

Table of Contents

Introduction	1.1
图论	1.2
dfs	1.2.1
双联通分量	1.2.1.1
强连通分量	1.2.1.2
2-SAT	1.2.1.3
二分图	1.2.1.4
最大匹配	1.2.1.4.1
完美匹配	1.2.1.4.2
最短路	1.2.1.5
dijkstra	1.2.1.5.1
bellman-ford	1.2.1.5.2
生成树	1.2.1.6
生成树计数	1.2.1.6.1
网络流	1.2.1.7
最大流	1.2.1.7.1
上下界网络流	1.2.1.7.2
最小费用流	1.2.1.7.3
启发式合并	1.2.1.8

图论

深度优先遍历

割点

1. 树根是割点当且仅当它有两个及以上的孩子
2. 非根节点 u 是割点当且仅当 u 存在一个子节点 v , v 及其子节点都没有反向边连向 u 的子节点

```
int tot, dfs_clock, bcc_cnt;
int to[M], nxt[M], head[N], dfn[N], low[N], iscut[N], bccno[N];

void init() {
    tot = dfs_clock = bcc_cnt = 0; memset(head, 0, sizeof(head));
    memset(dfn, 0, sizeof(dfn)); memset(iscut, 0, sizeof(iscut));
    memset(bccno, 0, sizeof(bccno));
}

void addEdge(int u, int v) {
    to[++tot] = v, nxt[tot] = head[u], head[u] = tot;
}

void dfs(int u, int fa = -1) {
    dfn[u] = low[u] = ++dfs_clock;
    int child = 0;
    for(int i = head[u]; i; i = nxt[i]) {
        int v = to[i];
        if(!dfn[v]) {
            child++;
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if(low[v] >= dfn[u]) iscut[u] = true;
        } else if(dfn[v] < dfn[u] && v != fa)
            low[u] = min(low[u], dfn[v]);
    }
    if(fa < 0 && child == 1) iscut[u] = 0;
}
```

点-双联通分量

不同双联通分量之间最多只有一个公共点，且一定是割点。计算点双联通分量的过程和计算割点类似，用一个栈来保存在当前BCC中的边。

```
int tot, dfs_clock, bcc_cnt;
int to[M], nxt[M], head[N], dfn[N], low[N], iscut[N], bccno[N];
stack<pair<int, int>> stk;
vector<int> bcc[N];

void init() {
    tot = dfs_clock = bcc_cnt = 0; memset(head, 0, sizeof(head));
    memset(dfn, 0, sizeof(dfn)); memset(iscut, 0, sizeof(iscut));
    memset(bccno, 0, sizeof(bccno));
}

void addEdge(int u, int v) {
```

```

    to[++tot] = v, nxt[tot] = head[u], head[u] = tot;
}

void dfs(int u, int fa = -1) {
    dfn[u] = low[u] = ++dfs_clock;
    int child = 0;
    for(int i = head[u]; i; i = nxt[i]) {
        int v = to[i];
        if(!dfn[v]) {
            stk.push(make_pair(u, v)), child++;
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if(low[v] >= dfn[u]) {
                iscut[u] = true; bcc_cnt++;
                bcc[bcc_cnt].clear();
                for(;;) {
                    int a = stk.top().first, b = stk.top().second; stk.pop();
                    if(bccno[a] != bcc_cnt) {
                        bcc[bcc_cnt].push_back(a); bccno[a] = bcc_cnt;
                    }
                    if(bccno[b] != bcc_cnt) {
                        bcc[bcc_cnt].push_back(b); bccno[b] = bcc_cnt;
                    }
                    if(a == u && b == v) break ;
                }
            }
        } else if(dfn[v] < dfn[u] && v != fa) {
            stk.push(make_pair(u, v));
            low[u] = min(low[u], dfn[v]);
        }
    }
    if(fa < 0 && child == 1) iscut[u] = 0;
}

```

割边

对于一条边 (u, v) ，若 v 及其后代都不能到达 u 及其祖先，则 (u, v) 是割边

```

void addEdge(int u, int v, int id) {
    to[++tot] = v, nxt[tot] = head[u], index[tot] = id, head[u] = tot;
}

void dfs(int u, int from = 0) {
    dfn[u] = low[u] = ++dfs_clock;
    for(int i = head[u]; i; i = nxt[i]) if(index[i] != from){
        int v = to[i];
        if(!dfn[v]) {
            dfs(v, index[i]);
            low[u] = min(low[u], low[v]);
            if(low[v] > dfn[u]) iscut[index[i]] = true;
        } else if(dfn[v] < dfn[u])
            low[u] = min(low[u], dfn[v]);
    }
}

```

```
    }  
}
```

边-双联通分量

求出割边之后，在原图中**dfs**，每个联通块为一个边-双联通分量

```
void getIdx(int u, int id) {  
    bccno[u] = id;  
    for(int i = head[u]; i; i = nxt[i])  
        if(!iscut[index[i]] && !bccno[to[i]]) getIdx(to[i], id);  
}
```

强连通分量

```
void init() {
    tot = dfs_clock = scc_cnt = 0;
    memset(head, 0, sizeof(head)); memset(dfn, 0, sizeof(dfn));
    memset(sccno, 0, sizeof(sccno));
}

void addEdge(int u, int v) {
    to[++tot] = v, nxt[tot] = head[u], head[u] = tot;
}

void tarjan(int u) {
    dfn[u] = low[u] = ++dfs_clock;
    S.push(u);
    for(int i = head[u]; i; i = nxt[i]) {
        int v = to[i];
        if(!dfn[v]) tarjan(v), low[u] = min(low[u], low[v]);
        else if(!sccno[v]) low[u] = min(low[u], dfn[v]);
    }
    if(low[u] == dfn[u]) {
        scc_cnt++;
        for(;;) {
            int x = S.top(); S.pop();
            sccno[x] = scc_cnt;
            if(x == u) break ;
        }
    }
}
```


2-SAT

字典序

逐一考虑每个没有赋值的变量，先假定为真，然后沿有向边标记所有能标记的节点，如果过程中发现某变量的两个对立节点都被标记，则不能为真。然后假定为假，再次进行标记，如果还是不能标记，则无解，不需要回溯。

```
/* poi2001 和平委员会
根据宪法, Byteland 民主共和国的公众和平委员会应该在国会中通过立法程序来创立。不幸的是, 由于某些党派代表之间的不和睦而使得这件事存在障碍。

此委员会必须满足下列条件:
每个党派都在委员会中恰有1个代表,
如果2个代表彼此厌恶, 则他们不能都属于委员会。

每个党在议会中有2个代表。代表从1编号到2n。 编号为2i-1和2i的代表属于第i个党派。
*/

int tot, c;
int to[M], nxt[M], head[N], mark[N], S[N];

void addEdge(int u, int v) {
    to[++tot] = v, nxt[tot] = head[u], head[u] = tot;
}

void init() {
    tot = 0; memset(mark, 0, sizeof(mark));
    memset(head, 0, sizeof(head));
}

bool dfs(int u) {
    if(mark[u^1]) return false;
    if(mark[u]) return true;
    mark[u] = true, S[c++] = u;
    for(int i = head[u]; i; i = nxt[i]) if(!dfs(to[i])) return false;
    return true;
}

void twoSAT(int n) {
    for(int i = 0; i < 2*n; i += 2) if(!mark[i] && !mark[i^1]) {
        c = 0;
        if(!dfs(i)) {
            while(c > 0) mark[S[--c]] = false;
            if(!dfs(i+1)) {
                printf("NIE\n"); return ;
            }
        }
    }
    for(int i = 0; i < 2*n; i++) if(mark[i]) printf("%d\n", i+1);
}
```

```

int main() {
    int n, m;
    while(scanf("%d%d", &n, &m) != EOF) {
        init();
        for(int i = 1, a, b; i <= m; i++) {
            scanf("%d%d", &a, &b);
            a--, b--;
            addEdge(a, b^1); addEdge(b, a^1);
        }
        twoSAT(n);
    }
}

```

强连通分量

对于每个析取范式 $a \vee b$ ，连两条边 $(\neg a, b)$ 和 $(\neg b, a)$ ，然后求强连通分量，进行缩点

如果存在两个对立的变量在一个强连通分量里面，则无解，否则必有解

缩点之后的DAG里面，对于两个对立变量 a 和 b ，若 a 的拓扑序在后边，则 a 为真

```

bool twoSAT(int n) {
    for(int i = 1; i <= 2*n; i++) if(!dfn[i]) tarjan(i);
    for(int i = 1; i <= n; i++) if(sccno[i] == sccno[i+n]) {
        printf("NO\n"); return ;
    };
    printf("YES\n");
    for(int i = 1; i <= n; i++) {
        if(sccno[i] < sccno[i+n]) printf("true\n");
        else printf("false\n");
    }
}

```

二分图

匈牙利算法

```
bool hungary(int u) {
    vis[u] = clk;
    int v;
    for(int i = head[u]; i; i = nxt[i]) if(vis[v = to[i]] != clk) {
        vis[v] = clk;
        if(!xlink[v] || hungary(ylink[v])) {
            xlink[u] = v, ylink[v] = u;
            return true;
        }
    }
    return false;
}
```

Hopcroft-Karp

```
int n, m, w;
int vis[N], xlink[N], ylink[N], dis[N];
vector<int>G[N];

bool BFS() {
    queue<int>q;
    memset(dis, -1, sizeof(dis));
    for(int i = 1; i <= m; i++)
        if(xlink[i] == -1) {
            q.push(i); dis[i] = 0;
        }
    w = INT_INF;
    while(!q.empty()) {
        int p = q.front(); q.pop();
        if(dis[p] > w) break;
        for(int i = 0; i < G[p].size(); i++) {
            int l = G[p][i];
            if(dis[l] == -1) {
                dis[l] = dis[p] + 1;
                if(ylink[l] == -1) w = dis[l];
            }
            else {
                dis[ylink[l]] = dis[l] + 1;
                q.push(ylink[l]);
            }
        }
    }
    return w != INT_INF;
}

int Find(int u) {
    for(int i = 0; i < G[u].size(); i++) {
        int l = G[u][i];
```

```

        if(!vis[l] && dis[l] == dis[u] + 1) {
            vis[l] = 1;
            if(ylink[l] != -1 && dis[l] == w) continue;
            if(ylink[l] == -1 || Find(ylink[l])) {
                xlink[u] = l, ylink[l] = u;
                return 1;
            }
        }
    }
    return 0;
}

int main() {
    scanf("%d%d", &n, &m);
    int u, v;
    while(scanf("%d%d", &u, &v) != EOF) G[u].push_back(v);
    memset(xlink, -1, sizeof(xlink));
    memset(ylink, -1, sizeof(ylink));
    int ans = 0;
    while(BFS()) {
        memset(vis, 0, sizeof(vis));
        for(int i = 1; i <= m; i++)
            if(xlink[i] == -1) ans += Find(i);
    }
    printf("%d\n", ans);
}

```

最佳完美匹配

```
bool match(int u) {
    S[u] = true;
    for(int v = 1; v <= n; v++) if(!T[v]) {
        int gap = lx[u]+ly[v]-w[u][v];
        if(gap == 0) {
            T[v] = true;
            if(!ylink[v] || match(ylink[v])) {
                xlink[u] = v, ylink[v] = u;
                return true;
            }
        } else {
            slack[v] = min(slack[v], gap);
        }
    }
    return false;
}

void update() {
    int a = inf;
    for(int i = 1; i <= n; i++) if(!T[i]) a = min(a, slack[i]);
    for(int i = 1; i <= n; i++) {
        if(S[i]) lx[i] -= a;
        if(T[i]) ly[i] += a;
        else slack[i] -= a;
    }
}

void KM() {
    for(int i = 1; i <= n; i++) {
        xlink[i] = ylink[i] = lx[i] = ly[i] = 0;
        for(int j = 1; j <= n; j++) lx[i] = max(lx[i], w[i][j]);
    }
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= n; j++) slack[j] = inf;
        while(true) {
            for(int j = 1; j <= n; j++) S[j] = T[j] = 0;
            if(match(i)) break; else update();
        }
    }
}
```

生成树计数

Kirchhoff矩阵：对于无向图 G ，它的**Kirchhoff**矩阵为 G 对应的度数矩阵减去它的邻接矩阵

Matrix-Tree定理：对于无向图 G ，它的生成树个数等于其**Kirchhoff**矩阵任何一个 $n - 1$ 阶主子式的行列式的绝对值

```
LL det(LL A[N][N], int n) {
    int t = 0;
    for(int i = 1; i <= n; i++) {
        for(int j = i+1; j <= n; j++) while(A[j][i]) {
            LL x = A[i][i]/A[j][i];
            t++;
            for(int k = 1; k <= n; k++)
                A[i][k] = (A[i][k]-x*A[j][k]%MOD+MOD)%MOD;
            for(int k = 1; k <= n; k++)
                swap(A[i][k], A[j][k]);
        }
    }
    LL ans = 1;
    for(int i = 1; i <= n; i++) ans = (ans*A[i][i])%MOD;
    if(t&1) ans = -ans;
    return (ans%MOD+MOD)%MOD;
}
```

网络流

最大流

```
const int N = 4100, M = 200010;
int tot = 1, src, sink;
int to[M], _next[M], cap[M], head[N], pre[N], vis[N], cur[N], used[N];

void addEdge(int u, int v, int c) {
    to[++tot] = v, _next[tot] = head[u], cap[tot] = c, head[u] = tot;
}

void init() {
    tot = 1; memset(head, 0, sizeof(head));
}

bool BFS() {
    memset(vis, false, sizeof(vis));
    queue<int>q; q.push(src);
    vis[src] = true;
    while(!q.empty()) {
        int u = q.front(); q.pop();
        for(int i = head[u]; i; i = _next[i]) {
            int v = to[i];
            if(!cap[i] || vis[v]) continue;
            vis[v] = true, pre[v] = pre[u]+1;
            q.push(v);
        }
    }
    return vis[sink];
}

int DFS(int u, int c) {
    if(u==sink || c==0) return c;
    int flow = 0, f;
    for(int &i = cur[u]; i; i = _next[i]) {
        int v = to[i];
        if(pre[v]==pre[u]+1 && (f=DFS(v, min(c, cap[i])))>0) {
            flow += f, c -= f, cap[i] -= f, cap[i^1] += f;
        }
    }
    return flow;
}

int maxFlow() {
    int flow = 0;
    while(BFS()) {
        memcpy(cur, head, sizeof(cur));
        flow += DFS(src, INT_INF);
    }
    return flow;
}
```


有上下界网络流

无源汇有上下界可行流

如果存在可行流，那么每条边的流量都大于等于流量的下界，因此可以令每条边的初始流量为流量的下界，在此基础上建立残量网络。初始流不一定满足流量守恒，考虑在残量网络上求出另一个不守恒的附加流。

1. 对于原图中每条 (u, v, l, r) 的边，在新图中建立 $(u, v, r-l)$ 的边
2. $du[i]$ 表示初始流中节点 i 的流入流量与流出流量之差
3. 对于新图，建立超级源点 $source$ 与汇点 $sink$
4. 对于满足 $du[i] > 0$ 的节点 i ，在新图中建立 $(source, i, du[i])$ 的边
5. 对于满足 $du[i] < 0$ 的节点 i ，在信徒中建立 $(i, sink, -du[i])$ 的边

若新图可以满流，则存在一个可行流，否则不存在。

```
bool lowbound_flow(int n, vector<int>U, vector<int>V, vector<int>L, vector<int>R) {
    dinic::init();
    memset(du, 0, sizeof(du));
    int ln = U.size();
    for(int i = 0; i < ln; i++) {
        if(R[i] < L[i]) return 0;
        dinic::addEdge(U[i], V[i], R[i]-L[i], i+1);
        dinic::addEdge(V[i], U[i], 0, 0);
        du[U[i]] -= L[i], du[V[i]] += L[i];
    }
    dinic::src = n+1, dinic::sink = n+2;
    int sum = 0;
    for(int i = 0; i <= n; i++) {
        if(du[i] > 0) {
            dinic::addEdge(dinic::src, i, du[i], 0);
            dinic::addEdge(i, dinic::src, 0, 0);
            sum += du[i];
        } else if(du[i] < 0) {
            dinic::addEdge(i, dinic::sink, -du[i], 0);
            dinic::addEdge(dinic::sink, i, 0, 0);
        }
    }
    return dinic::maxFlow() == sum;
}
```

有源汇有上下界可行流

连接一条 $(sink, src, inf)$ 的边，问题转化为无源汇有上下界可行流

有源汇有上下界最大流

先求一个有源汇有上下界可行流，然后再在原来的残量网络上面进行增广，最后的最大流即为可行流（原图中 $sink$ 到 src 的流量） $+$ 残量网络最大流

```
int lowboundMaxflow(int s, int t, int n, vector<int>U, vector<int>V, vector<int>L,
vector<int>R) {
    memset(du, 0, sizeof(du)); dinic::init();
```

```

int m = U.size();
dinic::src = n+1, dinic::sink = n+2;
for(int i = 0; i < m; i++) {
    if(L[i] > R[i]) return -1;
    dinic::addEdge(U[i], V[i], R[i]-L[i]);
    dinic::addEdge(V[i], U[i], 0);
    du[U[i]] -= L[i], du[V[i]] += L[i];
}
int sum = 0;
for(int i = 0; i <= n; i++) {
    if(du[i] > 0) {
        dinic::addEdge(dinic::src, i, du[i]);
        dinic::addEdge(i, dinic::src, 0);
        sum += du[i];
    } else if(du[i] < 0) {
        dinic::addEdge(i, dinic::sink, -du[i]);
        dinic::addEdge(dinic::sink, i, 0);
    }
}
dinic::addEdge(t, s, INF), dinic::addEdge(s, t, 0);
if(dinic::maxFlow() < sum) return -1;
else {
    int flow = dinic::cap[dinic::tot];
    dinic::cap[dinic::tot] = dinic::cap[dinic::tot-1] = 0;
    for(int i = dinic::head[dinic::src]; i; i = dinic::nxt[i])
        dinic::cap[i] = dinic::cap[i^1] = 0;
    for(int i = dinic::head[dinic::sink]; i; i = dinic::nxt[i])
        dinic::cap[i] = dinic::cap[i^1] = 0;
    dinic::src = s, dinic::sink = t;
    return flow + dinic::maxFlow();
}
}

```

有源汇有上下界最小流

求完可行流之后在残量网络上进行从 s 到 t 的最大流，可行流减去 s 到 t 的最大流即为最小流

最小费用最大流

```
void addEdge(int u, int v, int w, int c) {
    to[++tot] = v, nxt[tot] = head[u], cap[tot] = w, cost[tot] = c, head[u] = tot;
}

bool spfaMin(int &c, int &f) {
    for(int i = src; i <= sink; i++) d[i] = INT_INF, inq[i]=0;
    queue<int>q; q.push(src);
    d[src] = 0, inq[src] = 1, pre[src] = 0, a[src] = INT_INF;
    while(!q.empty()) {
        int p = q.front(); q.pop();
        inq[p] = 0;
        for(int i = head[p]; i; i = nxt[i]) {
            int l = to[i];
            if(cap[i] && d[l] > d[p] + cost[i]) {
                d[l] = d[p] + cost[i]; pre[l] = i; a[l] = min(a[p], cap[i]);
                if(!inq[l]) q.push(l); inq[l] = 1;
            }
        }
    }
    if(d[sink] == INT_INF) return false;
    c += d[sink]*a[sink], f += a[sink];
    int u = sink;
    while(u != src) {
        cap[pre[u]] -= a[sink], cap[pre[u]^1] += a[sink];
        u = to[pre[u]^1];
    }
    return true;
}
```

最大费用最大流

```
int tot = 1, src, sink;
int to[M], nxt[M], cap[M], cost[M], head[N], d[N], a[N], inq[N], pre[N];

void addEdge(int u, int v, int w, int c) {
    to[++tot] = v, nxt[tot] = head[u], cap[tot] = w, cost[tot] = c, head[u] = tot;
}

bool spfaMax(int &c, int &f) {
    for(int i = src; i <= sink; i++) d[i] = -INT_INF, inq[i]=0;
    queue<int>q; q.push(src);
    d[src] = 0, inq[src] = 1, pre[src] = 0, a[src] = INT_INF;
    while(!q.empty()) {
        int p = q.front(); q.pop();
        inq[p] = 0;
        for(int i = head[p]; i; i = nxt[i]) {
            int l = to[i];
            if(cap[i] && d[l] < d[p] + cost[i]) {
```

```

        d[l] = d[p] + cost[i]; pre[l] = i; a[l] = min(a[p], cap[i]);
        if(!inq[l]) q.push(l); inq[l] = 1;
    }
}
if(d[sink] == -INT_INF) return false;
c += d[sink]*a[sink], f += a[sink];
int u = sink;
while(u != src) {
    cap[pre[u]] -= a[sink], cap[pre[u]^1] += a[sink];
    u = to[pre[u]^1];
}
return true;
}

```