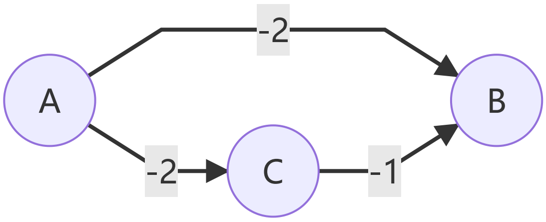
1

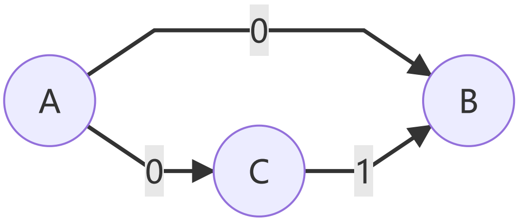
a

不可行. 考虑下图:



显然 $A \to B$ 的最短路为 $A \to C \to B$.

但若将每条边的权重都加 2 以消除负权:



$A \to B$ 的最短路变成了 $A \to B$.

b

为了记号的简洁性，我们将原题的 $h(s, v)$ 记为 h_v.

反之，若存在 $u, v \in V$，$w'(u, v) < 0$，即 $w(u, v) + h_u < h_v$，这恰恰说明 h_v 不是 s 到 v 的最短路径长，矛盾.

c

对于 u 到 v 的任意一条路径 $u \to t_1 \to \cdots \to t_n \to v$，其在新图中的路径长为

$$\begin{aligned} l' &= w'(u, t_1) + w'(t_1, t_2) + \cdots + w'(t_{n-1}, t_n) + w'(t_n, v) \\ &= w(u, t_1) + h_u - h_{t_1} + \cdots + w(t_n, v) + h_{t_n} - h_v \\ &= w(u, t_1) + w(t_1, t_2) + \cdots + w(t_{n-1}, t_n) + w(t_n, v) + h_u - h_v \\ &= l + h_u - h_v. \end{aligned}$$

其中 l 为同一路径在原图的路径长.

注意到，对任固定的 u, v，$l' - l$ 是固定的，与路径无关. 因此，新图上的最短路径就是原图上的最短路径.

由于新图上的权重都非负，Dijkstra 算法一定能正确给出 u 到 v 的最短路径长 m_{uv}，此时 $m_{uv} - h_u + h_v$ 即为原图中 u 到 v 的最短路径长.

注记

事实上，这正是 Johnson 算法.

一般来说，初始的 h_v 会用 Bellman-Ford 算法求出，时间复杂度为 $O(nm)$，其中 n 为图中顶点数，m 为边数.

之后，以每个顶点为源点运行一次 Dijkstra 算法，即可得到任意 u, v 间的最短路，这一步的时间为 $O(nm\log m)$，因此总的时间也是 $O(nm\log m)$.

对于稀疏图，Johnson 算法显著优于 Floyd 算法.

2

将罪犯视为顶点，矛盾对视为边，则原问题显然等价于判定该图是否是二分图.

这是一个经典问题，用 DFS 遍历所有顶点并进行着色即可.

```
int nV; // number of vertices
int color[N];   // 0 denotes undyed, 1, -1 denote two colors
int h[N];   // adjacent table head
int e[M], ne[M];    // end vertex (0 denotes none), next edge

// returns if dying fails
bool dfs(int i, int col){   // current vertex and its color
    color[i] = col;
    for (int p = h[i], j; p; p = ne[p]){
        j = e[p];
        if (color[j] == -col) continue;
        if (color[j] == col) return false;  // color conflict
        if (!dfs(j, -col)) return false;    // once fails, all impossible
    }
    return true;
}

// this function judges if this problem has a valid solution
bool solve(){
    if (nV <= 2) return true;
    for (int i = 1; i <= nV; ++i)   // traverse the entire graph
        if (!color[i] && !dfs(i, 1))
            return false;
    return true;
}

// ouput distribution plan
void output(){
    for (int i = 1; i <= nV; ++i)
        std::cout << "Prisoner No." << i << " should be located in prison " <<
(color[i]? 1: 2) << endl;
}
```

由于只有未被染色的顶点会触发 `dfs` 的调用，因此这个算法是 $O(n)$ 的，其中 n 为顶点数，即罪犯数目.

3

假设图 $G=<V,E>$ 存在两颗不同的最小生成树 $T_1=<V,E_1>,T_2=<V,E_2>,E_1\neq E_2$.

考虑 $E_1\otimes E_2$ 中权值最小的一条边 e，假设它在 E_1-E_2 中.

将其加入 T_2，会形成一个环. 我们断言，环中至少存在一条边 $e'\in E_2-E_1$，否则 T_1 有环. 由于 G 的边权互不相同，因此 e' 的权值大于 e.

那么，T_2+e-e' 也是一颗生成树，且其权值和小于 T_2，矛盾.

综上，边权互不相同的无向图的最小生成树唯一.

4

以同学间的排名先后关系建立有向无环图，按拓扑序遍历顶点并求出每个顶点的祖先（即直接或间接前驱）集.

具体方法是：为每个结点维护 n 位的 `std::bitset` 作为祖先集，每一位的 $0,1$ 代表对应标号的顶点是否是其祖先. 每访问一个点，就将其祖先集及该点本身加入该点每个后继的祖先集中——使用 `std::bitset` 的或运算即可.

同理，也可以求出每个结点的子孙（即直接或间接后继）集.

显然，一个点的排名能被确定，当且仅当其祖先与子孙数量之和为 $n-1$. 于是，统计出 `std::bitset` 中的 1 的数量，我们就可以知道哪些点的排名能被确定了.

由于图中有 $O(n)$ 条边，每条边都会引起 $O(n)$ 开销的祖先集更新，因此遍历时间为 $O(n^2)$. 统计 1 的数量也是 $O(n)$ 的，总共有 n 个顶点，因此这一步的开销也是 $O(n^2)$. 综上，整个算法是 $O(n^2)$ 的.

5

以 n 个城市和一口井为顶点建立无向图，两个城市 i,j 间的边权为 $b[i,j]$，城市 i 和井间的边权为 $a[i]$. 求解这张图的最小生成树，显然，其权值和即为最小花费.

由于是稀疏图，堆优化 Prim 算法或 Kruskal 算法都可以达到时间 $O(n\log n)$.