最短路径问题

- Dijkstra
- Bellman-Ford / SPFA

主讲：Coel
QQ：845813930
Luogu uid：148288

Chapter 1 前置知识

前置知识

一、带权图的存储与遍历（邻接表、链式前向星）
（请自行回顾第二讲
掌握其一即可，后面代码演示都使用链式前向星。

邻接表示例

```cpp
struct node {
    int v, int w;
};

vector<node> G[maxn]; // G[u] 存放所有以 u 为起点的边

void add(int u, int v, int w) {
    G[u].push_back({v, w});
}

void search(int u) { // 遍历 u 的所有出边
    for (int i = 0; i < (int)G[u].size(); i++) {
        int v = G[u][i].v, w = G[u][i].w;
        // do something
    }
}
```

链式前向星示例

```cpp
int head[maxn], nxt[maxn], to[maxn], val[maxn], cnt;

void add(int u, int v, int w) { // 使用前要把 head 数组初始化为 -1
    nxt[cnt] = head[u];
    to[cnt] = v;
    val[cnt] = w;
    head[u] = cnt++;
}

void search(int u) {
    for (int i = head[u]; i != -1; i = nxt[i]) {
        int v = to[i]; w = val[i];
        //do something
    }
}
```

前置知识

二、简单的 STL 容器使用

本节课的两个算法分别要用到优先队列（堆）和普通队列。

两种队列的基本用法

```
// 普通队列
queue<int> Q; // 定义一个队列 Q
Q.push(x); // 将元素 x 放进队列末尾
Q.front(); // 返回队列最前面的元素
Q.pop(); // 删除队列最前面的元素
Q.empty(); // 返回一个 bool 变量，表示队列是否为空
Q.size(); // 返回队列中元素的个数

// 优先队列
priority_queue<int> Q; // 定义一个从大到小排序的优先队列（即大根堆）
priority_queue<int, greater<int>, vector<int> > Q;
// 定义一个从小到大排序的优先队列（即小根堆）
Q.top(); // 返回优先队列最前面的元素
// 除没有 Q.front() 外，其余操作与普通队列一致
```

Chapter 2 Dijkstra 算法

单源最短路径算法是求什么的?

标准回答：求最短路的。

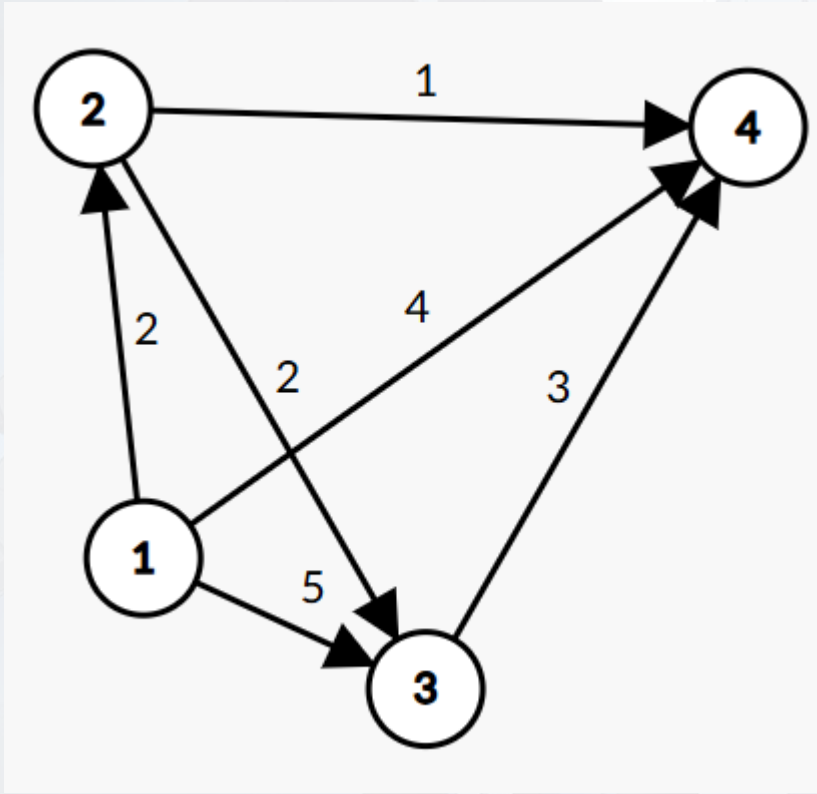
具体来说，这有两个模板：

P3371 单源最短路径（弱化版）

P4779 单源最短路径（标准版）

单源最短路径算法是求什么的?

样例长这样:



现在目的就是求从 1 出发到达 1、2、3、4 的最短路径。

Dijkstra 算法

可以处理非负权图的单源最短路径。

把所有的点划分到两个集合 S,T 之中，前者包含已经确定最短路径的点，后者包含未确定最短路径的点。

通过**松弛操作**，将所有节点从 T 移动到 S 中，直到 T 变为空集。

具有普遍最优性（Universal Optimality）。

Dijkstra 算法

用到的记号：
1.记从源点 s 到节点 u 的最短路径为 dis_u；
2.记节点 u 是否划分到集合 S 的标记为 vis_u（即：如果节点 i 的最短路已经确定，则 $vis_u = 1$，反之为 0）。

Dijkstra 算法

步骤如下：

1.初始化 $dis_s = 0$，其余节点 $dis_u = +\infty$；所有节点都在 T 集合中（即 $vis_u = 0$）。

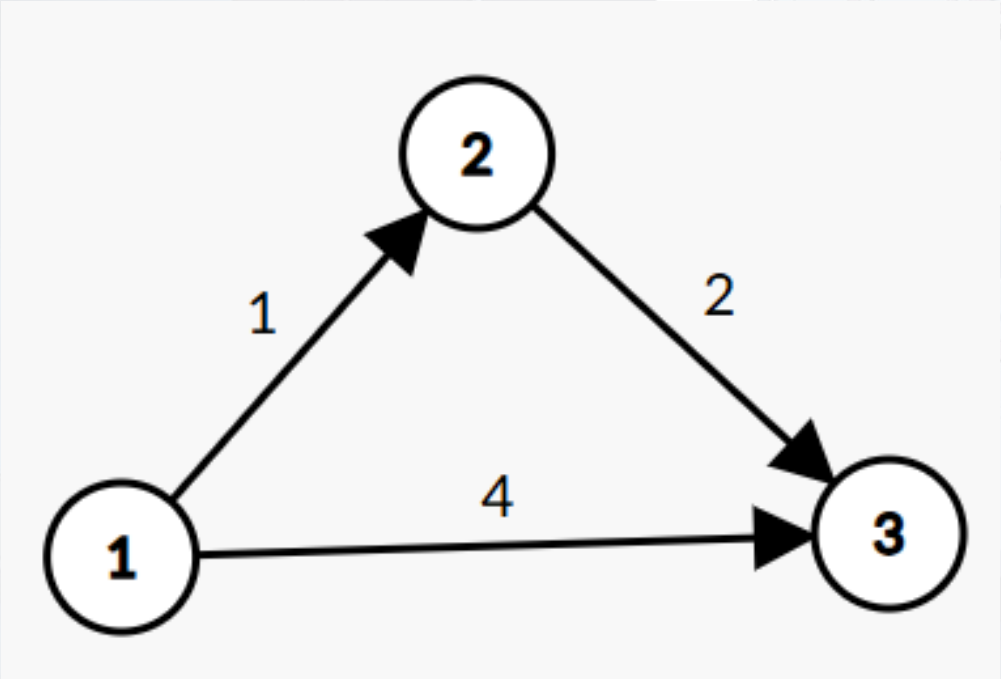
2.从 T 集合中选取一个 dis_u 最小的节点 u，并将其移入 S 集合（即标记 $vis_u = 1$）；

3.遍历节点 u 的所有出边，并进行**松弛操作**。

4.重复步骤 2、3，直到所有节点都被划分到集合 S 中。

松弛操作是什么？

一句话解释：如果$dis_u > dis_v + w$，则更新 $dis_u = dis_v + w$。



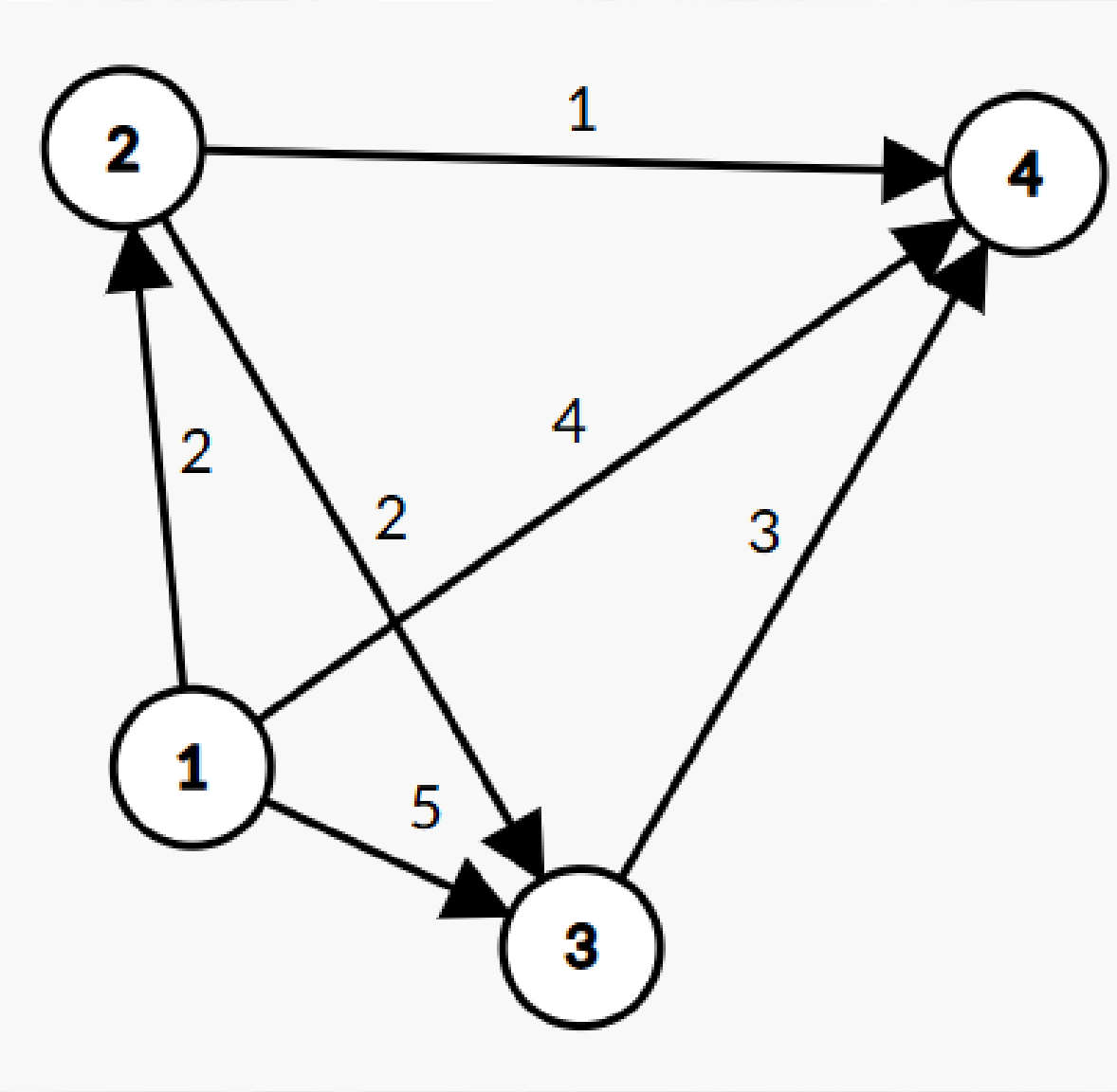
对于 dis_3：

原来值是 4
现在发现 $4 > 3$
操作后变 3

Dijkstra 算法

具体演示一下。
一开始的状况：

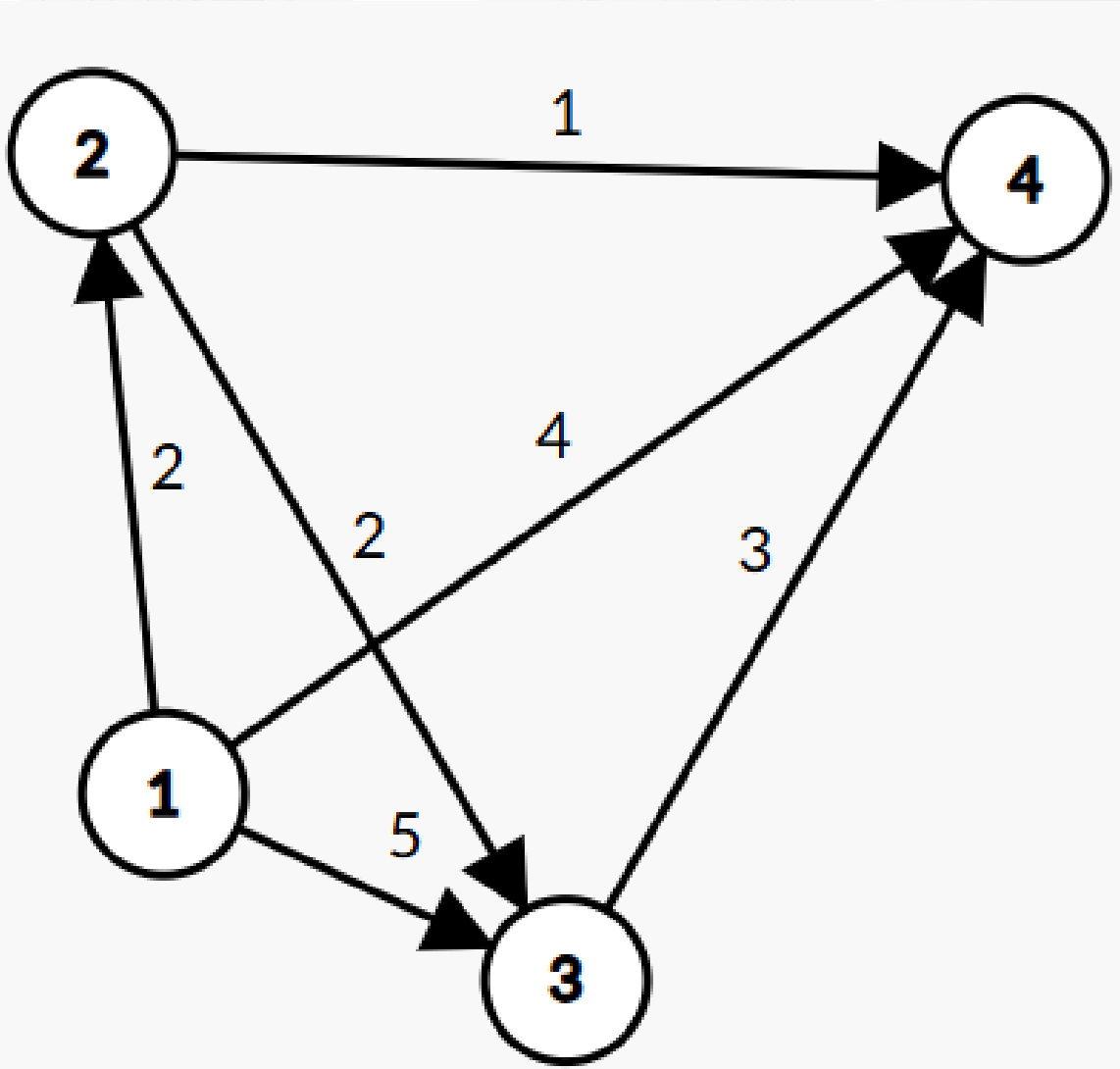
编号	dis	vis
1	0	False
2	Inf	False
3	Inf	False
4	inf	False



Dijkstra 算法

现在 dis_1最小，对它遍历出边并松弛。

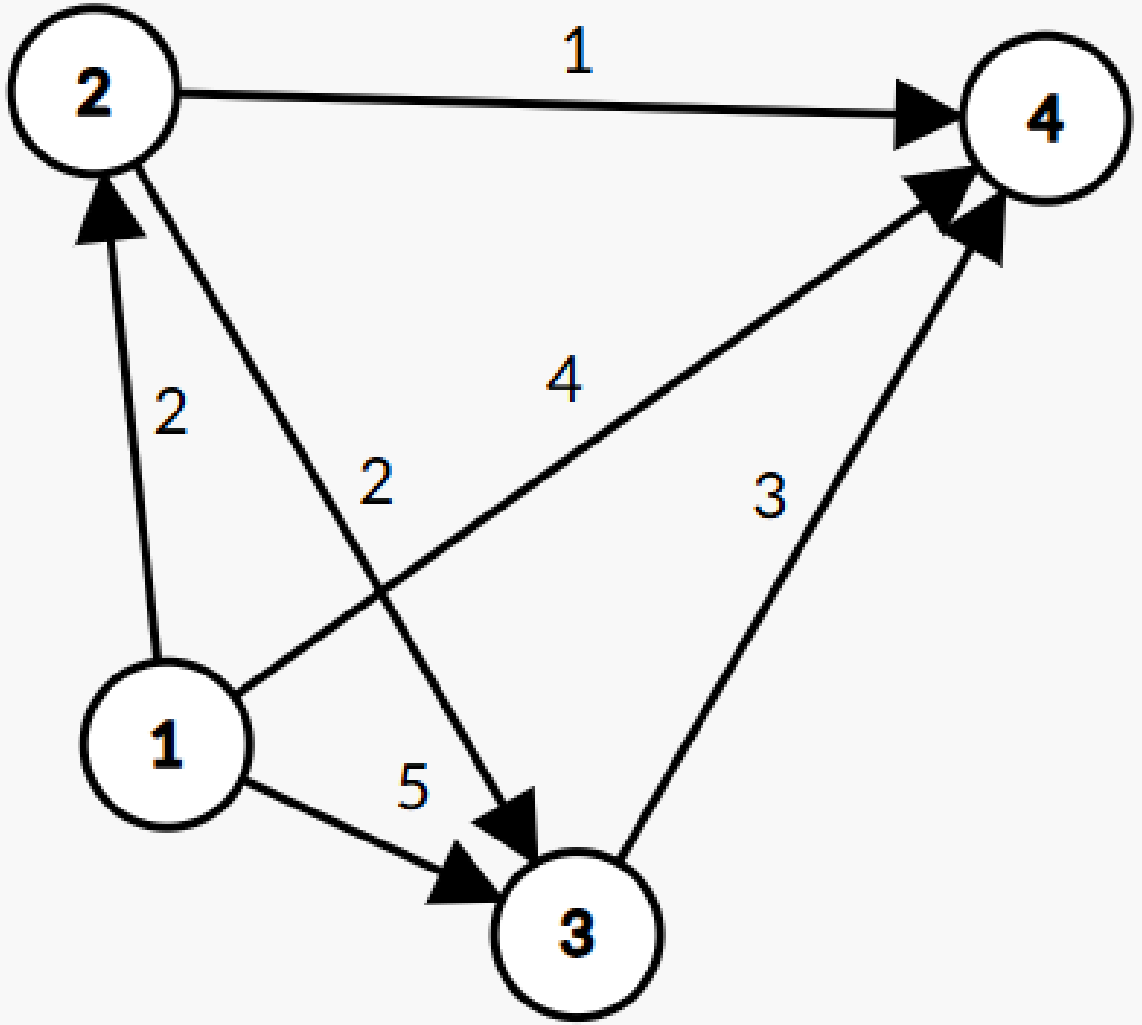
编号	dis	vis
1	0	True
2	Inf->2	False
3	Inf->5	False
4	Inf->4	False



Dijkstra 算法

现在 dis_2最小，对它遍历出边并松弛。

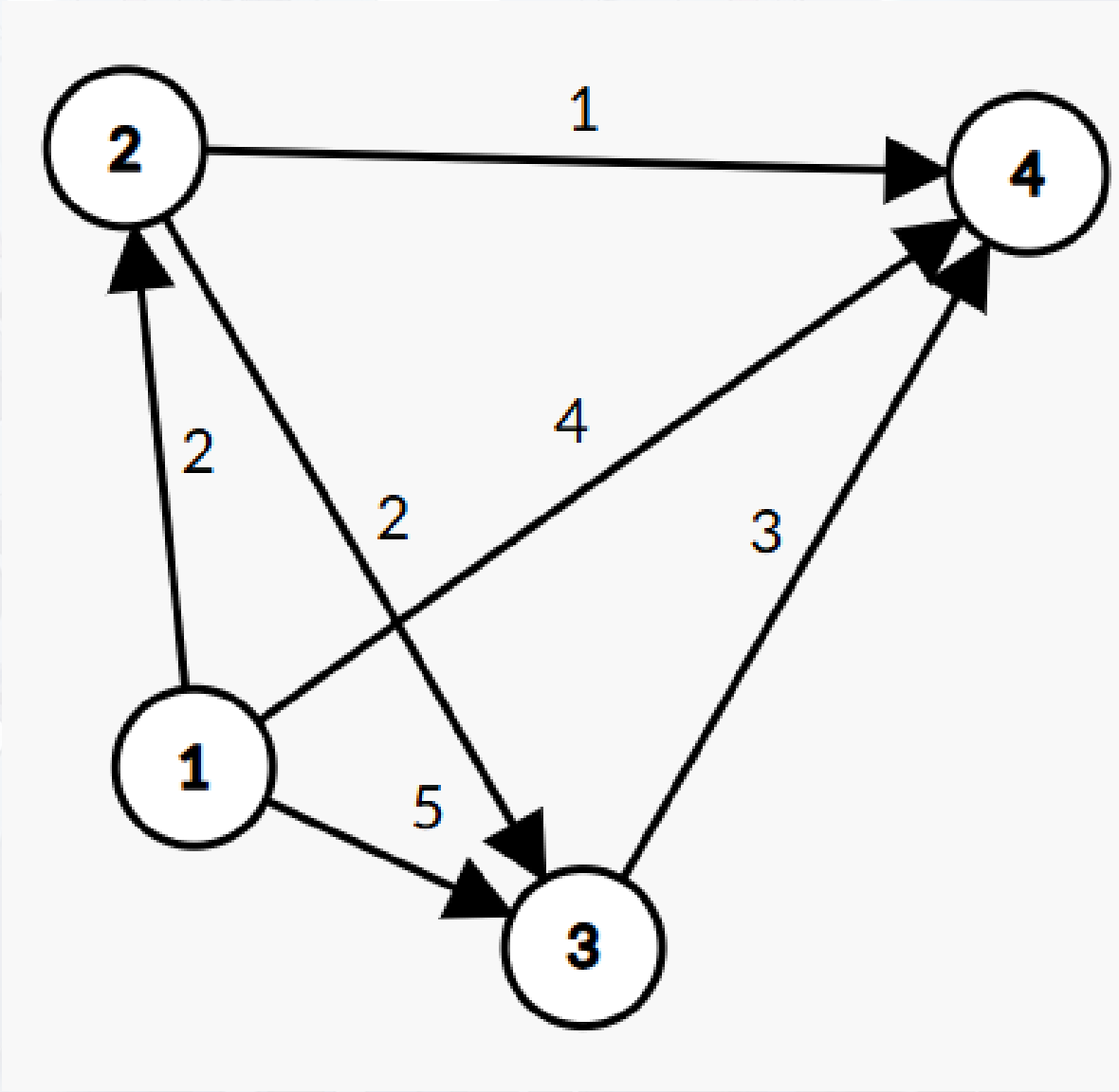
编号	dis	vis
1	0	True
2	2	True
3	5->4	False
4	4->3	False



Dijkstra 算法

现在 dis_4最小，但是没有出边。

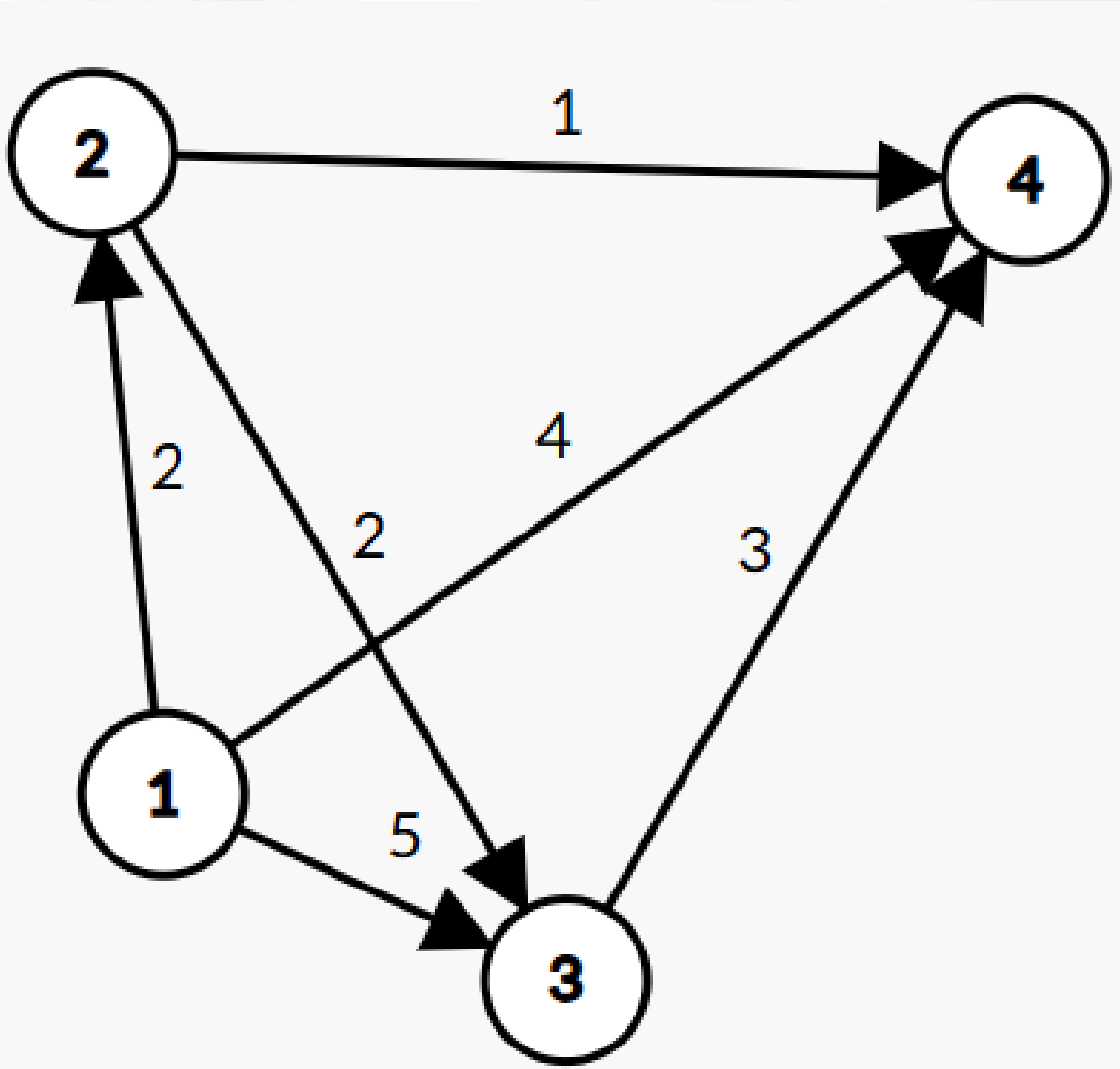
编号	dis	vis
1	0	True
2	2	True
3	4	False
4	3	True



Dijkstra 算法

现在 dis_3最小，遍历出边（没有需要松弛的点）

编号	dis	vis
1	0	True
2	2	True
3	4	True
4	3	True



代码示例 时间复杂度：$O(n^2)$

```
void Dijkstra(int s) {
    memset(dis, 0x3f, sizeof(dis));
    dis[s] = 0; // 第一步：初始化
    for (int i = 1; i <= n; i++) {
        int u = 0, minDis = 0x3f3f3f3f;
        for (int j = 1; j <= n; j++) {
            if (vis[j] == 0 && dis[j] < minDis) {
                minDis = dis[j];
                u = j;
            }
        }
        vis[u] = 1; // 第二步：寻找最小节点，并标记
        for (int j = head[u]; j != -1; j = nxt[j]) {
            int v = to[j], w = val[j];
            if (dis[v] > dis[u] + w) {
                dis[v] = dis[u] + w;
            }
        } //第三步：遍历，松弛操作
    }
}
```

优化

注意到时间瓶颈在于找到 dis_u 最小的节点 u（原本是直接暴力比较的）。

因为每次从 T 集合中拿出来的点一定是松驰过的点（为什么？），所以可以维护一个数据结构，每松弛一条边就把终点放进去（前提是这个点在 T 集合中），然后不断地从这个数据结构中拿出 dis_u 最小的节点，进行遍历和松弛操作。

这个数据结构需要实现“插入元素”和“查询并删除最值元素”两个工作，自然就是堆了。

代码示例 时间复杂度：$O(m \log n)$

```cpp
typedef pair<int, int> pii; // pair 的第一维存放 dis 值，第二维存放节点序号
void Dijkstra(int s) {
    priority_queue<pii, vector<pii>, greater<pii> > Q;
    memset(dis, 0x3f, sizeof(dis));
    dis[s] = 0; //第一步：初始化
    Q.push({0, s});
    while (!Q.empty()) {
        int u = Q.top().second;
        Q.pop();
        if (vis[u]) continue;
        vis[u] = 1; // 第二步：寻找最小节点，并标记
        for (int i = head[u]; i != -1; i = nxt[i]) {
            int v = to[i], w = val[i];
            if (dis[v] > dis[u] + w) {
                dis[v] = dis[u] + w;
                if (!vis[v]) Q.push({dis[v], v});
            }
        } //第三步：遍历，松弛操作
    }
}
```

Chapter 3 SPFA / Bellman-Ford

SPFA / Bellman-Ford

可以处理负权图的单源最短路径。

SPFA 通过对 Bellman-Ford 优化得到，在国外也被叫做 Bellman-Ford-Moore 算法。

Bellman-Ford 的流程比较无脑，而 SPFA 的优化有点玄学。

SPFA / Bellman-Ford

Bellman-Ford 算法的流程为：

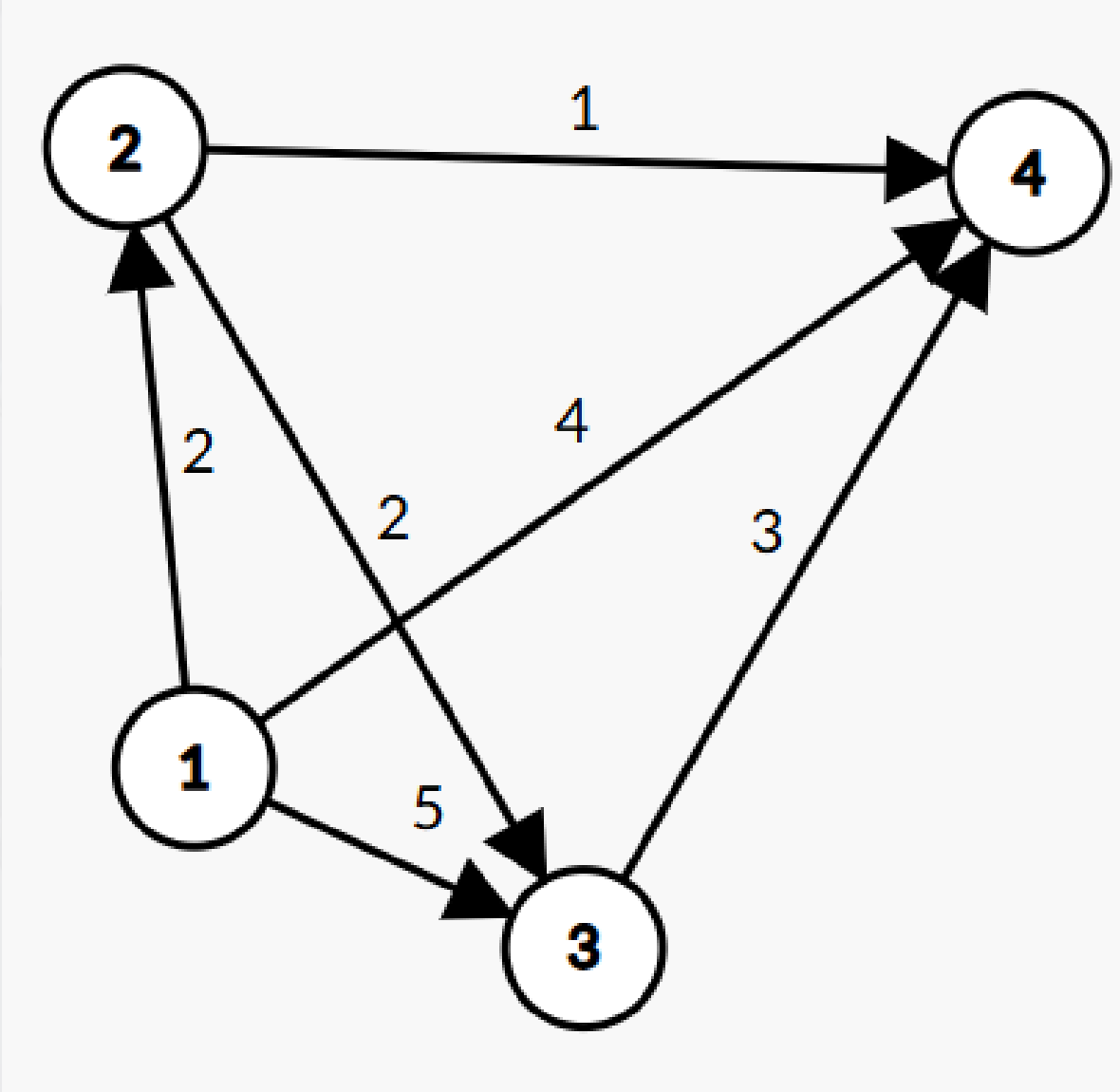
1. 初始化 $dis_s = 0$，其余节点 $dis_u = +\infty$；
2. 遍历所有边，对所有可以松弛的边进行松弛操作；
3. 重复第二步，直到无法进行任何松弛操作（最多重复 n 次）。

不管那么多，松弛就完了！

Bellman-Ford 算法

具体演示一下。
一开始的状况：

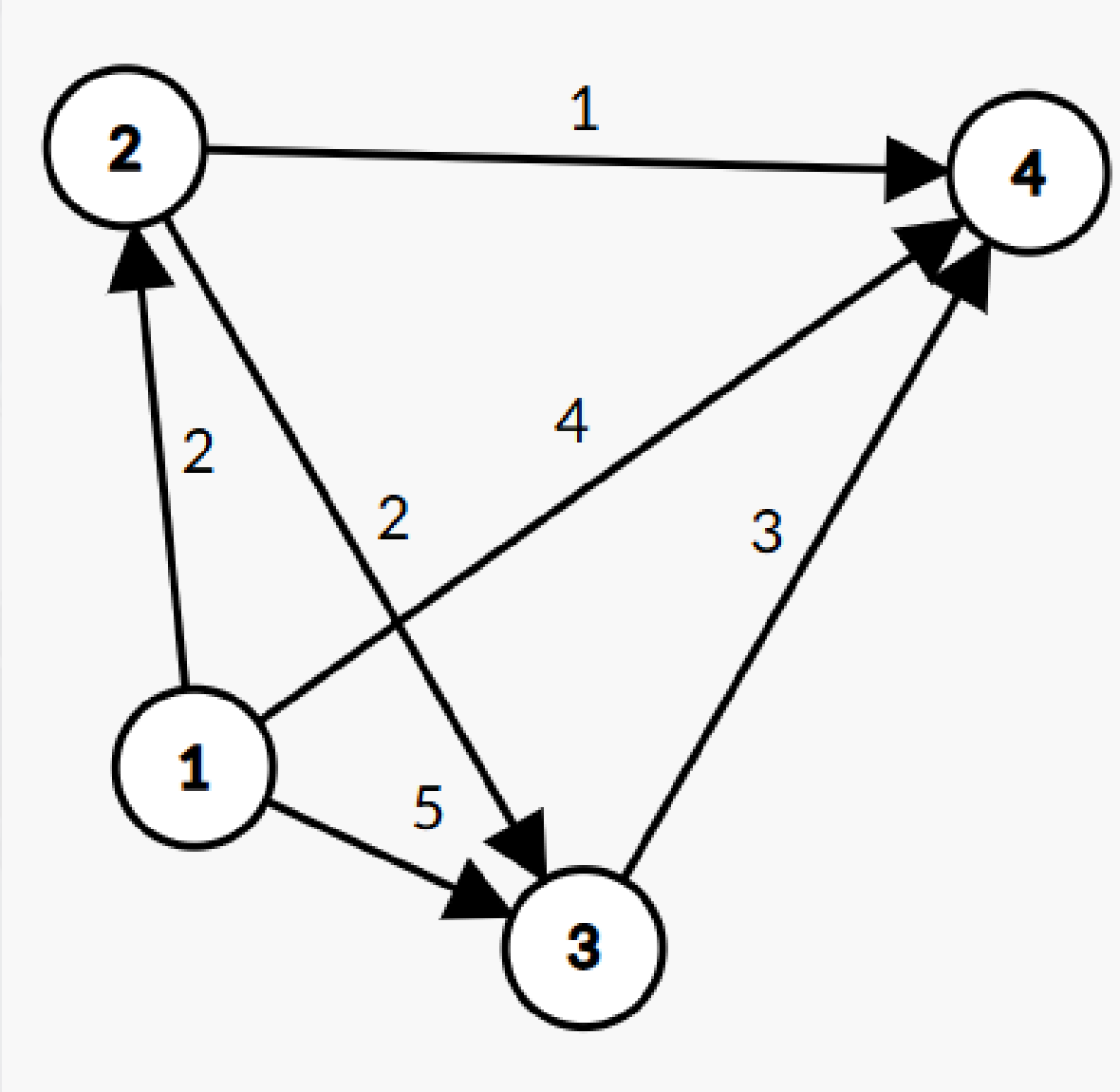
编号	dis
1	0
2	Inf
3	Inf
4	inf



Bellman-Ford 算法

第一轮:

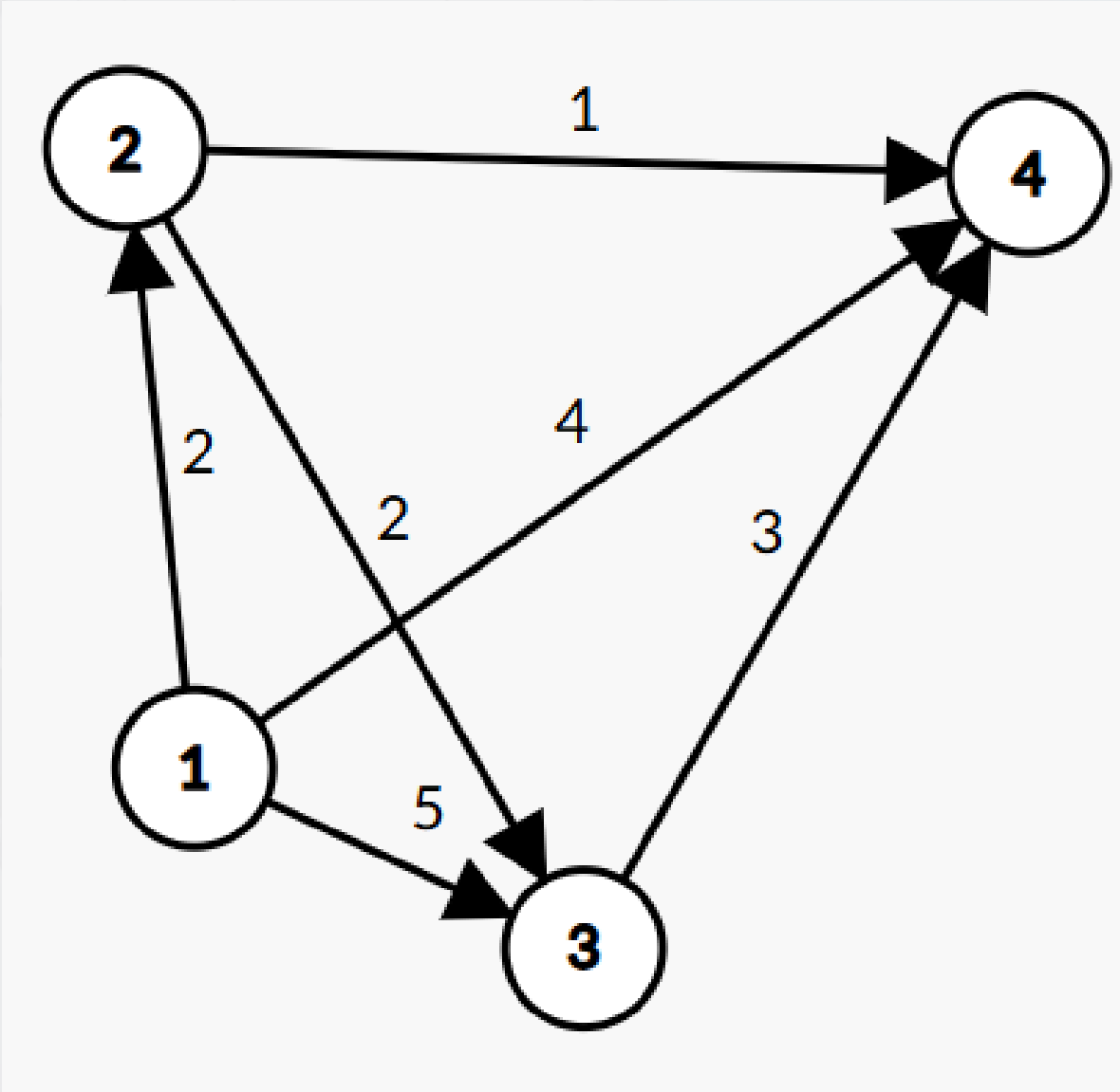
编号	dis
1	0
2	2
3	4
4	3



Bellman-Ford 算法

第二轮：由于不存在可以松弛的边，所以结束。

编号	dis
1	0
2	2
3	4
4	3

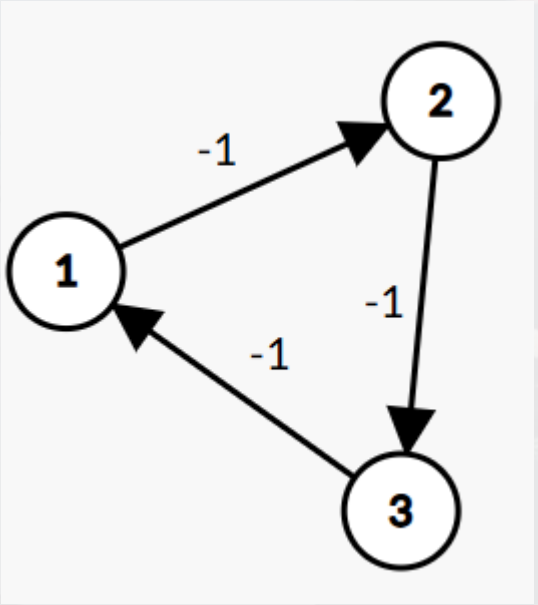


Bellman-Ford 算法

当然有时候负权图的最短路可能不存在，比如这个：

因为环上权值和是负的，所以每绕环一圈权值都会减少，这被叫做负环。

怎么判断有没有负环呢？由于 Bellman-Ford 限定第二步最多重复 n 次，所以看看 n 次之后有没有再松弛就行。



代码示例 时间复杂度：$O(mn)$

```
bool Bellman_Ford(int s) { // 返回值表示图中是否存在可以从点 s 到达的负环
    memset(dis, 0x3f, sizeof(dis));
    dis[s] = 0; // 第一步：初始化
    bool flag;
    for (int i = 1; i <= n; i++) { // 第二步：遍历所有边并进行松弛操作
        flag = false; // 标记是否进行松弛操作
        for (int u = 1; u <= n; u++) {
            for (int j = head[u]; j != -1; j = nxt[j]) {
                int v = to[j], w = val[j];
                if (dis[u] == inf) continue; // 如果 dis[u] 是无穷大，则显然无法进行松弛操作
                if (dis[v] > dis[u] + w) {
                    dis[v] = dis[u] + w;
                    flag = true;
                }
            }
        }
        if (flag == false) break;
    }
    return flag; // 如果仍然进行了松弛操作，则说明负环存在
}
```

优化

注意到只有上一次松驰过的节点才可能影响下一次节点能否进行松弛操作，所以只需要使用一个队列存储“可能需要松弛操作的节点”，这使得需要遍历的边大大减少。

至于判断负环，只需记录最短路的边数是否超过 $n-1$ 即可。虽然这个过程与 Bellman-Ford 算法不太相同，但背后的原理是一致的。

代码示例 时间复杂度（？？）

```cpp
bool SPFA(int s) {
    queue<int> Q;
    memset(dis, 0x3f, sizeof(dis));
    Q.push(s);
    dis[s] = 0; // 第一步：初始化
    vis[s] = true; // vis[u] 表示点 u 是否被存储在队列之中
    while (!Q.empty()) { // 第二步：遍历所有需要松弛的边
        int u = Q.front();
        Q.pop(), vis[u] = false;
        for (int i = head[u]; i != -1; i = nxt[i]) {
            int v = to[i], w = val[i];
            if (dis[v] > dis[u] + w) {
                dis[v] = dis[u] + w;
                tot[v] = tot[u] + 1; // tot[u] 记录到达 u 点的最短路边数
                if (tot[v] > n - 1) return false; // 如果边数大于 n - 1，则存在负环
                if (!vis[v]) Q.push(v),  vis[v] = true;
            }
        }
    }
    return true;
}
```

时间复杂度？？

设第(5)句 while 循环的次数为 m，即为顶点入队的次数，若平均每一个点入队一次，则 $m = n$；若平均每个点入队两次，则 $m = 2n$。算法编程后实际运行试算情况表明，m 一般没有超过 $2n$。事实上，虽然顶点入队次数 m 是一个事先不易分析出的数，但它确是一个随图的不同而略有不同的常数，所谓常数，即与 e 无关，也与 n 无关，仅与边的权值分布有关，一旦图确定，各边的权值确定，源点确定，m 也就是一个确定的常数，所以，SPFA 算法的时间复杂性为

$$T = O(m \cdot \frac{e}{n}) = O(\frac{m}{n} \cdot e)$$

令 $$K = \frac{m}{n}$$

则 $$T = O(k \cdot e)$$

因为 K 是一个常数，所以 SPFA 算法的时间复杂性为 $O(e)$。(证毕)。

对……对吗？

时间复杂度？？

要是 SPFA 那么厉害，要 Dijkstra 干什么。

SPFA 算法的时间复杂度为 $O(kn)$，通常来说 k 是一个很小的常数，但是在特殊结构的图中 k 的值可以近似于 m，与普通的 Bellman-Ford 算法相差不大，远远落后于 Dijkstra 算法。因此，如果给定的图是非负权图，尽量不要使用 SPFA 算法。

几道习题

P1339 [USACO09OCT] Heat Wave G
P1629 邮递员送信
P5960 差分约束
[USACO06NOV] Roadblocks G