEdge(int n) :n(n), d(n, -1), id(n, -1), map(n) {};

private :
    void _cutedge(int now, int _edge) {
        d[now] = id[now] = ++tot;
        for (auto tag: map[now]) {
            auto &here = edge[tag].second;
            if (!~d[here]) {
                _cutedge(here, tag);
                id[now] = min(id[now], id[here]);
                if (id[here] > d[now]) {
                    ans.push_back(tag);
                }
            } else if (tag != (_edge ^ 1)) {
                id[now] = min(id[here], id[now]);
            }
        }
    }

public:
    void addedge(int u, int v) {
        edge.push_back({u, v});
        map[u].push_back(int(edge.size()) - 1);
    }

    void cutedge(int u, int _edge) {
        _cutedge(u, _edge);
    }
};
```

割点

```cpp
struct CutPoint {
    int n, tot = -1, root = -1;
    vector<vector<int>> map;
    vector<int> d, id;
    vector<bool> iscutpoint;

    CutPoint(int n): n(n), map(n), d(n, -1), id(n, -1), iscutpoint(n, 0) {};

private:
    void _cutpoint(int now) {
        d[now] = id[now] = ++tot;
        int child = 0;
        for (auto u: map[now]) {
            if (!~d[u]) {
                _cutpoint(u);
                id[now] = min(id[now], id[u]);
                if (id[u] >= d[now]) {
                    ++child;
                    if (now != root || child >= 2) {
                        iscutpoint[now] = 1;
                    }
                }
            } else id[now] = min(d[u], id[now]);
        }
    }

public:
    void addedge(int u, int v) {
        map[u].push_back(v);
    }
    void cutpoint(int now, int root) {
        this->root = root;
        _cutpoint(now);
        this->root = -1;
    }
};
```

SCC 强连通分量

```cpp
struct SCC {
    int n;
    std::vector<std::vector<int>> adj;
    std::vector<int> stk;
    std::vector<int> dfn, low, bel;
    int cur, cnt;

    SCC() {}
    SCC(int n) {
        init(n);
    }

    void init(int n) {
        this->n = n;
        adj.assign(n, {});
        dfn.assign(n, -1);
        low.resize(n);
        bel.assign(n, -1);
        stk.clear();
        cur = cnt = 0;
    }

    void addEdge(int u, int v) {
        adj[u].push_back(v);
    }

    void dfs(int x) {
        dfn[x] = low[x] = cur++;
        stk.push_back(x);

        for (auto y : adj[x]) {
            if (dfn[y] == -1) {
                dfs(y);
                low[x] = std::min(low[x], low[y]);
            } else if (bel[y] == -1) {
                low[x] = std::min(low[x], dfn[y]);
            }
        }

        if (dfn[x] == low[x]) {
```

```
            int y;
            do {
                y = stk.back();
                bel[y] = cnt;
                stk.pop_back();
            } while (y != x);
            cnt++;
        }
    }

    std::vector<int> work() {
        for (int i = 0; i < n; i++) {
            if (dfn[i] == -1) {
                dfs(i);
            }
        }
        return bel;
    }
};
```

```cpp
std::vector<int> id(n, -1), dfn(n, -1), low(n, -1);
std::vector<int> stk;
int now = 0, cnt = 0;
std::function<void(int)> tarjan = [&](int u) {
    stk.push_back(u);
    dfn[u] = low[u] = now++;
    for (auto v : e[u]) {
        if (dfn[v] == -1) {
            tarjan(v);
            low[u] = std::min(low[u], low[v]);
        }
        else if (id[v] == -1) {
            low[u] = std::min(low[u], dfn[v]);
        }
    }
    if (dfn[u] == low[u]) {
        int v;
        do {
            v = stk.back();
            stk.pop_back();
            id[v] = cnt;
        } while (v != u);
        ++cnt;
    }
    };
for (int i = 0; i < n; ++i) if (dfn[i] == -1) tarjan(i);
```

拓扑排序

```
using namespace std;
const int N = 100010;
int e[N], ne[N], idx; //邻接表存储图
int h[N];//邻接表的每个头链表
int q[N], hh = 0, tt = -1; //队列保存入度为0的点，也就是能够输出的点
int n, m; //保存图的点数和边数
int d[N];//保存各个点的入度

void add(int a, int b) {
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

void topsort() {
    for (int i = 1; i <= n; i++) {//遍历一遍顶点的入度。
        if (!d[i])//如果入度为0，则可以入队列
            q[++tt] = i;
    }
    while (tt >= hh) { //循环处理队列中点的
        int a = q[hh++];
        for (int i = h[a]; i != -1; i = ne[i]) {
            int b = e[i]; //a 有一条边指向b
            d[b]--;//删除边后，b的入度减1
            if (!d[b])//如果b的入度减为 0,则 b 可以输出，入队列
                q[++tt] = b;
        }
    }
    if (tt == n - 1) {//如果队列中的点的个数与图中点的个数相同，则可以进行拓扑排序
        for (int i = 0; i < n; i++)//队列中保存了所有入度为0的点，依次输出
            printf("%d ", q[i]);
    } else//如果队列中的点的个数与图中点的个数不相同，则可以进行拓扑排序
        cout << -1;
}

int main() {
    cin >> n >> m; //保存点的个数和边的个数
    memset(h, -1, sizeof h); //初始化领接矩阵
    while (m--) { //依次读入边
        int a, b;
        cin >> a >> b;
        d[b]++;//顶点b的入度+1
```

```
        add(a, b); //添加到邻接矩阵
    }
    topsort();//进行拓扑排序
    return 0;
}
```

欧拉图

完全图最长欧拉通路

```
std::vector<i64> primes;

void solve() {
    int n;std::cin >> n;
    int m = 1;
    while (n - 1 > (m % 2 == 1 ? m * (m + 1) / 2 : m * m / 2 + 1))m++;

    std::vector<int> ans;ans.reserve(n);
    std::vector<std::vector<bool>> adj(m, std::vector<bool>(m, 1));
    std::vector<int> cur(m);
    if (m % 2 == 0)for (int i = 1;i < m - 1;i += 2)adj[i][i + 1] = adj[i + 1][i] = 0;

    auto euler = [&](auto&& self, int x)->void {
        for (int& i = cur[x];i < m;++i) {
            if (adj[x][i]) {
                adj[x][i] = adj[i][x] = 0;
                self(self, i);
            }
        }
        ans.push_back(primes[x]);
        };
    euler(euler, 0);

    ans.resize(n);
    for (int x : ans)std::cout << x << ' ';
}
```

dfs序

可以在dfs序上进行差分操作

```
std::vector<pii> dfsx(n);
int idx = 1;
std::function<int(int, int)> dfs2 = [&](int u, int p) {
    dfsx[u].first = dfsx[u].second = idx++;
    for (int v : py[u])if (v != p) {
        dfsx[u].second = std::max(dfsx[u].second, dfs2(v, u));
    }
    return dfsx[u].second;
    };
dfs2(0, -1);
```

prufur 序列

对树建立 Prüfer 序列

Prüfer 是这样建立的：每次选择一个编号最小的叶结点并删掉它，然后在序列中记录下它连接到的那个结点。重复 $n-2$ 次后就只剩下两个结点，算法结束。

显然使用堆可以做到 $O(n\log n)$ 的复杂度

```
// 代码摘自原文，结点是从 0 标号的
vector<vector<int>> adj;

vector<int> pruefer_code() {
  int n = adj.size();
  set<int> leafs;
  vector<int> degree(n);
  vector<bool> killed(n, false);
  for (int i = 0; i < n; i++) {
    degree[i] = adj[i].size();
    if (degree[i] == 1) leafs.insert(i);
  }

  vector<int> code(n - 2);
  for (int i = 0; i < n - 2; i++) {
    int leaf = *leafs.begin();
    leafs.erase(leafs.begin());
    killed[leaf] = true;
    int v;
    for (int u : adj[leaf])
      if (!killed[u]) v = u;
    code[i] = v;
    if (--degree[v] == 1) leafs.insert(v);
  }
  return code;
}
```

```python
# 结点是从 0 标号的
adj = [[]]


def pruefer_code():
    n = len(adj)
    leafs = set()
    degree = [0] * n
    killed = [False] * n
    for i in range(1, n):
        degree[i] = len(adj[i])
        if degree[i] == 1:
            leafs.intersection(i)
    code = [0] * (n - 2)
    for i in range(1, n - 2):
        leaf = leafs[0]
        leafs.pop()
        killed[leaf] = True
        for u in adj[leaf]:
            if killed[u] == False:
                v = u
        code[i] = v
        if degree[v] == 1:
            degree[v] = degree[v] - 1
            leafs.intersection(v)
    return code
```

Cayley 公式 (Cayley's formula)

完全图 K_n 有 n^{n-2} 棵生成树。

怎么证明？方法很多，但是用 Prüfer 序列证是很简单的。任意一个长度为 $n-2$ 的值域 $[1,n]$ 的整数序列都可以通过 Prüfer 序列双射对应一个生成树，于是方案数就是 n^{n-2}。

图连通方案数

Prüfer 序列可能比你想得还强大。它能创造比 凯莱公式 更通用的公式。比如以下问题：

> 一个 n 个点 m 条边的带标号无向图有 k 个连通块。我们希望添加 $k-1$ 条边使得整个图连通。求方案数。

设 s_i 表示每个连通块的数量。我们对 k 个连通块构造 Prüfer 序列，然后你发现这并不是普通的 Prüfer 序列。因为每个连通块的连接方法很多。不能直接淦就设啊。于是设 d_i 为第 i 个连通块的度数。由于度数之和是边数的两倍，于是 $\sum_{i=1}^{k} d_i = 2k-2$。则对于给定的 d 序列构造 Prüfer 序列的方案数是

$$n^{k-2} \cdot \prod_{i=1}^{k} s_i$$

网络流

最大流

```
constexpr int inf = 1E9;
template<class T>
struct MaxFlow {
    struct _Edge {
        int to;
        T cap;
        _Edge(int to, T cap) : to(to), cap(cap) {}
    };

    int n;
    std::vector<_Edge> e;
    std::vector<std::vector<int>> g;
    std::vector<int> cur, h;

    MaxFlow() {}
    MaxFlow(int n) {
        init(n);
    }

    void init(int n) {
        this->n = n;
        e.clear();
        g.assign(n, {});
        cur.resize(n);
        h.resize(n);
    }

    bool bfs(int s, int t) {
        h.assign(n, -1);
        std::queue<int> que;
        h[s] = 0;
        que.push(s);
        while (!que.empty()) {
            const int u = que.front();
            que.pop();
            for (int i : g[u]) {
                auto [v, c] = e[i];
                if (c > 0 && h[v] == -1) {
```

```cpp
                h[v] = h[u] + 1;
                if (v == t) {
                    return true;
                }
                que.push(v);
            }
        }
    }
    return false;
}

T dfs(int u, int t, T f) {
    if (u == t) {
        return f;
    }
    auto r = f;
    for (int &i = cur[u]; i < int(g[u].size()); ++i) {
        const int j = g[u][i];
        auto [v, c] = e[j];
        if (c > 0 && h[v] == h[u] + 1) {
            auto a = dfs(v, t, std::min(r, c));
            e[j].cap -= a;
            e[j ^ 1].cap += a;
            r -= a;
            if (r == 0) {
                return f;
            }
        }
    }
    return f - r;
}
void addEdge(int u, int v, T c) {
    g[u].push_back(e.size());
    e.emplace_back(v, c);
    g[v].push_back(e.size());
    e.emplace_back(u, 0);
}
T flow(int s, int t) {
    T ans = 0;
    while (bfs(s, t)) {
        cur.assign(n, 0);
        ans += dfs(s, t, std::numeric_limits<T>::max());
    }
```

```
        return ans;
    }

    std::vector<bool> minCut() {
        std::vector<bool> c(n);
        for (int i = 0; i < n; i++) {
            c[i] = (h[i] != -1);
        }
        return c;
    }

    struct Edge {
        int from;
        int to;
        T cap;
        T flow;
    };
    std::vector<Edge> edges() {
        std::vector<Edge> a;
        for (int i = 0; i < e.size(); i += 2) {
            Edge x;
            x.from = e[i + 1].to;
            x.to = e[i].to;
            x.cap = e[i].cap + e[i + 1].cap;
            x.flow = e[i + 1].cap;
            a.push_back(x);
        }
        return a;
    }
};
```

two-sat

2sat方案计数为NPC问题

```cpp
struct TwoSat {
    int n;
    std::vector<std::vector<int>> e;
    std::vector<bool> ans;
    TwoSat(int n) : n(n), e(2 * n), ans(n) {}
    void addEdge(int u, bool f, int v, bool g) {
        e[2 * u + f].push_back(2 * v + g);
    }
    void addClause(int u, bool f, int v, bool g) {
        addEdge(u, !f, v, g);
        addEdge(v, !g, u, f);
    }
    void notClause(int u, bool f, int v, bool g) {
        addClause(u, !f, v, !g);
    }
    bool satisfiable() {
        std::vector<int> id(2 * n, -1), dfn(2 * n, -1), low(2 * n, -1);
        std::vector<int> stk;
        int now = 0, cnt = 0;
        std::function<void(int)> tarjan = [&](int u) {
            stk.push_back(u);
            dfn[u] = low[u] = now++;
            for (auto v : e[u]) {
                if (dfn[v] == -1) {
                    tarjan(v);
                    low[u] = std::min(low[u], low[v]);
                }
                else if (id[v] == -1) {
                    low[u] = std::min(low[u], dfn[v]);
                }
            }
            if (dfn[u] == low[u]) {
                int v;
                do {
                    v = stk.back();
                    stk.pop_back();
                    id[v] = cnt;
                } while (v != u);
                ++cnt;
            }
        };
        for (int i = 0; i < 2 * n; ++i) if (dfn[i] == -1) tarjan(i);
        for (int i = 0; i < n; ++i) {
```

```
            if (id[2 * i] == id[2 * i + 1]) return false;
            ans[i] = id[2 * i] > id[2 * i + 1];
        }
        return true;
    }
    std::vector<bool> answer() { return ans; }
};
```