数据结构与算法
（七）
2.图的基础算法--最短路

杜育根

Ygdu@sei.ecnu.edu.cn

7.2. 最短路

- 这节课重点介绍带权图中的最短路问题。
- 从一个起点出发的最短路问题，被称为单源最短路，可以用 SPFA 算法、Dijkstra 算法解决；
- 从多个起点出发则被称为多源最短路问题，可以用 Floyd 算法解决。
- 在单源最短路部分，着重介绍了它的一个经典应用——差分约束系统。

7.2.1 Dijkstra 最短路算法

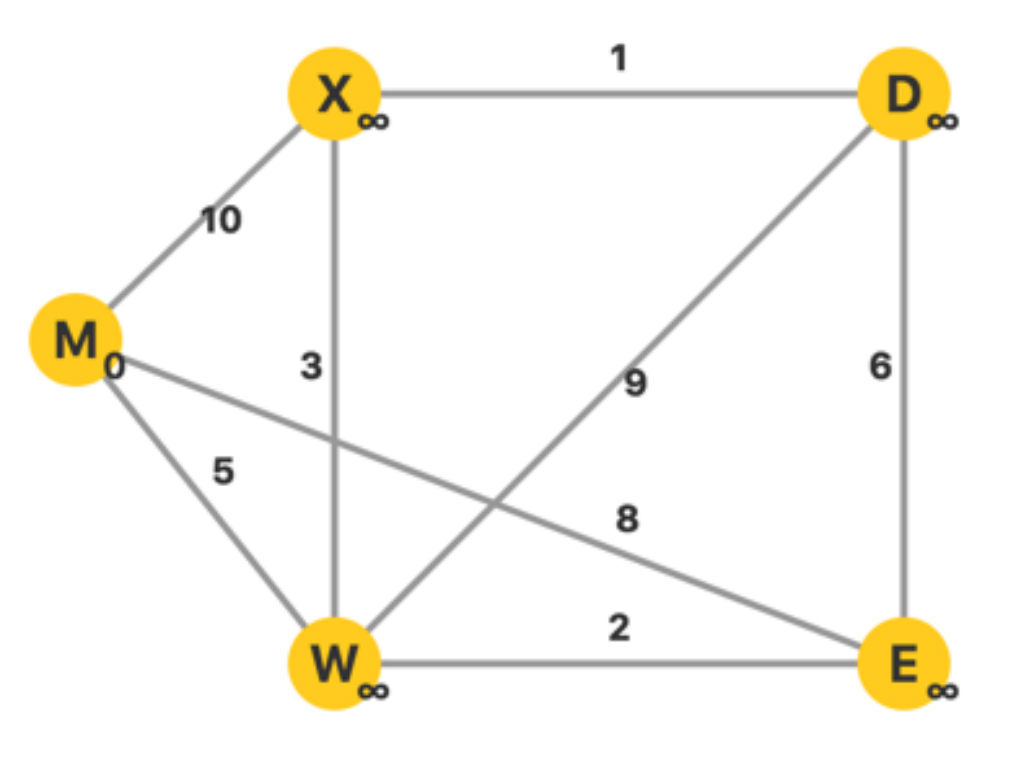
- 单源最短路问题
- 在带权图 G=(V,E) 中，每条边都有一个权值 wi，即边的长度。路径的长度为路径上所有边权之和。单源最短路问题是指：求源点 s 到图中其余各顶点的最短路径。
- 概述
- 解决单源最短路径问题常用 Dijkstra 算法，用于计算一个顶点到其他所有顶点的最短路径。Dijkstra 算法的主要特点是以起点为中心，逐层向外扩展，每次都会取一个最近点继续扩展，直到取完所有点为止。
- 注意：Dijkstra 算法要求图中不能出现负权边。

算法流程

- 我们定义带权图 G 所有顶点的集合为 V，接着我们再定义已确定从源点出发的最短路径的顶点集合为U，初始集合U为空，记从源点s出发到每个顶点v的距离为 $dist_v$，初始$dist_s$=0。接着执行以下操作：
- 1. 从V−U 中找出一个距离源点最近的顶点 v，将v加入集合U，并用$dist_v$和顶点v连出的边来更新和v相邻的、不在集合U中的顶点的dist；
- 2. 重复第一步操作，直到V=U 或找不出一个从s出发有路径到达的顶点，算法结束。
- 如果最后V≠U，说明有顶点无法从源点到达；否则每个$dist_i$表示从s出发到顶点 i 的最短距离。

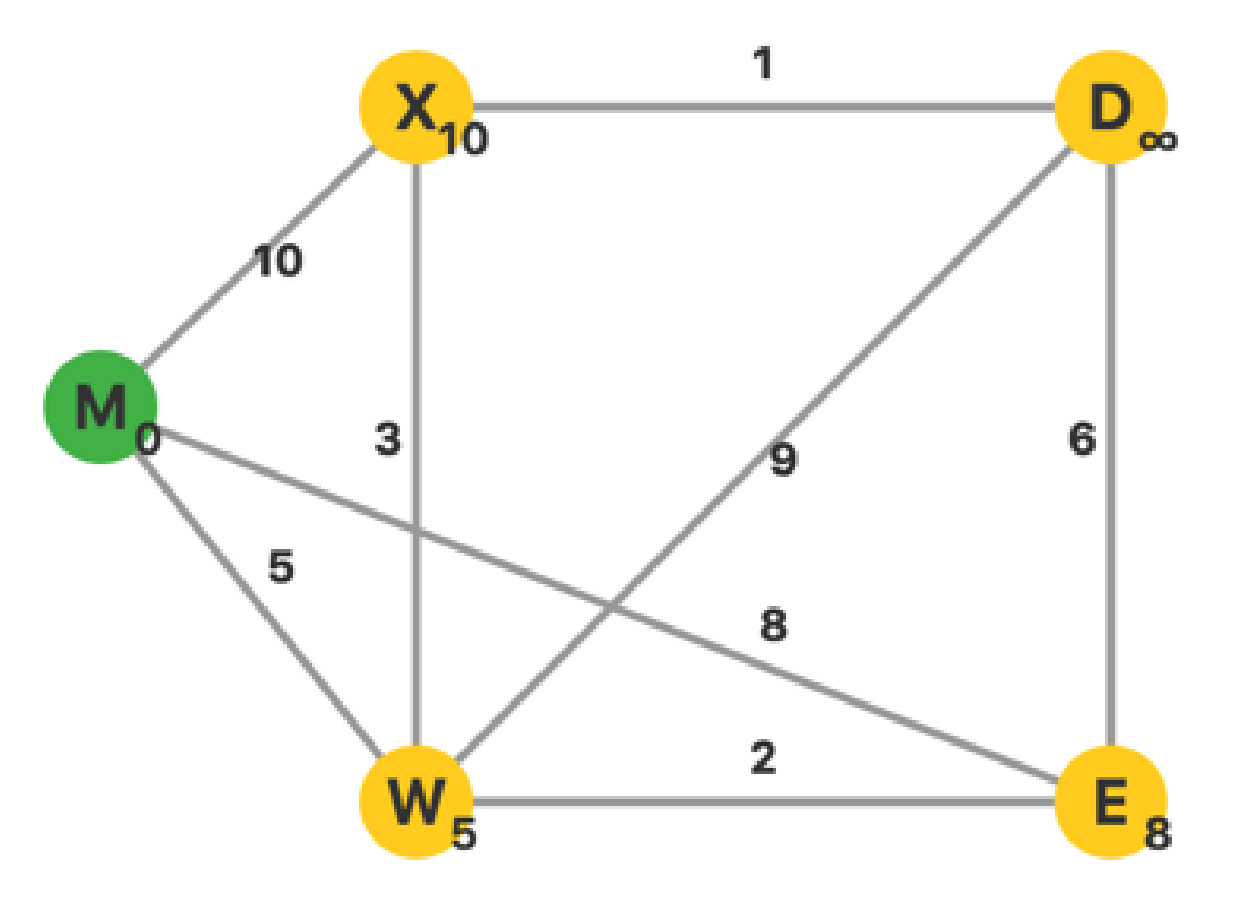
Dijkstra 算法演示

- 接下来，我们用一个例子来说明这个算法。

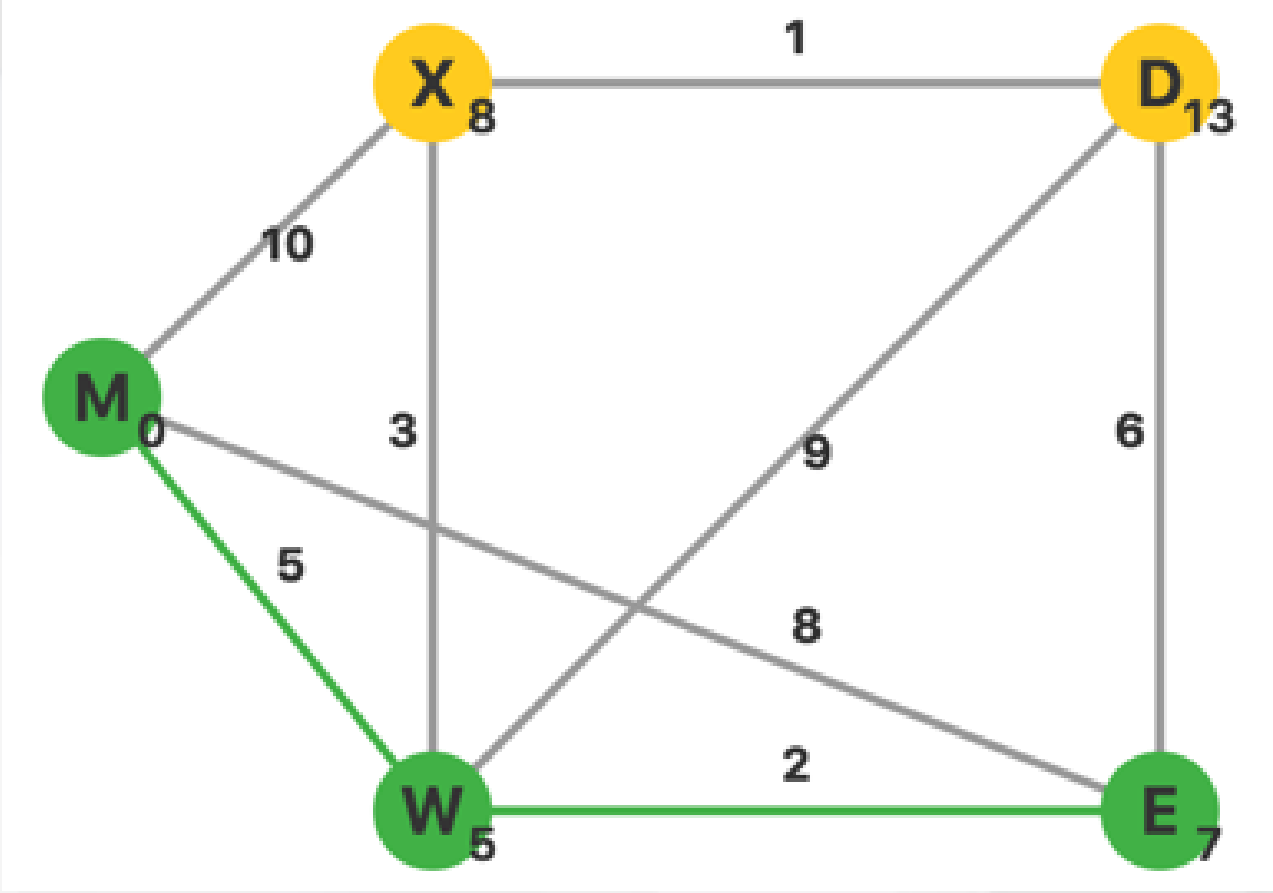


- 初始每个顶点的dist设置为无穷大 inf，源点 M的 distM 设置为0。当前 U=∅，V−U 中 dist最小的顶点是M。从顶点M 出发，更新相邻点的dist。

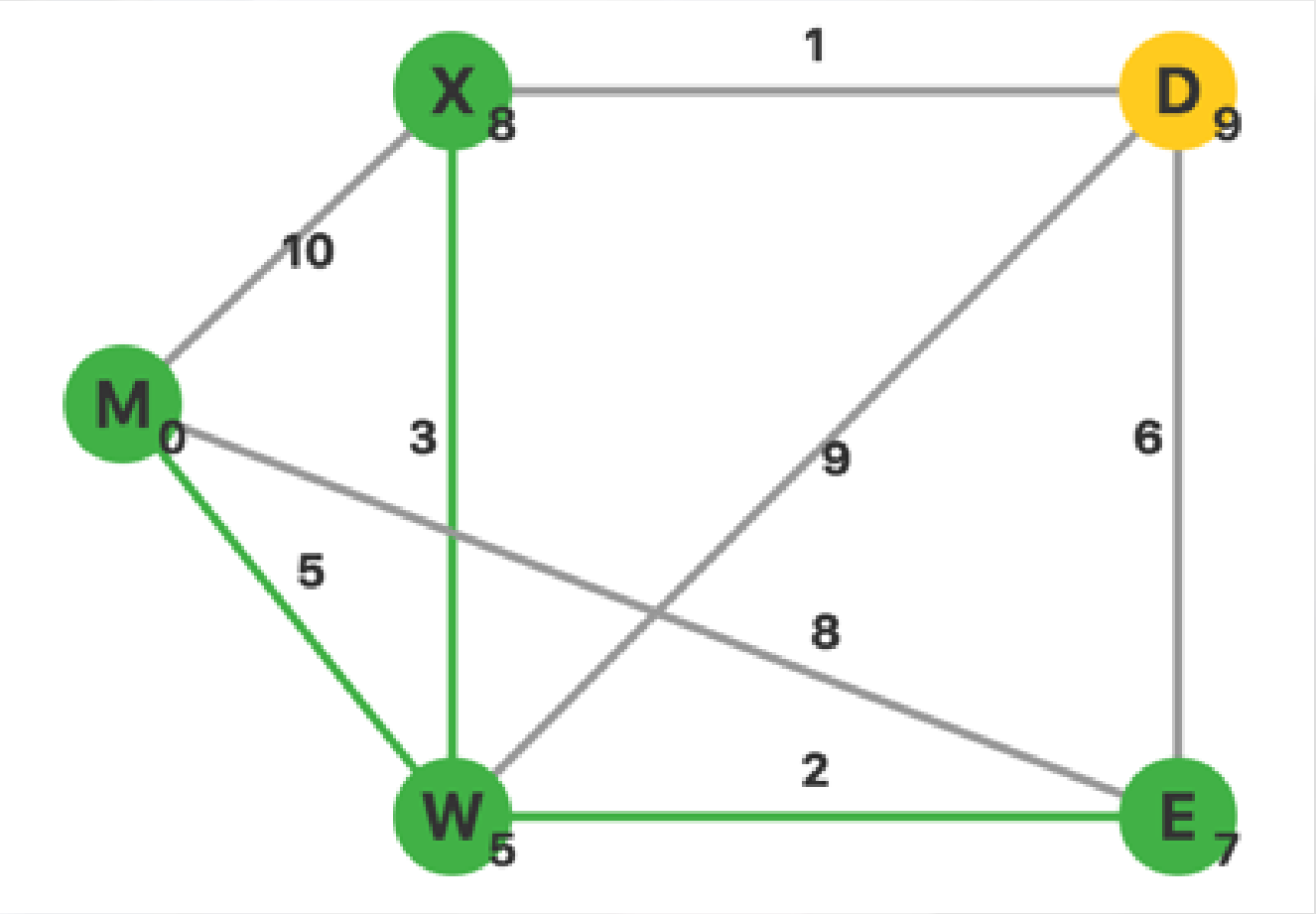
- 更新完毕，此时 U={M}，V−U 中 dist最小的顶点是 W。从 W出发，更新相邻点的 dist。



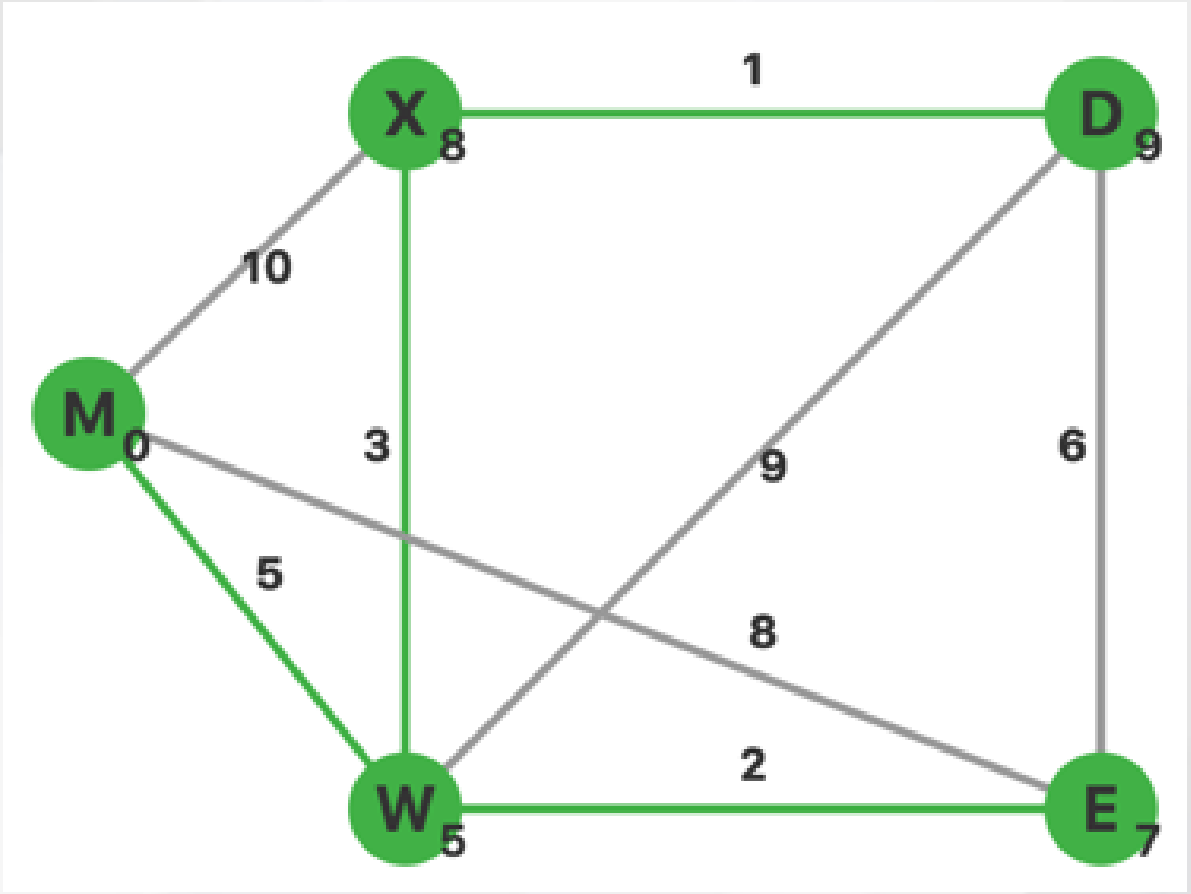
- 更新完毕，此时 U={M,W}，V−U 中 dist 最小的顶点是E。从E 出发，更新相邻顶点的dist。



- 更新完毕，此时 U={M,W,E}，V−U中 dist 最小的顶点是X。从X 出发，更新相邻顶点的 dist。



- 更新完毕，此时 U={M,W,E,X}，V−U 中 dist 最小的顶点是D。从D出发，没有其他不在集合U中的顶点。



- 此时 U=V，算法结束，单源最短路计算完毕。

参考代码

```
const int MAX_N = 10000;
const int MAX_M = 100000;
const int INF = 0x3f3f3f3f;
struct edge {
    int v, w, next;
} e[MAX_M];
int p[MAX_N], eid, n;
void mapinit() {
    memset(p, -1, sizeof(p));
    eid = 0;
}
void insert(int u, int v, int w) {  // 插入带权有向边
    e[eid].v = v;
    e[eid].w = w;
    e[eid].next = p[u];
    p[u] = eid++;
}
void insert2(int u, int v, int w) {  // 插入带权双向边
    insert(u, v, w);
    insert(v, u, w);
}
int dist[MAX_N];  // 存储单源最短路的结果
bool vst[MAX_N];  // 标记每个顶点是否在集合 U 中
bool dijkstra(int s) {
    memset(vst, 0, sizeof(vst));
    memset(dist, 0x3f, sizeof(dist));//dist初值设置为0x3f3f3f3f
    dist[s] = 0;
    for (int i = 0; i < n; ++i) {
        int v, min_w = INF;  // 记录 dist 最小的顶点编号和 dist 值
        for (int j = 0; j < n; ++j) {
            if (!vst[j] && dist[j] < min_w) {
                min_w = dist[j];
                v = j;
            }
        }
        if (min_w == INF) {  // 没有可用的顶点，算法结束，说明有
                             //顶点无法从源点到达
            return false;
        }
        vst[v] = true;  // 将顶点 v 加入集合 U 中
        for (int j = p[v]; j != -1; j = e[j].next) {
            // 如果和 v 相邻的顶点 x 满足 dist[v] + w(v, x) < dist[x]
           //则更新 dist[x]，这一般被称作“松弛”操作
            int x = e[j].v;
            if (!vst[x] && dist[v] + e[j].w < dist[x]) {
                dist[x] = dist[v] + e[j].w;
            }
        }
    }
    return true;  // 源点可以到达所有顶点，算法正常结束
}
```

堆优化

- 如果每次暴力枚举选取距离最小的元素，则总的时间复杂度是 $O(V^2)$。
- 如果考虑用堆优化，维护一个小根堆，取出距离最小的顶点，再进行扩展，时间复杂度 $O((V+E)\log V)$，对于稀疏图的优化效果非常好。如果用斐波那契堆的话可以将时间复杂度优化到 $O(V\log V+E)$，不过通常我们只用普通的小根堆就能解决单源最短路的题目了。
- 有很多同学也许见过用priority_queue实现的版本，一定要注意，这样写的时间复杂度是$O(V\log V+E\log E)$，会比正确的堆优化效率更低。大部分题目不会被卡，不过还是推荐用下面给出的这种写法，可以避免在个别题目中被“卡常数”。

小根堆优化的 Dijkstra 示例代码如下：

```
const int MAX_N = 10000;
const int MAX_M = 100000;
const int inf = 0x3f3f3f3f;
struct edge {
    int v, w, next;
} e[MAX_M];
int p[MAX_N], eid, n;
void mapinit() {
    memset(p, -1, sizeof(p));
    eid = 0;
}
void insert(int u, int v, int w) {  // 插入带权有向边
    e[eid].v = v;
    e[eid].w = w;
    e[eid].next = p[u];
    p[u] = eid++;
}
void insert2(int u, int v, int w) {  // 插入带权双向边
    insert(u, v, w);
    insert(v, u, w);
}
typedef pair<int, int> PII;
set<PII, less<PII> > min_heap;  // 用 set 来伪实现一个小根堆，并具有映射二叉堆的功能。堆中 pair<int, int> 的 second 表示顶点下标，first 表示该顶点的 dist 值
int dist[MAX_N];  // 存储单源最短路的结果
bool vst[MAX_N];  // 标记每个顶点是否在集合 U 中
```

```
bool dijkstra(int s) {    // 初始化 dist、小根堆和集合 U
    memset(vst, 0, sizeof(vst));
    memset(dist, 0x3f, sizeof(dist))
    min_heap.insert(make_pair(0, s));
    dist[s] = 0;
    for (int i = 0; i < n; ++i) {
        if (min_heap.size() == 0) {  // 如果小根堆中没有可用顶点，
                                     //说明有顶点无法从源点到达，算法结束
            return false;
        }
        // 获取堆顶元素，并将堆顶元素从堆中删除
        auto iter = min_heap.begin();
        int v = iter->second;
        min_heap.erase(*iter);
        vst[v] = true;
        // 进行和普通 dijkstra 算法类似的松弛操作
        for (int j = p[v]; j != -1; j = e[j].next) {
            int x = e[j].v;
            if (!vst[x] && dist[v] + e[j].w < dist[x]) {
     // 先将对应的 pair 从堆中删除，再将更新后的 pair 插入堆
                min_heap.erase(make_pair(dist[x], x));
                dist[x] = dist[v] + e[j].w;
                min_heap.insert(make_pair(dist[x], x));
            }
        }
    }
    return true;  // 存储单源最短路的结果
}
```

例题：特殊的生成树

- 给定一张无向图，其中边权都是正数，你需要求出总代价最小的生成树，生成树上每条边（u,v)的代价为 w(u,v)*count(v)，其中 w(u,v)为边（u,v)的权值，count(v) 是v 所在子树的结点数总和。
- 解法：虽然看起来是一道生成树的题，实际上却是一道单源最短路的题目。我们把整棵树的代价加起来，实际上就等于从根结点出发到每个顶点的最短路之和。因此，从每个顶点出发求一遍单源最短路，取其中最短路总和最小的一个顶点作为根，其到所有顶点的单源最短路的总和就是最终答案。

例题：次短路

- 对于一个带权图，求两个顶点之间的次短路。次短路表示除最短路以外长度最小的路径。
- 次短路问题是一个非常经典的问题，有很多种常见解法。一个简单直接的解法是，枚举两个顶点之间最短路上的每条边，每次在去掉这条边的剩下的图中计算最短路，取其中最小的一个答案就是最终次短路的答案。这个方法可以求解不重复经过同一顶点的次短路。
- 还有一种解法是，在进行 dijkstra 的过程中记录两个数组：dist0 和 dist1，分别表示最短路和次短路的答案。每次更新时需要依次判断是否可以更新次短路和最短路的值。由于需要计算次短路，所以调整后的 dijkstra 算法需要至少循环 $2n-1$ 次才可以获得最终答案。这个方法可以求解可重复经过同一顶点的次短路。

习题：骑车比赛

- 小明准备去参加骑车比赛，比赛在 n 个城市间进行，编号从1 到n。选手们都从城市1 出发，终点在城市n。
- 已知城市间有m 条道路，每条道路连接两个城市，注意道路是双向的。现在小明知道了他经过每条道路需要花费的时间，他想请你帮他计算一下，他这次比赛最少需要花多少时间完成。
- 输入格式:第一行输入两个整数 n,m（1≤n≤1,000,1≤m≤5,000），分别代表城市个数和道路总数。接下来输入m 行，每行输入三个数字 a,b,c（1≤a,b≤n,1≤c≤200），分别代表道路的起点和道路的终点，以及小明骑车通过这条道路需要花费的时间。保证输入的图是连通的。
- 输出格式:输出一行，输出一个整数，输出小明完成比赛需要的最少时间。
- 样例输入
- 5 6
- 1 2 2
- 2 3 3
- 2 5 5
- 3 4 2
- 3 5 1
- 4 5 1
- 样例输出
- 6

习题：圣诞树

- 圣诞节快到了，小明准备做一棵大圣诞树。
- 这棵树被表示成一组被编号的结点和一些边的集合，树的结点从 1 到n 编号，树的根永远是1。每个结点都有一个自身特有的数值，称为它的权重，各个结点的权重可能不同。对于一棵做完的树来说，每条边都有一个价值 ve，若设这条边 e 连接结点 i 和结点j，且i 为j 的父结点（根是最老的祖先），则该边的价值 $v_e=s_j \times w_e$，s_j 表示结点j 的所有子孙及它自己的权重之和，w_e 表示边 e 的权值。
- 现在小明想造一棵树，他有 m 条边可以选择，使得树上所有边的总价值最小，并且所有的点都在树上，因为小明喜欢大树。
- 输入格式: 第一行输入两个整数 n 和 m（$0 \le n,m \le 50,000$），表示结点总数和可供选择的边数。接下来输入一行，输入 n 个整数，依次表示每个结点的权重。接下来输入 m 行，每行输入 3 个正整数 a,b,c（$1 \le a,b, \le n, 1 \le c \le 10,000$），表示结点 a 和结点 b 之间有一条权值为 c 的边可供造树选择。
- 输出格式: 输出一行，如果构造不出这样的树，请输出No Answer，否则输出一个整数，表示造树的最小价值。
- 样例输入
- 4 4
- 10 20 30 40
- 1 2 3
- 2 3 2
- 1 3 5
- 2 4 1
- 样例输出
- 370

习题：迷阵突围

- 小明陷入了坐标系上的一个迷阵，迷阵上有 n个点，编号从 1 到 n。小明在编号为 1 的位置，他想到编号为 n 的位置上。小明当然想尽快到达目的地，但是他觉得最短的路径可能有风险，所以他会选择第二短的路径。现在小明知道了 n 个点的坐标，以及哪些点之间是相连的，他想知道第二短的路径长度是多少。
- 注意，每条路径上不能重复经过同一个点。
- 输入格式
- 第一行输入两个整数 n (1≤n≤200) 和 m，表示一共有 n 个点和 m 条边。
- 接下来输入 n 行，每行输入两个整数 xi,yi (−500≤xi,yi≤500)，代表第 i 个点的坐标。
- 接下来输入 m 行，每行输入两个整数 pj,qj (1≤pj,qj≤n)，表示点 pj 和点 qj 之间相连。
- 输出格式: 输出一行，输出包含一个数，表示第二短的路径长度（小数点后面保留两位），如果第一短路径有多条，则答案就是第一最短路径的长度；如果第二最短路径不存在，则输出 −1。
- 样例输入
- 3 3
- 1 1
- 2 2
- 3 2
- 1 2
- 2 3
- 1 3
- 样例输出
- 2.41

7.2.2 SPFA 单源最短路算法

- SPFA（Shortest Path Faster Algorithm）算法是单源最短路径的一种算法，通常被认为是 Bellman-ford 算法的队列优化，在代码形式上接近于广度优先搜索 BFS，是一个在实践中非常高效的单源最短路算法。
- Dijkstra 不能处理有负权的图，而 SPFA 可以处理任意不含负环（负环是指总边权和为负数的环）的图 的最短路，并能判断图中是否存在负环。

SPFA 算法

- 在 SPFA 算法中，使用 d_i 表示从源点到顶点 i 的最短路，额外用一个队列来保存即将进行拓展的顶点列表，并用 inq_i 来标识顶点i 是不是在队列中。
- 1. 初始队列中仅包含源点，且源点 s 的 $d_s=0$。
- 2. 取出队列头顶点u，扫描从顶点u 出发的每条边，设每条边的另一端为v，边 <u,v> 权值为w，若 $d_u+w<d_v$，则
  - 将 d_v 修改为 d_u+w
  - 若 v 不在队列中，则
    - 将v 入队
- 3. 重复步骤 2 直到队列为空
- 最终d 数组就是从源点出发到每个顶点的最短路距离。如果一个顶点从没有入队，则说明没有从源点到该顶点的路径。
- 负环判断
- 在进行 SPFA 时，用一个数组 cnt_i 来标记每个顶点入队次数。如果一个顶点入队次数 cnt_i 大于顶点总数n，则表示该图中包含负环。

运行效率

- 很显然，SPFA 的空间复杂度为 O(V)。如果顶点的平均入队次数为k，则 SPFA 的时间复杂度为 O(kE)，对于较为随机的稀疏图，根据经验 k 一般不超过 4。
- SPFA 思想
- 在一定程度上，可以认为 SPFA 是由 BFS 的思想转化而来。从不含边权或者说边权为 1个单位长度的图上的 BFS，推广到带权图上，就得到了 SPFA。正如我们前面所说，SPFA 的本质是 Bellman-ford 算法的队列优化。由于 SPFA 没有改变 Bellaman-ford 的时间复杂度，国外一般来说不认为 SPFA 是一个新的算法，而仅仅是 Bellman-ford 的队列优化。

C++ 示例代码

```
bool inq[MAX_N];
int  d[MAX_N];    //  如果到顶点i的距离是
//0x3f3f3f3f，则说明不存在源点到i的最短路
void spfa(int s) {
    memset(inq, 0, sizeof(inq));
    memset(d, 0x3f, sizeof(d));
    d[s] = 0;
    inq[s] = true;
    queue<int> q;
    q.push(s);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        inq[u] = false;
for (int i = p[u]; i != -1; i = e[i].next) {
            int v = e[i].v;
            if (d[u] + e[i].w < d[v]) {
                d[v] = d[u] + e[i].w;
                if (!inq[v]) {
                    q.push(v);
                    inq[v] = true;
                }
            }
        }
    }
}
```

SPFA 算法有两个优化策略 SLF 和 LLL：

- SLF：Small Label First 策略，设要加入的顶点是 j，队首元素为i，若 d[j]<d[i]，则将j 插入队首，否则插入队尾；
- LLL：Large Label Last 策略，设队首元素为i，队列中所有最短距离值的平均值为x，若 d[i]>x 则将i 插入到队尾，查找下一元素，直到找到某一顶点i 使得 d[i]≤x，则将i 出队进行松弛操作。
- SLF 可使速度提高 15~ 20%；SLF + LLL 可提高约 50%。

- 在解决算法题目时，不带优化的 SPFA 就足以解决问题；而一些题目会故意制造出让 SPFA 效率低下的数据，即使你使用这两个优化也无法避免“被卡”。因此，SLF 和 LLL 两个优化仅作了解就可以了，在竞赛中不必使用。
- 对于稀疏图而言，SPFA 相比堆优化的 Dijkstra 有很大的效率提升，但是对于稠密图而言，SPFA 最坏为 O(VE)，远差于堆优化 Dijkstra 的 O((V+E)logV)。当然，在图中包含负权边时，SPFA 几乎是唯一的选择。因此，大家在做题时，还是要根据数据的具体情况来判断使用哪种最短路算法。

例题 1：最长路

- 在给定的图中，计算从源点到所有顶点的最长路。保证图中没有正环。
- 解法：将图中的边权取相反数，然后用 SPFA 计算图中的最短路就可以了。
- 或者把 SPFA 的更新操作修改为：
- • 若 $d_u+w>d_v$，则将 d_v 更新为 d_u+w
- 并在初始化时将 d 数组全部初始化为一个极小值（如0xbf），其余部分和用 SPFA 求最短路一样。

例题 2：换汇

- 你可以在银行进行各种货币的兑换，例如，你可以用1 美元兑换0.5 英镑， 1 英镑兑换10.0 法郎，1 法郎0.21 美元。这样，你就可以用初始的1 美元兑换出1.05 美元。像这样通过换汇获得盈利的过程被称为“套汇”。给定若干货币之间的汇率兑换比例，你需要判断其中是否存在“套汇”。
- 解法：在一个套汇路线中，如果初始货币为1 单位，则最终“套汇”路线就意味着边权乘积大于1 的环。根据对数的基本运算规则：
  - $\log(ab)=\log(a)+\log(b)$
- 如果我们把每条边权取对数，则环上所有边权对数的累加和大于0。因此，如果我们把边权进行如下的对应转换：
  - $w'=-\log(w)$
- 接下来只需要在转换后的图中判断是否存在负环就可以了。

习题：闯关游戏

- 小明在玩一个很好玩的游戏，这个游戏一共有至多 100 个地图，其中地图 1 是起点，房间n 是终点。有的地图是补给站，可以加ki 点体力，而有的地图里存在怪物，需要消耗 ki 点体力，地图与地图之间存在一些单向通道链接。
- 小明从 1 号地图出发，有 100 点初始体力。每进入一个地图的时候，需要扣除或者增加相应的体力值。这个过程持续到走到终点，或者体力值归零就会 Game Over。不过，他可以经过同个地图任意次，且每次都需要接受该地图的体力值。
- 输入格式: 第 1 行一个整数 n (n≤100)。第 2 ~ n+1 行，每行第一个整数表示该地图体力值变化。接下来是从该房间能到达的房间名单，第一个整数表示房间数，后面是能到达的房间编号。
- 输出格式: 若玩家能到达终点，输出Yes，否则输出No。
- 样例输入
- 5
- 0 1 2
- -60 1 3
- -60 1 4
- 20 1 5
- 0 0
- 样例输出
- No

习题：成仙之路

- 有个蘑菇精想要成仙，但是他必须要收集 10000 个精灵宝石，不过他要是有花精灵的泪水，就只要8000 个精灵宝石就可以了，或者如果他有花精灵的血滴，就只要5000 个精灵宝石便可以成仙了。蘑菇精可以和森林里的其他精灵交换东西，但是修为等级差距过大的交换会影响修炼
- 蘑菇精就跑到花精灵那里，向他索要泪水或血滴，花精灵要他用精灵宝石来换，或者替他弄来其他的东西，他可以降低价格。蘑菇精于是又跑到其他地方，其他精灵也提出了类似的要求，或者直接用精灵宝石换，或者找到其他东西就可以降低价格。不过蘑菇精没必要用多样东西去换一样东西，因为这不会得到更低的价格。蘑菇精现在很需要你的帮忙，让他用最少的精灵宝石帮助他成仙。另外他要告诉你的是，在这个森林里，交换东西，修为差距超过一定限制的两个精灵之间不会进行任何形式的直接接触，包括交易，否则会影响修炼。蘑菇精是外来精灵，所以可以不受这些限制。但是如果他和某个修为等级较低的精灵进行了交易，修为等级较高的的精灵有可能不会再和他交易，他们认为这样等于是间接接触，反过来也一样。因此你需要在考虑所有的情况以后给他提供一个最好的方案。
- 为了方便起见，我们把所有的物品从1开始进行编号，成仙也看作一个物品，并且编号总是1。每个物品都有对应的代价P，主人精灵的修为等级L，以及一系列的替代品Ti 和该替代品所对应的“优惠”Vi。如果两个精灵修为等级差距超过了M，就不能“间接交易”。你必须根据这些数据来计算出蘑菇精最少需要多少精灵宝石才能成仙。
- 输入格式：第一行是两个整数 M,N(1≤M≤20,1≤N≤20000)，依次表示修为等级差距限制和物品的总数。接下来按照编号从小到大依次给出了 N个物品的描述。每个物品的描述开头是三个P,L,X（X<N），依次表示该物品的代价、主人精灵的修为等级和替代品总数。接下来X 行每行包括两个整数T 和V，分别表示替代品的编号和“优惠价格”(∑X≤200000)。
- 输出格式: 对于每个测试数据，在单独一行内输出最少需要的精灵宝石数。

样例输入
1 4
10000 3 2
2 8000
3 5000
1000 2 1
4 200
3000 2 1
4 200
50 2 0
样例输出
5250

7.2.3 差分约束系统

- 差分约束系统是最短路的一类经典应用。如果一个不等式组由n个变量和m个约束条件组成，且每个约束条件都是形如 $x_i - x_j \le k, 1 \le i,j \le n$ 的不等式，则称其为 差分约束系统(system of difference constraints)。差分约束系统是求解一组变量的不等式组的算法。
- 问题转化
- 我们在求解差分约束系统时，可以将其转化为图论中单源最短路（或最长路）问题。
- 对于不等式中的其中一组 $x_j - x_i \le k$，我们会发现它类似最短路网络（全部由最短路上的边组成的子图）中的三角不等式 $d_v - d_u \le w<u,v>$，即 $d_u + w<u,v> \ge d_v$，所以我们可以理解成从顶点u 到顶点 v 连一条权值为 $w<u,v>$ 的边，用最短路算法得到最短路的答案 d_i，也就求出了原不等式组的一个解。
- 因此我们可以将每个变量xi作为一个顶点，对于约束条件 $x_j - x_i \le k$，连接一条边权为 k 的有向边 <i,j>。我们再增加一个超级源s，s 连向其余每个顶点，边权均为0。对这个图执行单源最短路算法，如果程序正常结束，那么得到的最短路答案数组di 就是满足条件的一组解；若图中存在负环，则该不等式组无解。

两种连边方法

- 连边有两种方法，第一种是连边后求最长路的方法，第二种是连边后求最短路的方法。
- 例： $x_j - x_i \le k$
- 若求最短路，则变形为 $x_i + k \ge x_j$，从i 到j 连一条权值为k 的边。
- 若求最长路，则变形为 $x_j - k \le x_i$，从j 到i 连一条权值为k 的边。

- 考虑到差分约束系统的边权可能为负，我们套用前面介绍的 SPFA 算法可解决这个问题。

习题：小明的银行卡

- 虽然小明并没有多少钱，但是小明办了很多张银行卡，共有 n 张，以至于他自己都忘记了每张银行卡里有多少钱了。
- 他只记得一些含糊的信息，这些信息主要以下列三种形式描述：
- 1. 银行卡a 比银行卡b 至少多c 元。
- 2. 银行卡a 比银行卡b 至多多c 元。
- 3. 银行卡a 和银行卡c 里的存款一样多。
- 但是由于小明的记忆有些差，他想知道是否存在一种情况，使得银行卡的存款情况和他记忆中的所有信息吻合。
- 输入格式: 第一行输入两个整数n 和m，分别表示银行卡数目和小明记忆中的信息的数目。(1≤n,m≤10000)
- 接下来 m 行：
- 如果每行第一个数是1，接下来有三个整数a,b,c，表示银行卡a 比银行卡b 至少多c 元。
- 如果每行第一个数是2，接下来有三个整数a,b,c，表示银行卡a 比银行卡b 至多多c 元。
- 如果每行第一个数是3，接下来有两个整数 a,b，表示银行卡a 和b 里的存款一样多。(1≤n,m,a,b,c≤10000)
- 输出格式: 如果存在某种情况与小明的记忆吻合，输出Yes，否则输出No。
- 样例输入
- 3 3
- 3 1 2
- 1 1 3 1
- 2 2 3 2
- 样例输出
- Yes

7.2.4 floyd 多源最短路算法

- Floyd 算法是一种利用动态规划的思想、计算给定的带权图中任意两个顶点之间最短路径的算法。相比于重复执行多次单源最短路算法，Floyd 具有高效、代码简短的优势，在解决图论最短路题目时比较常用。
- 算法过程
- Floyd 的基本思想是：对于一个顶点个数为 n 的有向图，并有一个 n×n 的方阵 G(k)，除对角元素 $G_{i,i}=0$ 以外，其他元素 $G_{i,j}(i\neq j)$ 表示从顶点 i 到顶点j 的有向路径长度。
- 1. 初始时 k=−1,G(−1)=E，E 是图的邻接矩阵，满足如下要求：
  - 对于任意两个顶点 i,j，若它们之间存在有向边，则以此边权上的权值作为$E_{i,j}$；
  - 若两个顶点 i,j 之间不存在有向边，则 $E_{i,j}$ 为无穷大 INF。
- 2. 对于阶段 k，尝试在G(k−1) 中增加一个中间顶点 k，如果通过中间顶点使得最短路径变短了，就更新作为新的 G(k) 的结果。
- 3. 累加 k，重复执行步骤 2，直到 k=n。
- 算法结束后，矩阵 G(n−1) 中的元素就代表着图中任意两点之间的最短路径长度。
- 通常，Floyd 算法用邻接矩阵来实现。空间复杂度为 $O(V^2)$，时间复杂度为 V^3。

C++代码

```
const int inf = 0x3f3f3f3f;
int g[MAX_N][MAX_N];  // 算法中的 G 矩阵

// 初始化 g 矩阵
void init() {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (i == j) {
                g[i][j] = 0;
            } else {
                g[i][j] = inf;
            }
        }
    }
}
// 插入一条带权有向边
void insert(int u, int v, int w) {
    g[u][v] = w;
}
// 核心代码
void floyd() {
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (g[i][k] + g[k][j] < g[i][j]) {
                    g[i][j] = g[i][k] + g[k][j];
                }
            }
        }
    }
}
```

例题：奶牛的比赛

有 n只奶牛参加比赛，其中有m 对关系，在每对关系(a,b) 中，奶牛a 一定可以赢奶牛b。如果奶牛a 可以赢奶牛b，奶牛b 可以赢奶牛c，则奶牛a 一定可以赢奶牛c，也就是说，关系具有传递性。确保给定的关系之间没有矛盾。问一共有多少只奶牛可以确定最终的排名。

输入：第1行：两个空格分隔的整数：n和m

第2行：m+1：每行包含两个空格分隔的整数，描述单轮比赛的竞争对手和结果（第一个整数，A是优胜者）：A和B

输出：第1行：表示可以确定其等级的奶牛数量的单个整数。

样本输入	样本输出
5 5 4 3 4 2 3 2 1 2 2 5	2

思路：

将 Floyd 算法的代码进行一些调整，矩阵G 保存的结果为：

- Gi,j=0：奶牛i 不确定能否赢奶牛j
- Gi,j=1：奶牛i 一定可以赢奶牛j

可以写出如下的核心代码：

```
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (g[i][k] && g[k][j]) {
                g[i][j] = true;
            }
        }
    }
}
```

算完以后，如果有a 只奶牛一定可以赢奶牛i，奶牛i 一定可以赢b只奶牛，且 a+b=n−1，则说明奶牛i 的排名是确定的。

习题：工厂年会

- 工厂要开年会了，所有的员工都要参加。
- 每两个员工之间都有一个亲密度。在同一个项目工作过的员工之间的亲密度为 1。如果 A 和 B、B 和 C 均在同一个项目中工作过，而 A 和 C 没有，那么 A 和 C 之间的亲密度为 1+1=2。
- 同理，如果 A 和 B 之间的亲密度为 x，B 和 C 之间的亲密度为y，则 A 和 C 之间的一种 可能亲密度 为 x+y。两个人之间的 亲密度 为所有的 可能亲密度 之中的 最小值。
- 因为工厂里员工之间非常有爱，所以 保证每两个员工之间都可以算出一个亲密度。
- 现在有一个名单，已知工厂在这一年一共进行过 M 个项目，N 名员工都想知道自己与其他所有员工的亲密度的平均值，现在你需要找出所有员工中，与其他所有员工 亲密度平均值 最小的员工，即和其他所有员工最亲密的一名员工，来担任这次年会的主持人。
- 输入格式
- 一行两个整数N 和 M，(1≤M≤10000,2≤N≤300)。
- 接下来 M 行，表示 M 个项目名单,每行第一个整数表示参加这一个项目的员工人数，后面是这些员工的编号，所有员工的编号从1 开始计数。
- 输出格式
- 一行一个整数，为最小的亲密度平均值乘 100 以后向下取整。
- 样例输入
- 4 2
- 3 1 2 3
- 2 3 4
- 样例输出
- 100

习题：小明的训练室

- 小明的训练室有 N 个站点，另外有M 条单向边连接这些站点。第i 条路从 Si 站到 Ei 站，有高度为 Hi 的围栏，小明是需要跳跃的。
- 现在小明们有 T 个任务要完成。第i 个任务，小明要从 Ai 站到Bi 站，小明想要他们路径中最高围栏尽可能小。请你确定这个高度。
- 输入格式：第一行输入三个整数 N,M,T。($1 \le N \le 300, 1 \le M \le 25000, 1 \le T \le 40000$)。接下来 M 行，每行三个整数 Si,Ei,Hi。($1 \le Si, Ei \le N, 1 \le Hi \le 10^6$)。再接下来 T 行，每行两个整数 Ai,Bi。($1 \le Ai, Bi \le N$)
- 输出格式：对于每个询问，输出最小的最大高度。若无法到达，则输出 −1。
- 样例输入
- 5 6 3
- 1 2 12
- 3 2 8
- 1 3 5
- 2 5 3
- 3 4 4
- 2 4 8
- 3 4
- 1 2
- 5 1
- 样例输出
- 4
- 8
- -1

习题：美好的邂逅

- 小明走在校园里，邂逅了一个美女，可是小明胆怯了，并没有上前搭讪。回到宿舍的小明越想越难过，好在室友提出了一个很好的办法，可以通过同学的同学的同学这样的关系，当同学的同学这样的关系叠加到足够大的时候，覆盖的人群也就会足够多，这样就能找到这个美女了，而且小明的室友很肯定的说，最多隔 6 个人，就一定能找到这个美女，小明有点不相信了，想验证一下室友的想法，他和室友一起对 N个人展开了调查，得到了他们之间的相识关系，现在请你编写一个程序判断一下，是不是这 N个人当中的每个人最多通过六个人就能找到其他任意一个人。
- 输入格式：第一行包含两个整数 N,M(0<N<100,0<M<200)，分别代表小明和他的室友统计的人数（这些人分别编成1 ~ N 号)，以及他们之间的关系数。接下来有 M 行，每行两个整数 A,B(1≤A,B≤N)表示编号为 A和编号 B 的人互相认识。
- 除了这 M 组关系，其他任意两人之间均不相识。
- 输出格式：如果数据能满足室友提出的猜想，那么就在一行里输出Yes，否则输出No。
- 样例输入
- 8 7
- 1 2
- 2 3
- 3 4
- 4 5
- 5 6
- 6 7
- 7 8
- 样例输出
- Yes