# 图论

## 单源最短路

拿到题目，请按以下顺序问自己问题：

1. **图中有负权边吗?**
    - **没有** (所有边权 $\geq 0$) $\rightarrow$ **Dijkstra (堆优化版)** 【首选，最稳】
    - **有** $\rightarrow$ 进入下一步。
2. **图是有向无环图 (DAG) 吗?**
    - **是** (比如任务调度、层级结构) $\rightarrow$ **基于拓扑排序的 DP** 【最快，线性时间】
    - **不是** (有环，或者不知道有没有环) $\rightarrow$ 进入下一步。
3. **需要判断负环，或者由于负边无法使用 Dijkstra?**
    - 一般情况 $\rightarrow$ **SPFA** 【平均很快，但可能被卡】
    - 数据量极小 ($V \leq 500$) 或限制极其严格 $\rightarrow$ **Bellman-Ford** 【最慢，但绝对正确】

| 算法 | 是否支持负权边 | 是否支持负环 | 适用结构 | 时间复杂度 (V点 E边) | 评价 |
| --- | --- | --- | --- | --- | --- |
| **Dijkstra** (堆优化) | ❌ 不支持 | ❌ | 任意图 (无负边) | $O(E \log V)$ | **正权图必选**，稳定高效 |
| **SPFA** | ✅ 支持 | ✅ (可检测) | 任意图 | 平均 $O(kE)$<br>最坏 $O(VE)$ | **负权图首选**，容易被恶意数据卡死 |
| **Bellman-Ford** | ✅ 支持 | ✅ (可检测) | 任意图 | $O(VE)$ | 太慢，通常作为 SPFA 的理论基础 |
| **DAG Topo** | ✅ 支持 | ❌ (图本身无环) | **仅 DAG** | $O(V+E)$ | **理论最快**，但适用范围窄 |

### SPFA

```cpp
#include <iostream>
#include <cstring>
#include <queue>
#include <algorithm>

using namespace std;

// ==== 你原本的结构定义 ====
const int N = 10005;  // 最大点数 V_MAX
const int M = 20005;  // 最大边数 E_MAX (无向图要 x2)
const int INF = 0x3f3f3f3f;

struct Edge {
```

```
    int to, w, next;
} e[M]; // 注意：这里用 M

int cnt_E = 0;
int head[N]; // 注意：这里用 N

// 初始化函数 (多组数据时必须调用)
void initList() {
    memset(head, -1, sizeof(head));
    cnt_E = 0; // 别忘了重置边计数器!
}

void addEdge(int x, int y, int w) {
    e[cnt_E].to = y;
    e[cnt_E].w = w;
    e[cnt_E].next = head[x];
    head[x] = cnt_E++;
}

// ==== SPFA 核心板子 ====
int dist[N];      // 存储起点到各点的最短距离
bool in_queue[N]; // 标记是否在队列中 (SPFA的核心优化)
int cnt_update[N]; // (可选) 用于判断负环：记录每个点入队次数

// 返回 true 表示成功求出最短路
// 返回 false 表示图中存在“负环”，无法求最短路
bool spfa(int s, int n) { // s: 起点, n: 总点数(用于判负环)
    // 1. 初始化
    memset(dist, 0x3f, sizeof(dist)); // 初始化为无穷大
    memset(in_queue, false, sizeof(in_queue));
    memset(cnt_update, 0, sizeof(cnt_update));

    // 2. 起点入队
    dist[s] = 0;
    queue<int> q;
    q.push(s);
    in_queue[s] = true;

    // 3. 循环松弛
    while (!q.empty()) {
        int u = q.front(); q.pop();
        in_queue[u] = false; // 出队后标记为不在队列

        // 遍历所有出边
        for (int i = head[u]; i != -1; i = e[i].next) {
            int v = e[i].to;
            int w = e[i].w;

            // 【松弛操作】如果走 u->v 比以前的路径更短
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;

                // 判负环逻辑 (如果不需要判负环，可以删掉下面这块)
```

```
                cnt_update[v] = cnt_update[u] + 1;
                if (cnt_update[v] >= n) return false; // 存在负环

                // 如果 v 不在队列中，才加入队列
                if (!in_queue[v]) {
                    q.push(v);
                    in_queue[v] = true;
                }
            }
        }
    }
    return true; // 正常结束
}

```

# Dinic - 最大流

Dinic 的核心逻辑是：**不断重复“分层（BFS）”和“多路增广（DFS）”这两个过程，直到无法到达终点。**

---

第一阶段：（BFS 分层）

**目的**：在残量网络中，给每个点打上“层级”标签（`dep[]` 或 `level[]`），标记它离起点 $S$ 有多远（最短路径步数）。

**规则**：

1. **只走有剩余容量的边**（`cap > 0`）。
2. 起点 $S$ 的层级为 1（或 0），它的邻居是 2，邻居的邻居是 3.....
3. 如果无法从 $S$ 走到 $T$（即 $T$ 没被标记层级），说明网络断了，**算法结束，输出最大流。**

**为什么要做这一步？** 为了**限制 DFS 不乱跑。**我们在 DFS 时严格要求：只能从第 $i$ 层走到第 $i+1$ 层。这保证了我们在走最短路，防止在圈里打转。

> **关键动作**：每次 BFS 结束，顺便把 `cur[]` 数组重置，让它指回 `head[]`（所谓“将书签拨回第一页”）。

---

第二阶段：（DFS 多路增广）

**目的**：在第一阶段划定的“层级地图”上，尽可能多地把流量从 $S$ 推到 $T$。这被称为寻找**阻塞流（Blocking Flow）**。

**流程**：从起点 $S$ 开始 DFS：

1. **检查路标**：看当前点 $u$ 的 `cur[u]` 指向哪条边。
2. **筛选路径**：这条边 $(u, v)$ 必须满足两个条件：
    - 有容量（`cap > 0`）。
    - 是通向下一层的（`dep[v] == dep[u] + 1`）。
3. **递归推进**：如果满足，就递归计算 $v$ 能往后推多少流（`dfs(v, limit)`）。
4. **更新容量**：

- 假设算出来能推 `flow` 的流量。
- **正向边**减去 `flow`。
- **反向边**加上 `flow`（这是网络流反悔机制的核心）。
- $u$ 节点的剩余推流能力减去 `flow`。

5. **当前弧优化（关键）**：
   - 如果某条边 $(u, v)$ 被榨干了（或者 $v$ 后面走不通了），**修改 `cur[u]` 指向下一条边。**
   - 下次再来到 $u$，直接看下一条边，绝不回头。
6. **剪枝**：如果 $u$ 发现所有邻居都帮不了忙（流推不出去），把 `dep[u]` 设为 -1（或者其它无效值），把这个点从分层图中"删掉"，免得下次别的点又跑来问它。

---

第三阶段：循环

**逻辑**：

1. BFS 建立分层图。
2. 如果 $T$ 还有层级（能走到），就跑 DFS 榨干这一层级图。
3. 榨干后，原来的短路肯定断了，回到第 1 步，重新跑 BFS 建立新的（更长的）分层图。
4. 重复，直到 BFS 找不到去 $T$ 的路。

```cpp
#include <iostream>
#include <cstring>
#include <queue>
#include <algorithm>

using namespace std;

// ================== 参数配置 ==================
const int MAXN = 10005;  // 最大点数
const int MAXM = 200005; // 最大边数（注意：网络流需要反向边，所以数组大小要是题目边数的2倍！）
const int INF = 0x3f3f3f3f;

// ================== 链式前向星结构 ==================
struct Edge {
    int to, w, next;
} e[MAXM];

int head[MAXN];
int cnt_E = 0;

// Dinic特需数组
int dep[MAXN]; // 深度数组（分层图）
int cur[MAXN]; // 当前弧优化数组（记录DFS遍历到了哪条边）

// 初始化函数
void init() {
    memset(head, -1, sizeof(head));
    cnt_E = 0;
}
```

```
30
31  // 加边函数：同时加入 正向边(cap) 和 反向边(0)
32  // 注意：网络流中通常由 addFlowEdge 调用底层的 addEdge
33  void addEdge(int u, int v, int w) {
34      e[cnt_E].to = v;
35      e[cnt_E].w = w;
36      e[cnt_E].next = head[u];
37      head[u] = cnt_E++;
38  }
39
40  // 对外调用的加边函数
41  void addFlowEdge(int u, int v, int w) {
42      addEdge(u, v, w); // 正向边，索引为偶数 (如 0)
43      addEdge(v, u, 0); // 反向边，索引为奇数 (如 1)
44  }
45
46  // ================== Dinic 核心部分 ==================
47
48  // BFS: 构建分层图，判断是否能到达汇点 T
49  bool bfs(int S, int T) {
50      memset(dep, -1, sizeof(dep)); // 初始化深度为 -1
51      queue<int> q;
52
53      dep[S] = 0;
54      q.push(S);
55
56      // 每次BFS前，把 head 复制给 cur，重置当前弧
57      // 也可以放在 dinic 主函数循环里，但这里写不容易忘
58      memcpy(cur, head, sizeof(head));
59
60      while (!q.empty()) {
61          int u = q.front();
62          q.pop();
63
64          for (int i = head[u]; i != -1; i = e[i].next) {
65              int v = e[i].to;
66              int w = e[i].w;
67
68              // 如果有剩余容量 且 未被访问过
69              if (w > 0 && dep[v] == -1) {
70                  dep[v] = dep[u] + 1;
71                  q.push(v);
72              }
73          }
74      }
75      return dep[T] != -1; // 如果汇点被标记深度，说明有路
76  }
77
78  // DFS: 多路增广
79  // u: 当前节点, limit: 当前路径流过来的最小流量限制, T: 汇点
80  int dfs(int u, int limit, int T) {
81      if (u == T || limit == 0) return limit; // 到达汇点或无流量
82
```

```
    int flow = 0; // 本次DFS能从u推出去的总流量

    // 【关键】当前弧优化：从 cur[u] 开始遍历，而不是 head[u]
    // 引用 &i 直接修改 cur[u] 的值，下次进这个点直接从下一条边开始
    for (int &i = cur[u]; i != -1; i = e[i].next) {
        int v = e[i].to;
        int w = e[i].w;

        // 必须满足：层级是下一层 且 有剩余容量
        if (dep[v] == dep[u] + 1 && w > 0) {
            // 递归寻找增广量
            int f = dfs(v, min(limit, w), T);

            if (f > 0) {
                e[i].w -= f;        // 正向边减少容量
                e[i ^ 1].w += f;    // 反向边增加容量 (利用异或: 0^1=1, 1^1=0)
                flow += f;          // 累加推出去的流
                limit -= f;         // 剩余可推流量减少

                if (limit == 0) break; // 如果流干了，就不用看后面的边了
            }
        }
    }
    return flow;
}

// Dinic 主入口
int dinic(int S, int T) {
    int max_flow = 0;
    // 只要能建立分层图(有路)，就持续增广
    while (bfs(S, T)) {
        // 注意：bfs里已经重置了 cur 数组
        max_flow += dfs(S, INF, T);
    }
    return max_flow;
}

// ================== 使用示例 ==================
int main() {
    int n, m, S, T; // 点数，边数，源点，汇点
    // 假设输入格式: N M S T
    // 之后 M 行: u v w
    if (cin >> n >> m >> S >> T) {
        init(); // 别忘了初始化
        for (int i = 0; i < m; i++) {
            int u, v, w;
            cin >> u >> v >> w;
            addFlowEdge(u, v, w);
        }
        cout << dinic(S, T) << endl;
    }
    return 0;
}
```

```
136
```

# Dinic实现二分图

这是基于刚才讲解的 **Dinic 算法** 改写的 **二分图最大匹配** 模板。

核心思路：将“匹配”转化为“流”

1. **建立超级源点 $S$ 和超级汇点 $T$**。
2. $S \to$ **左侧点**：连容量为 1 的边（限制每个左侧点只能匹配 1 个）。
3. **右侧点** $\to T$：连容量为 1 的边（限制每个右侧点只能被匹配 1 个）。
4. **左侧点** $\to$ **右侧点**：如果有连边，连容量为 1 的边（表示可以匹配）。
5. **跑 Dinic**：最大流量即为最大匹配数。

此模板的时间复杂度为 $O(E\sqrt{V})$，是二分图匹配中效率极高的算法。

---

C++ 模板代码

```cpp
#include <iostream>
#include <cstring>
#include <queue>
#include <algorithm>

using namespace std;

// ================== 参数配置 ==================
// 二分图匹配中：
// MAXN >= 左边点数 + 右边点数 + 2 (源点+汇点)
// MAXM >= (左边点数 + 右边点数 + 题目给出的边数) * 2 (反向边)
const int MAXN = 2005;   // 假设左1000 + 右1000 + 2
const int MAXM = 200005; // 假设边比较多
const int INF = 0x3f3f3f3f;

// ================== 链式前向星结构 ==================
struct Edge {
    int to, w, next;
} e[MAXM];

int head[MAXN];
int cnt_E = 0; // 这里的 cnt_E 最好初始化为 0 或 -1，配合 memset head -1 使用

// Dinic特需数组
int dep[MAXN];
int cur[MAXN];

// 初始化函数
void init() {
    memset(head, -1, sizeof(head));
    cnt_E = 0;
}

```

```cpp
// 加边函数
void addEdge(int u, int v, int w) {
    e[cnt_E].to = v;
    e[cnt_E].w = w;
    e[cnt_E].next = head[u];
    head[u] = cnt_E++;
}

// 对外调用的加边函数 (正向边w, 反向边0)
void addFlowEdge(int u, int v, int w) {
    addEdge(u, v, w);
    addEdge(v, u, 0);
}

// ================== Dinic 核心部分 (逻辑不变) ==================

bool bfs(int S, int T) {
    memset(dep, -1, sizeof(dep));
    queue<int> q;

    dep[S] = 0;
    q.push(S);

    // 这里的 cur 初始化也可以放在 dinic 函数里，这里写也没问题
    memcpy(cur, head, sizeof(head));

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int i = head[u]; i != -1; i = e[i].next) {
            int v = e[i].to;
            int w = e[i].w;

            if (w > 0 && dep[v] == -1) {
                dep[v] = dep[u] + 1;
                q.push(v);
            }
        }
    }
    return dep[T] != -1;
}

int dfs(int u, int limit, int T) {
    if (u == T || limit == 0) return limit;

    int flow = 0;

    for (int &i = cur[u]; i != -1; i = e[i].next) {
        int v = e[i].to;
        int w = e[i].w;

        if (dep[v] == dep[u] + 1 && w > 0) {
```

```
            int f = dfs(v, min(limit, w), T);

            if (f > 0) {
                e[i].w -= f;
                e[i ^ 1].w += f;
                flow += f;
                limit -= f;

                if (limit == 0) break;
            }
        }
    }
    return flow;
}

int dinic(int S, int T) {
    int max_flow = 0;
    while (bfs(S, T)) {
        max_flow += dfs(S, INF, T);
    }
    return max_flow;
}

// ================== Main: 二分图建图逻辑 ==================
int main() {
    // 优化输入输出速度
    ios::sync_with_stdio(false);
    cin.tie(0);

    int n, m, k; // n:左边点数, m:右边点数, k:给定的边数

    // 读入数据
    if (cin >> n >> m >> k) {
        init(); // 必须初始化!

        // 1. 设定超级源点和超级汇点
        // 习惯上：源点 S=0，左边点 1~n，右边点 n+1~n+m，汇点 T=n+m+1
        int S = 0;
        int T = n + m + 1;

        // 2. 建立 源点 -> 左边点 的边 (容量1)
        for (int i = 1; i <= n; i++) {
            addFlowEdge(S, i, 1);
        }

        // 3. 建立 右边点 -> 汇点 的边 (容量1)
        for (int i = 1; i <= m; i++) {
            // 右边第 i 个点，在图中的真实编号是 i + n
            addFlowEdge(i + n, T, 1);
        }

        // 4. 建立由于题目给出的 左边 -> 右边 的边 (容量1)
        for (int i = 0; i < k; i++) {
```

```
            int u, v;
            cin >> u >> v;
            // 题目输入 u v，表示左边 u 和 右边 v 连边
            // 同样，右边的 v 在图中编号为 v + n
            if (u <= n && v <= m) { // 加上边界检查是个好习惯
                addFlowEdge(u, v + n, 1);
            }
        }

        // 5. 跑 Dinic
        cout << dinic(S, T) << endl;
    }
    return 0;
}

```

代码使用注意事项

1. **节点编号映射**:
   二分图通常给出的是"左边第 $u$ 个"和"右边第 $v$ 个"。
   在建图时，为了避免重号，**右边的点通常映射为** $v+n$。
   - 左边点范围：$[1, n]$
   - 右边点范围：$[n+1, n+m]$
   - $S=0, T=n+m+1$
2. **数组大小**:
   - `MAXN`：至少要大于 $N+M+2$。
   - `MAXE`：至少要大于 $(K+N+M)$，其中 $K$ 是题目给的边数。因为还要加上源点和汇点连出去的边。
3. **时间复杂度**:
   对于这种所有边容量都为 1 的网络（单位网络），Dinic 的复杂度是严格的 $O(E\sqrt{V})$。在二分图匹配问题上，通常比匈牙利算法（$O(VE)$）快很多，尤其是图比较密的时候。

只要题目满足以下条件，Dinic 是**首选**甚至是**唯一解**:

(1) 大规模无权二分图最大匹配

这是 Dinic 的统治区。

- **数据规模**:
  - 点数 $V$ 可达 $10^4 \sim 10^5$。
  - 边数 $E$ 可达 $10^5 \sim 2 \times 10^5$。
- **效率**:
  - **匈牙利算法**：$O(V \cdot E)$。如果 $V=10^4, E=10^5$，运算量约为 $10^9$，会超时（TLE）。
  - **Dinic**：$O(E\sqrt{V})$。同样的规模，运算量大幅下降，通常能稳过。
  - *注：实际上 Dinic 在二分图上的表现等价于 Hopcroft-Karp 算法。*

(2) 多重匹配（Multi-Matching）

- **场景**：如果不只是"一夫一妻"，而是"左边的点 $u$ 最多可以匹配 $L_u$ 个右边的点，右边的点 $v$ 最多可以接受 $R_v$ 个左边的点"。

- **Dinic 优势**：只需在建图时修改源点/汇点的容量即可。
  - $S \to u$ 容量设为 $L_u$。
  - $v \to T$ 容量设为 $R_v$。
  - 匈牙利算法处理这种情况需要拆点，非常麻烦且效率低；Dinic 天然支持。

(3) 带有复杂限制的匹配

- **场景**：比如“点A和点B不能同时被匹配”或者更复杂的流量限制。
- **优势**：由于 Dinic 本质是网络流，你可以通过在这个图中间加额外的点、边和容量限制来通过建模解决复杂的逻辑约束，这是单纯的匹配算法做不到的。

# 匈牙利算法 - 二分图

> 代码：P98
>
> 以下为讲解

为了方便理解，通常把这个算法比喻成 **“相亲配对”** 或 **“找对象”**。

- **左边的点 ($u$)**：男生
- **右边的点 ($v$)**：女生
- **边 ($u \to v$)**：男生 $u$ 对女生 $v$ 有好感（愿意配对）。
- `match[v] = u`：记录女生 $v$ 当前的男朋友是 $u$。

**算法流程（也就是 `dfs` 的过程）：**

假设现在轮到男生 **阿强** 找对象：

1. 阿强看上了 **小红**。
2. **情况一：小红单身**（`match[小红] == 0`）。
   - **结果**：太好了，阿强和小红配对成功！（`match[小红] = 阿强`）。
3. **情况二：小红已经有男朋友了**，男朋友是 **阿珍**（`match[小红] == 阿珍`）。
   - 这时候阿强不会通过打架抢人，而是**尝试协商**。
   - 阿强对小红说：“你能让你男朋友阿珍换个对象吗？如果他能换，你就能腾出来跟我了。”
   - 于是，系统递归去问 **阿珍**（`dfs(阿珍)`）：你还有其他备选吗？
     - 如果阿珍发现他还喜欢 **小丽**，且小丽单身（或者小丽的男朋友也能换人...）。
     - 阿珍就去和小丽配对了。
     - 阿珍腾出了小红。
   - **结果**：阿强和小红配对成功！

O(VE)

# 以下都基于链式前向星

```cpp
/*
 * 参数说明：
 * V_MAX: 最大点数
 * E_MAX: 最大边数 (注意：如果是无向图，E_MAX 需要开边数的2倍)
 * head[]: 存储每个点头部边的索引，初始化为-1
 * Edge结构体: to(终点), w(权值), next(下一条同起点的边)
 */

const int N = 10005;      // 最大点数 (根据题目改)
const int M = 20005;      // 最大边数 (无向图记得 * 2)
const int INF = 0x3f3f3f3f; // 无穷大 (约10亿，memset可用)

struct Edge{
    int to, w, next; //终点，权值，前驱
} e[E_MAX];
int cnt_E = 0;
int head[V_MAX]; //需要先初始化为-1
void initList(int n){
    memset(head, -1, sizeof(head));
}

//  加边函数 (无向图需要调用两遍)
void addEdge(int x, int y, int w) {
    e[cnt_E].to = y;
    e[cnt_E].w = w;
    e[cnt_E].next = head[x];
    head[x] = cnt_E++;
}

//  【核心】遍历模版：访问从 u 点出发的所有边
// 考试时直接套用这个 for 循环
void traverse(int u) {
    // i 代表边的索引
    for (int i = head[u]; i != -1; i = e[i].next) {
        int v = e[i].to;  // 这条边的终点
        int w = e[i].w;   // 这条边的权值

        // 在这里进行你的操作，例如：
        // if (!visited[v]) dfs(v);
        // if (dist[v] > dist[u] + w) ...
    }
}

```

## 1. LCA (最近公共祖先) —— 倍增法

- **对应问题**：求树上任意两点距离、判断两点路径上包含什么边。
- **简单讲解**：
    1. 先跑一遍 BFS/DFS，算出每个点的深度 `depth`，以及每个点“向上跳 $2^0$ 步（即父节点）”是谁。
    2. 利用倍增公式 `fa[u][i] = fa[ fa[u][i-1] ][ i-1 ]`（跳 $2^i$ 步等于先跳 $2^{i-1}$ 再跳 $2^{i-1}$），预处

理出所有 $2^i$ 的祖先。

3. 查询时，先把两点跳到同一高度，然后一起往上跳，直到相遇。

```cpp
// ==== 在你的全局变量区添加 ====
const int MAX_LOG = 20; // 2^20 > 10000，够用了
int parent[N][MAX_LOG]; // parent[u][i] 表示 u 向上跳 2^i 步到达的祖先
int depth[N];           // 深度

// ==== 1. 预处理 (BFS版本，防爆栈) ====
void bfs_lca(int root) {
    memset(depth, 0, sizeof(depth));
    memset(parent, 0, sizeof(parent));

    queue<int> q;
    q.push(root);
    depth[root] = 1;
    parent[root][0] = root; // 根节点的父亲设为自己或0均可，看习惯

    while(!q.empty()) {
        int u = q.front(); q.pop();

        // 遍历 u 的所有邻居
        for(int i = head[u]; i != -1; i = e[i].next) {
            int v = e[i].to;
            if(depth[v] > 0) continue; // 访问过了（因为是树，访问过就是父亲）

            depth[v] = depth[u] + 1;
            parent[v][0] = u; // 2^0 步就是父节点

            // 【核心】倍增预处理：v的2^j祖先 = v的2^(j-1)祖先 的 2^(j-1)祖先
            for(int j = 1; j < MAX_LOG; j++) {
                parent[v][j] = parent[ parent[v][j-1] ][ j-1 ];
            }
            q.push(v);
        }
    }
}

// ==== 2. 查询函数 ====
int lca(int x, int y) {
    if(depth[x] < depth[y]) swap(x, y); // 保证 x 是深度较深的那个

    // 1. 把 x 跳到和 y 同一层
    for(int i = MAX_LOG - 1; i >= 0; i--) {
        if(depth[ parent[x][i] ] >= depth[y]) {
            x = parent[x][i];
        }
    }

    if(x == y) return x; // 如果跳上来发现重合了，说明 y 本来就是 x 的祖先

    // 2. x 和 y 一起往上跳，直到跳到 LCA 的下一层
    for(int i = MAX_LOG - 1; i >= 0; i--) {
```

```
        if(parent[x][i] != parent[y][i]) {
            x = parent[x][i];
            y = parent[y][i];
        }
    }

    return parent[x][0]; // 再往上跳一步就是 LCA
}

// 使用方法：
// 1. 建好树 (addEdge)
// 2. bfs_lca(根节点);
// 3. int ans = lca(u, v);
```

## 2. SPFA —— 判负环 / 差分约束

- **对应问题**：图中是否有负权回路？不等式组是否有解？带负权边的最短路。
- **简单讲解**：
  - 基于队列的 BFS。我更新了你的距离，你就入队去更新你的邻居。
  - **判负环原理**：如果一个点进出队列被更新了 $N$ 次以上，说明在绕圈圈（负环）。

```
// ==== 在你的全局变量区添加 ====
int dist[N];
bool in_queue[N]; // 是否在队列中
int cnt[N];       // 记录每个点入队/被更新的次数

// ==== 返回 true 表示发现负环，false 表示正常 ====
bool spfa(int start, int n) {
    memset(dist, 0x3f, sizeof(dist));
    memset(in_queue, 0, sizeof(in_queue));
    memset(cnt, 0, sizeof(cnt));

    queue<int> q;
    q.push(start);
    dist[start] = 0;
    in_queue[start] = true;

    while(!q.empty()) {
        int u = q.front(); q.pop();
        in_queue[u] = false;

        for(int i = head[u]; i != -1; i = e[i].next) {
            int v = e[i].to;
            int w = e[i].w;

            // 松弛操作
            if(dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                cnt[v] = cnt[u] + 1; // 记录路径长度(边数)

```

```
                // 【核心】如果一条路径经过了 >= n 个点，说明有负环
                if(cnt[v] >= n) return true;

                if(!in_queue[v]) {
                    q.push(v);
                    in_queue[v] = true;
                }
            }
        }
    }
    return false; // 无负环
}
```

## 3. Tarjan —— 强连通分量 (SCC)

- **对应问题**：缩点。把有向有环图变成有向无环图(DAG)。
- **简单讲解**：
  - DFS 遍历图。`dfn` 记录第一次访问的时间，`low` 记录能回溯到的最早时间。
  - 用栈记录访问过的点。
  - 如果 `dfn[u] == low[u]`，说明 `u` 是这个团伙（SCC）的头目，栈里 `u` 之上的点都是这个团伙的。

```
// ==== 在你的全局变量区添加 ====
int dfn[N], low[N], timer;
int stk[N], top;       // 手写栈
bool in_stack[N];      // 是否在栈中
int scc_id[N], scc_cnt; // 每个点所属的连通分量ID，连通分量总数

void tarjan(int u) {
    dfn[u] = low[u] = ++timer;
    stk[++top] = u;
    in_stack[u] = true;

    for(int i = head[u]; i != -1; i = e[i].next) {
        int v = e[i].to;

        if(!dfn[v]) { // 如果没访问过
            tarjan(v);
            low[u] = min(low[u], low[v]);
        }
        else if(in_stack[v]) { // 如果访问过且还在栈里(说明是回边，构成了环)
            low[u] = min(low[u], dfn[v]);
        }
    }

    // 【核心】发现一个强连通分量的根
    if(dfn[u] == low[u]) {
        scc_cnt++;
        int y;
        do {
```

```
            y = stk[top--]; // 弹栈
            in_stack[y] = false;
            scc_id[y] = scc_cnt; // 标记属于哪个分量
            // 这里可以做缩点后的逻辑，比如 scc_val[scc_cnt] += val[y]
        } while(u != y);
    }
}

// 使用方法：
// 循环所有点，如果 !dfn[i] 则调用 tarjan(i);
// 因为图可能不连通
// for(int i=1; i<=n; i++) if(!dfn[i]) tarjan(i);
```

## 4. 树的直径

- **对应问题**：树上最远两点的距离。
- **简单讲解**：两次 BFS/DFS。第一次随便从一点出发找最远点 P，第二次从 P 出发找最远点 Q。P 到 Q 的距离就是直径。

```
// ==== 全局变量 ====
int max_dist, end_point;

// 需要复用 dfs，所以多传一个参数: current_dist
void dfs_diameter(int u, int fa, int current_dist) {
    if (current_dist > max_dist) {
        max_dist = current_dist;
        end_point = u; // 记录最远的点
    }

    for (int i = head[u]; i != -1; i = e[i].next) {
        int v = e[i].to;
        int w = e[i].w;
        if (v == fa) continue; // 只要不走回头路即可
        dfs_diameter(v, u, current_dist + w);
    }
}

// 使用方法：
// 1. max_dist = 0; dfs_diameter(1, 0, 0); // 从1出发找最远点 P (存入 end_point)
// 2. int P = end_point;
// 3. max_dist = 0; dfs_diameter(P, 0, 0); // 从 P 出发找最远点 Q
// 4. cout << max_dist; // 直径
```

## 5. 最小费用最大流 (MCMF)

**注意**：这个算法**必须**修改你的 `Edge` 结构体，因为除了权值（即费用 `w`），还需要容量 `cap`。

- **对应问题**：二分图带权匹配、货物调配成本最低。
- **原理**：Edmonds-Karp 算法的变种。把 EK 里的 `BFS`（找最短跳数路径）改成 `SPFA`（找费用最小的路径）。

对比：

- **普通最大流 (Edmonds-Karp)**：用 BFS 找路径。BFS 只看跳数（经过几条边），它可能会傻乎乎地先选一条“容量大但死贵”的路，导致最后算出来的钱不是最少的。
- **最小费用最大流**：用 **SPFA** 找路径。SPFA 是在找**“最短路”**。
    - 在这里，边的“长度” = **费用**。
    - SPFA 找到的“最短路” = **“最便宜的路径”**。

```cpp
#include <iostream>
#include <cstring>
#include <queue>
#include <algorithm>
using namespace std;

// ==== 0. 参数设置 ====
const int N = 5005;   // 最大点数
const int M = 100005; // 最大边数 (注意：要开题目原边数的2倍，因为有反向边)
const int INF = 0x3f3f3f3f;

// ==== 1. 链式前向星 ====
struct Edge {
    int to, nxt;
    int cap;  // 容量 (Capacity)
    int cost; // 费用 (Cost)
} e[M];

int head[N], cnt = 0; // cnt 从 0 或 1 开始均可，这里用 0，方便 i^1 运算

// 初始化 (多组数据必写)
void init() {
    memset(head, -1, sizeof(head));
    cnt = 0;
}

// 加边：同时建立【正向边】和【反向边】
void add(int u, int v, int cap, int cost) {
    // 正向边：容量为 cap，费用为 cost
    e[cnt] = {v, head[u], cap, cost}; head[u] = cnt++;

    // 反向边：容量为 0 (初始不可逆流)，费用为 -cost (反悔退钱)
    e[cnt] = {u, head[v], 0, -cost};  head[v] = cnt++;
}

// ==== 2. SPFA / MCMF 核心变量 ====
int d[N];      // dist: 到源点的最小费用
int incf[N];   // incf: 增广路上各点的"剩余流量限制" (incremental flow)
int pre[N];    // pre: 记录前驱节点 (用于回溯路径)
int pree[N];   // pree: 记录前驱边的索引 (edge index)
bool vis[N];   // in_queue: 是否在队列中

// ==== 3. SPFA: 寻找费用最小的增广路 ====
bool spfa(int s, int t) {
```

```
    memset(d, 0x3f, sizeof(d)); // 费用初始化为无穷大
    memset(vis, 0, sizeof(vis));

    queue<int> q;
    q.push(s);
    vis[s] = true;
    d[s] = 0;
    incf[s] = INF; // 源点有无穷多的流量可以流出

    while (!q.empty()) {
        int u = q.front(); q.pop();
        vis[u] = false;

        for (int i = head[u]; i != -1; i = e[i].nxt) {
            int v = e[i].to;
            int c = e[i].cap;
            int w = e[i].cost;

            // 核心条件：1. 还有剩余容量 (路没堵死)  2. 费用更少 (更便宜)
            if (c > 0 && d[v] > d[u] + w) {
                d[v] = d[u] + w;           // 更新最小费用
                incf[v] = min(incf[u], c); // 流量受限于路径中最窄的管子
                pre[v] = u;                // 记录我是从 u 来的
                pree[v] = i;               // 记录我是走第 i 条边来的

                if (!vis[v]) {
                    q.push(v);
                    vis[v] = true;
                }
            }
        }
    }
    return d[t] != INF; // 如果 d[t] 还是 INF，说明没路了
}

// ==== 4. 主函数: 累加流量和费用，更新残量网络 ====
// maxf: 最大流结果，minc: 最小费用结果
void MCMF(int s, int t, int &maxf, int &minc) {
    maxf = 0; minc = 0;

    // 只要能找到更便宜的路 (spfa 返回 true)，就接着流
    while (spfa(s, t)) {
        int f = incf[t]; // 这一趟增广路能流过的最大流量

        maxf += f;
        minc += f * d[t]; // 这一趟的总花费 = 流量 * 单价

        // 【关键】倒着往回爬，更新正反向边的容量
        int x = t;
        while (x != s) {
            int i = pree[x]; // 找到通向 x 的那条边

            e[i].cap -= f;   // 正向边流量减少
```

```
            e[i ^ 1].cap += f; // 反向边流量增加 (i^1 能自动找到反向边索引)

            x = pre[x];       // 爬回前驱节点
        }
    }
}

// 使用示例
/*
int main() {
    init();
    // 读入 u, v, cap, cost ...
    // add(u, v, cap, cost);
    int max_flow, min_cost;
    MCMF(start_node, end_node, max_flow, min_cost);
    cout << max_flow << " " << min_cost << endl;
}
*/

```