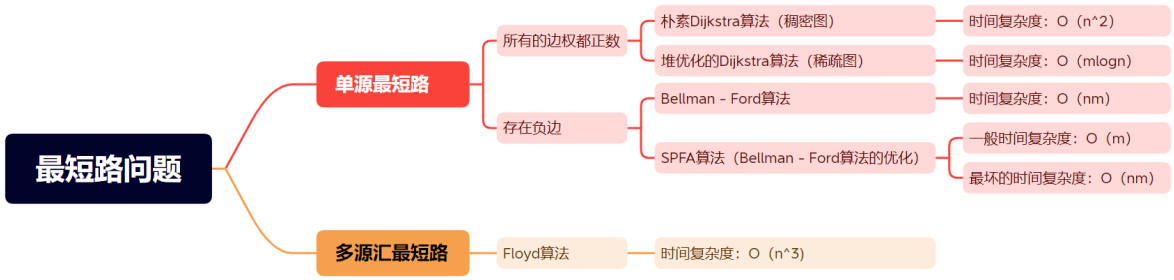
算法基础（十二）： 图 - 最短路问题2 - Bellman Ford与SPFA

复习一下这张图：



下面开始带负边的单源最短路分析

Bellman Ford

基本思想：

`Bellman - ford` 算法其原理为连续进行松弛，在每次松弛时把每条边都更新一下

松弛的意思就是外层的`for`循环每循环一次就表示一次松弛，`bellman - ford`算法是一个很傻的算法，它在每一次的松弛中都会遍历所有的边，然后更新对应的顶点，在下一次的松弛中又会利用上一次松弛的结果来继续更新所有的边

伪代码：

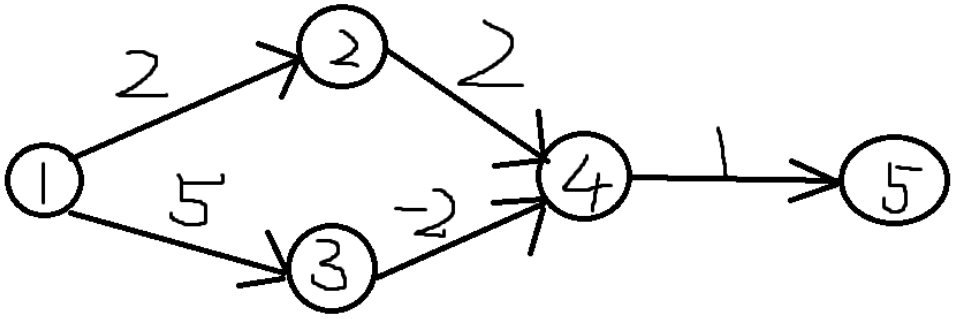
```
for n次
    for 所有边 a,b,w（松弛操作）
    dist[b] = min(dist[b],back[a] + w)
```

两重循环，时间复杂度是`O(nm)`

需要注意的有以下几点：

1. **为什么`Dijkstra`不能解决负权的边**

我们看如下的一张图，这张图来自`acwing`的题解



在`dijkstra`中

首先`dist[1] = 0`，从`1`点开始

第一次，更新`2 3`点，得到`dist[2] = 2, dist[3] = 5`，`st[1] = true`表示`1`号点加入到了`S`集合

第二次，从`2`点开始，更新`4`，得到`dist[4] = 4`，然后`st[2] = true`表示`2`号点加入到了集合`S`

第三次，我们发现最短路径是`4`号点，然后更新的是`5`号点，`st[4] = true`表示`4`号点加入到了集合`S`

我们可以看到，`4`号点这是在`dijkstra`中表示已经是最短距离了，下次更新点的时候不再更新它，没有考虑负边的存在

但是`Bellman - ford`不一样，它始终遍历所有的边进行更新，从而不会错过负边

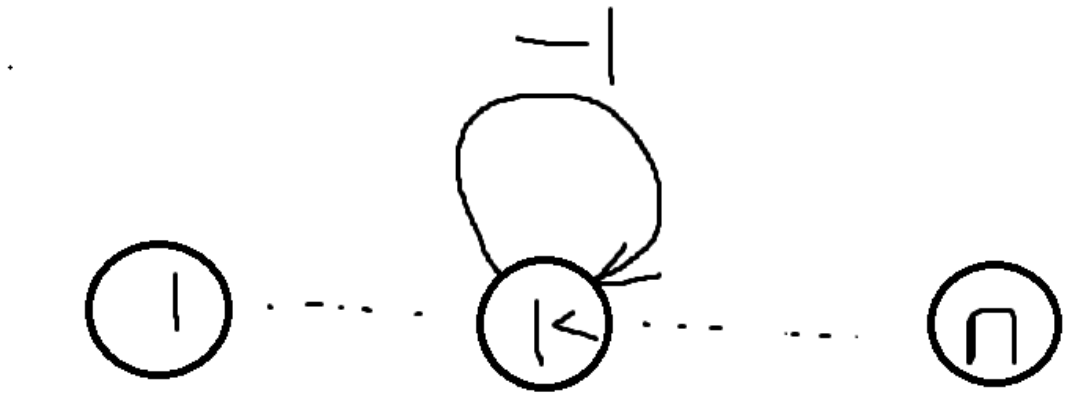
2. **松弛操作外层循环的意义**

外层循环`k`次代表的意义就是，源点经过不超过`k`条边到达每个点的最短距离，若源点到这个点的距离仍然是正无穷，表示源点经过不超过`k`条边无法到达这个点，而在第`k`次的时候又更新了某些顶点的话就说明存在一条最短路径它上面的边的个数是`k`条边

比如上图中经过一次松弛操作，`1`可以到达`2`，此时的`dist[]`数组表示的就是经过一次松弛操作源点到达图中各点的最短距离

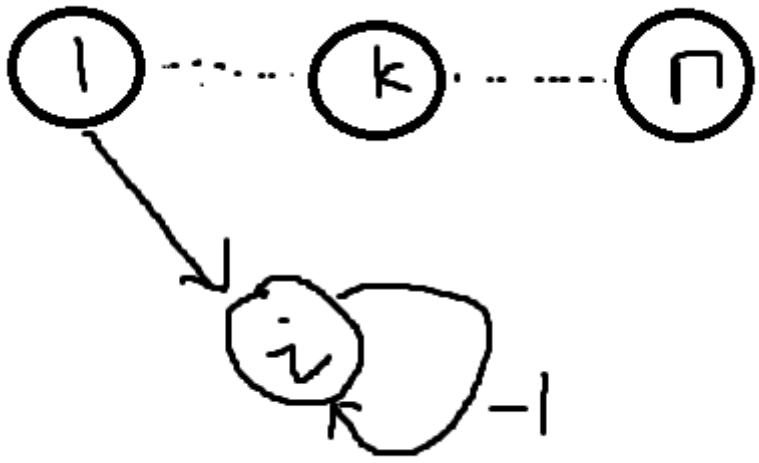
3. **关于负环的问题**

当图中存在负环的时候假如源点可以到达点`n`，`Bellman - ford`算法不一定可以找到源点到点`n`的最短距离，但是可以找到经过不超过`k`条边到达点`n`的最短距离，比如在源点到`n`的路径上存在一个负环



这时我们进行松弛操作，每次更新边的时候都会`-1`(在算法中是更新`-1`这条边)，这样会导致`dist[k] > dsit[k - 1] + w`一直更新下去，直到负无穷，所以源点到`n`的距离会一直减小，不存在最短距离

但是有负环不一定代表源点到`n`的最短路径不存在，比如下面这个图：



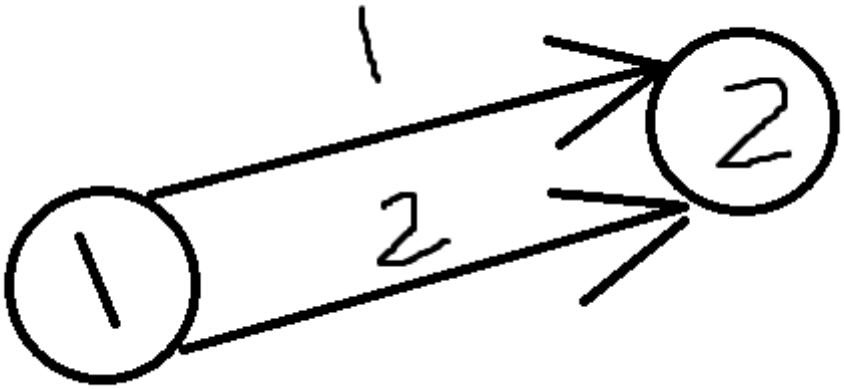
图中虽然存在负环，但是源点到`n`的路径上面没有负环，所以到`n`的最短路径还是存在的

同样`Bellman - ford`算法可以判断图中是否存在负环，假设图中有`n`个点，当经过`n - 1`次松弛操作之后，源点到图中的每一个点的最短距离（假设可以达到）都经过了不超过`n - 1`条边，**当经过`n`次松弛操作之后仍然有更新，说明图中必然有负环**，因为源点到`k`（小于等于`n`)点不经过环的情况下经过的边的个数不超过`n - 1`，而在`n`次松弛的时候更新，说明某个点到源点的最短距离经过了`n`条边，`n`条边上有`n + 1`个顶点，所以必然有两个顶点是相同的，所以必然经过了环，而它又更新了，说明长度在减小，说明必然经过的是负环。

综上，`Bellman - ford`算法解决的是边存在负权的情况下，找到经过不超过`k`条边，源点到图中点的最短路径，加入源点到顶点`n`的最短距离存在的话，我们经过`n`次松弛必然可以找到。

4. **关于重边的问题：**

假如下面的这个图：



我们在更新的时候，用的是这样一个语句`dist[b] = min(dist[b],back[a] + w)`，在一轮的松弛过程中，当遍历到权值为`1`的这条边的时候条件是`dist[b] > back[a] + 1`，于是更新`dist[b]`，然后遍历到权值为`2`的这条边的时候`dist[b] < back[a] + 2`，不会更新，也就忽略了权值为`2`的这条重边

5. **关于自环（正值）的问题**

其实跟上面的重边处理类似，自环始终是`dist[a] < backup[a] + w`的，不会更新`dist[a]`，也就忽略了自环

6. **关于`backup[]`数组**

其实在后面代码中会说这个问题，`bellman`算法是分层次的，当前层次松弛更新的结果`dist`只能在下层的松弛过程中使用，因为在本层松弛时其会更新，所以我们在本层松弛的时候必须记住原来的`dist[]`，所以将其保存到`backup[]`数组

代码实现：

给定一个 n 个点 m 条边的有向图，图中可能存在重边和自环， **边权可能为负数。**

请你求出从 1 号点到 n 号点的最多经过 k 条边的最短距离，如果无法从 1 号点走到 n 号点，输出 `impossible`。

注意：图中可能 **存在负权回路**。

输入格式

第一行包含三个整数 n, m, k。

接下来 m 行，每行包含三个整数 x, y, z，表示存在一条从点 x 到点 y 的有向边，边长为 z。

点的编号为 $1 \sim n$。

输出格式

输出一个整数，表示从 1 号点到 n 号点的最多经过 k 条边的最短距离。

如果不存在满足条件的路径，则输出 `impossible`。

数据范围

$1 \leq n, k \leq 500$,
$1 \leq m \leq 10000$,
$1 \leq x, y \leq n$,
任意边长的绝对值不超过 10000。

输入样例：

```
3 3 1
1 2 1
2 3 1
1 3 3
```

输出样例：

```
3
```

```cpp
#include<iostream>
#include<cstring>
#include<algorithm>
using namespace std;
const int N = 510, M = 15010;
```

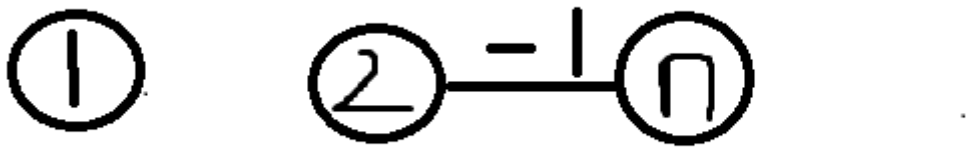
```cpp
int n, m, k;

int dist[N], backup[N];
//定义一个结构体来存所有的边
//beelman-ford算法是一个很傻的算法，他就是两重循环，内层循环遍历所有的边
//通过内存循环遍历一遍边之后，更新边指向的那个点的距离
//所以我们可以简单一点，直接存储边的信息然后直接遍历即可
//当然也可以按照原来的邻接表的形式存，不过这种方式更加直接
struct Edge
{
    //a, b表示边的起点和终点，w表示权重
    int a, b, w;
}edges[M];

int bellman_ford()
{
    memset(dist, 0x3f, sizeof dist);
    //不超过k条边的最短路，所以迭代k次
    //如样例
    dist[1] = 0;
    for(int i = 0; i < k; i ++)
    {
        //为了防止串联，保证不超过k条边的最短路
        //在更新某个顶点的最短距离的时候dsit[b] = dist[a] + w
        //但是dsit[a]可能在内存for循环的某次循环中被更改了，我们必须使用的是原来的循环
        //所以得先备份一份原来的，也就是back[]数组的作用
        memcpy(backup, dist, sizeof dist);
        //遍历所有的边
        //其实本质上和前面的遍历所有的点更新是一样的
        for(int j = 1; j <= m; j ++)
        {
            //cout << "b: " << dist[b] << " ";
            int a = edges[j].a, b = edges[j].b, w = edges[j].w;
            dist[b] = min(dist[b], backup[a] + w);
            //cout << "b: " << dist[b] << " ";
        }
        //cout << endl;
    }

    //应该直接返回dist[n]，然后在主函数中判断路径是否存在
    //因为带有负权所以dist[n]也可能等于 -1, return 也是返回-1，这时如果用原来的代码
    //会导致dist[n] = -1的时候输出也是impossible
    //if(dist[n] > 0x3f3f3f3f / 2) return -1;
    return dist[n];
}

int main()
{
    scanf("%d%d%d", &n, &m, &k);

    for(int i = 1; i <= m; i ++)
    {
```

```
        //读入所有的边
        int a, b, w;
        cin >> a >> b >> w;
        edges[i] = {a, b, w};
    }

    int t = bellman_ford();
    //cout << t <<endl;
    if(t > 0x3f3f3f3f >> 1) puts("impossible");
    else cout << t << endl;

    return 0;
}

```

需要注意的是：

```
if(t > 0x3f3f3f3f >> 1) puts("impossible");
else cout << t << endl;
```

这段代码，为什么`if`中的判断条件是`t > 0x3f3f3f3f >> 1`，我们需要注意，这个算法在每次松弛的过程中都会更新所有的顶点，假如有如下这种情况：



`1`到`2`不可达，所以`dist[2] = 0x3f3f3f3f`，因为我们无穷设置的是一个比较大的数，所以在每次松弛的时候更新点`n`的时候始终满足`dsit[n] > dist[2] + 1`，在更新的过程中`dsit[n]`这个比较大的数仍然会减小，不能一直保持`0x3f3f3f3f`，但他仍然表示无法达到，而且假如它的权值是`-10000`，松弛`500`次，每次都减去这个数，它最后还是大于`0x3f3f3f3f >> 1`，所以我们用大于`0x3f3f3f3f >> 1`来判断无法达到

另外，我们原来的最短路的算法中返回是这样写的：

```
if(dist[n] > 0x3f3f3f3f / 2) return -1;
return dist[n];
```

然后在主函数中写如下的代码：

```
t = bellman_ford;
if(t == -1) puts("impossible");
else cout << t << endl;
```

我们需要注意的是，由于存在负边，所以`dist[n]`的值也可能是`-1`，这时最短路是存在的，但是我们返回到主函数是`t == -1`然后输出`impossible`，结果不对，所以改成上面正确的写法。

Spfa

基本思想：

`spfa`算法是对`bellman - ford`算法的优化，我们注意`bellman - ford`算法的一次松弛过程：

```
for(int j = 0; j < m; j ++)
{
    int a = edges[j].a, b = edges[j].b, w = edges[j].w;
    min(dist[b], backup[a] + w );
}
```

其中代码`min(dist[b], backup[a] + w );`中的`dist[b]`发生更新的时候，`dsit[a]`必然在上一次的松弛中更新了，因为对于边`a -> b`，权重是`w`，如果在上一次松弛中`dist[a]`没有更新，那么上一次松弛后必然满足`dist[b] <= dist[a] + w`

于是在本次松弛中，由于仍然满足`dist[b] <= backup[a] + w`，所以`dist[b]`不会被更新

所以我们发现一个规律就是，只有当上一次松弛过程中点`i`的`dist[i]`更新了，那么在这一次松弛边`i -> k`，点`k`的`dist[k]`才会被更新

所以我们可以记录上一次松弛更新的点，在这一次松弛中只更新这些更新的点的出边对应的点，而其他的点跳过，这就是对`bellman - ford`算法的优化，也就是`spfa`算法

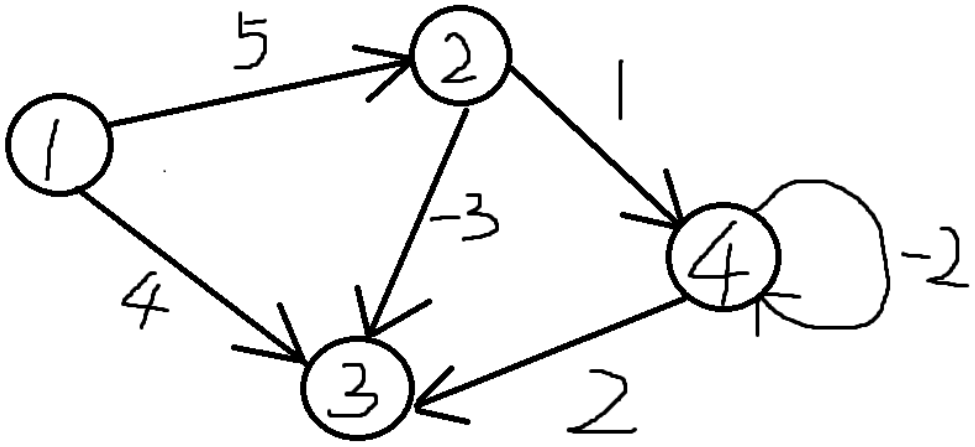
我们用一个队列来记录所有更新后的点，比如对于边`a -> b`，如果`b`的`dist[b]`发生了更新，我们就将点`b`加入到队列中，表示本次松弛过程`b`被更新，说明`b`的邻边可能会更新

最开始是源点`1`，`dist[1] = 0`，将`1`加入队列中，然后对于`1 -> k`的每一条边，若有`dsit[k] > dsit[1] + w`，将点`k`加入到队列中表示`k`的`dist[k]`被更新，而到出队列到点`k`的时候再对`k`的每一条边`k -> j`做如上操作

需要解决的一些问题：

1. **过程是怎样的？最后队列为空是什么情况？**

以下图为例：



进入`while`循环之前，第一个点入队列，`dist[1] = 0`，此时队列的元素为`1`， `st[] = t f f f`

进入`while`循环：

第一次`while`循环，第一个点出队列，此时队列元素为空，`st[] = f f f f`；遍历`1`的邻接点`2 3`，满足条件`dist[j] > dist[1] + w[i]`，`dsit[2] = 5， dist[3] = 4`，将这两个顶点加入队列，此时队列元素：`2 3`，`st[] = f t t f`

第二次`while`循环，顶点`2`出队列 ，此时队列元素为`3`，`st[] = f f t f`；遍历`2`的邻接点`4`，满足条件`dist[4] > dist[1] + w[i]`，`dist[4] = 6`，加入队列，此时队列元素`3 4`，`st[] = f f t t`；遍历`2`的邻接点`3`，满足条件`dist[3] > dist[1] + w[i]`，`dist[3] = 2`，虽然此时我们应该把顶点`3`加入队列，表示它的邻接点在下次应该更新，但是由于 队列中已经有顶点`3`了，在下一次`3`出队列的时候，我们就可以利用此时更新的`dsit[3]`去更新`3`的邻接点，也就不需要再重复加入`3`进队列了

第三次`while`循环，`3`出队列，`3`没有邻接点，此时队列元素`4`，`st[] = f f f t`

第四次`while`循环，`4`出队列，此时队列元素为空， `st[] = f f f f`，

遍历`4`的邻接点`4`，满足条件：`dist[4] > dist[4] + w[i]`，`dist[4] = 4`，加入队列，此时队列元素`4`，`st[] = f f f t`；遍历`4`的邻接点`3`，不满足条件

第五次`while`循环，`4`出队列，此时队列元素为空， `st[] = f f f f`，

遍历`4`的邻接点`4`，满足条件：`dist[4] > dist[4] + w[i]`，`dist[4] = 2`，加入队列，此时队列元素`4`，`st[] = f f f t`；遍历`4`的邻接点`3`，不满足条件

第六次`while`循环，`4`出队列，此时队列元素为空， `st[] = f f f f`，

遍历`4`的邻接点`4`，满足条件：`dist[4] > dist[4] + w[i]`，`dist[4] = 0`，加入队列，此时队列元素`4`，`st[] = f f f t`；遍历`4`的邻接点`3`，不满足条件

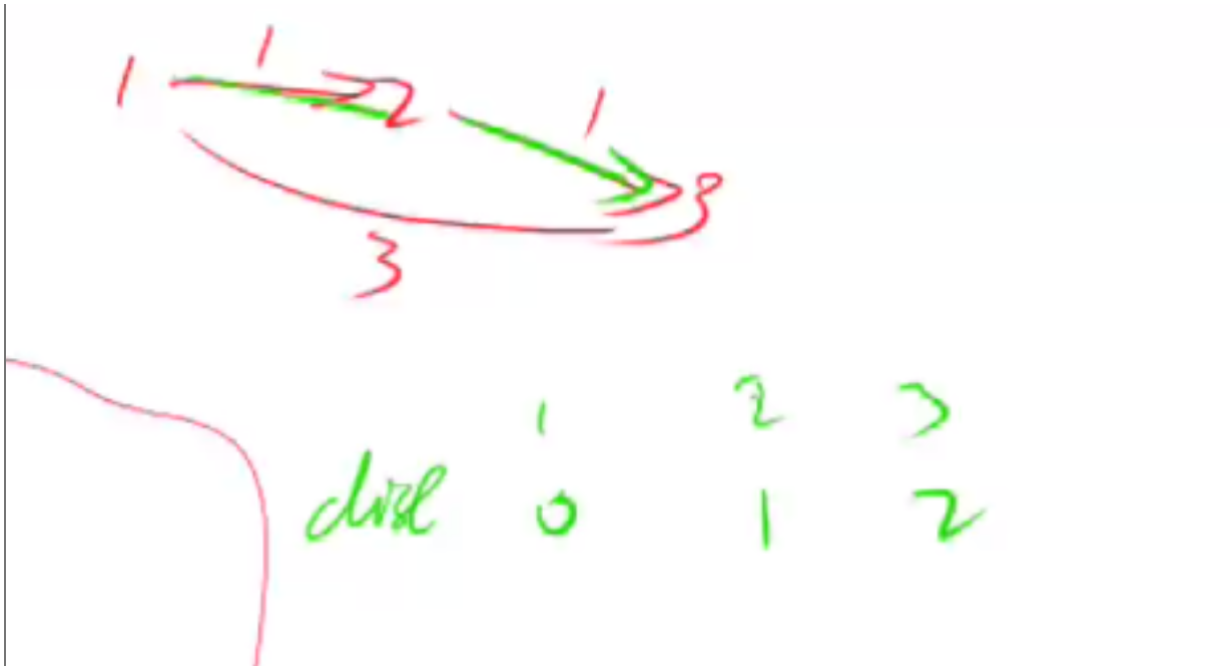
第七次`while`循环`4`出队列，此时队列元素为空， `st[] = f f f f`，

遍历`4`的邻接点`4`，满足条件：`dist[4] > dist[4] + w[i]`，`dist[4] = -2`，加入队列，此时队列元素`4`，`st[] = f f f t`；遍历`4`的邻接点`3`，满足条件`dsit[3] > dsit[4] + w[i]`，此时`dsit[3] = 0`，`3`加入队列，此时队列元素`4 3`，`st[] = f f t t`

然后重复类似于上面的操作，由于负环而进入无限循环

2. **为什么不需要backup数组?**

先看我们在 `bellman - ford` 算法中为什么使用 `backup` 数组，假如图是以下的情况：



在 `bellman` 算法中我们要求不经过一条边，源点到图中点的最短距离，如果不使用 `backup` 数组，再更新完 `1->2` 这条边后，再更新 `2->3` 的时候会将 `dist[3]` 更新为 `dsit[3] = 1 + 1 = 2`，显然 `dsit[3] = 2` 是经过了两条边，不符合题意，所以我们再此处需要使用 `backup` 数组

通过这个分析我们也可以发现 `spfa` 与 `bellman` 的不同之处，`spfa` 始终更新的是最短的距离，而无法做到经过几条边的限制

3. **他的次序是按照一次次 `bellman` 松弛的顺序操作的吗？可以使用别的数据结构吗？比如栈**

通过上述分析，`spfa` 虽然是对 `bellman` 的优化，但是它并不是按照 `bellman` 那样的松弛顺序进行顶点距离的更新的，`bellman` 算法通过 `backup` 数组，每一轮仅用原来的 `dsit[]` 即 `backup[]` 数组这个一轮轮的顺序，来满足经过了 `k` 条边这个限定

而这个顺序对 `spfa` 是没有意义的，因为 `spfa` 更新邻接点的时候都是使用上一个点最新的 `dsit[]` 进行更新的，比如边 `a -> b`，更新 `dsit[b]` 的时候使用的是最新更新过的 `dist[a] + w`，而若 `b` 更新之后，假如 `b` 有邻接点 `c`，那么就得将 `b` 加入队列，表示它的邻接点 `c` 也可能更新，而若原来的队列中已经有 `b` 了的话，我们就使用 `st[]` 数组，保证队列中不重复出现顶点，但是仍更新 `dsit[b]`，不将 `b` 加入队列，这样下次更新 `b` 的邻接点的时候仍然是用最新的 `dsit[b]`，没有影响

所以 `spfa` 的核心就两点：

1. 用一个数据结构记录更新了 `dist[]` 的点 `i`，表示 `i` 的邻接点下次会更新，下次从数据结构中取出 `i` 的时候用来更新 `i` 的邻接点，在 `i` 加入数据结构到取出 `i` 的这个过程中 `dist[i]` 可以继续更新
2. 保证每次更新 `i` 的邻接点的时候使用的 `dist[i]` 都是最新的

所以我们看到 `spfa` 其实不是太像对 `bellman` 的优化，或者说，不单单是记录更新了 `dist[]` 的点 `i`，表示 `i` 的邻接点下次会更新这样的优化，**`spfa` 更像是一个提出来的独立的算法**

所以，至于使用什么数据结构并没有影响，只用记录就行，比如上面的图，如果是一个栈的话 `1` 出栈，压入 `2 3`，`2` 出栈，压入 `4`，然后 `4` 反复出栈，入栈进入一个死循环，这是与队列等价的，所以 `spfa` 的核心思想与谁先更新是无关的

4. **它的时间复杂度是多少?**

严格来讲，`spfa`与`bellman - ford`才是处理最短路问题的一个通用算法，`Dijkstra`只是在没有负边的情况下将每次图中距离最短的点认为是已经找到最短距离了，不需要再更改，从这个角度，其实`Dijkstra`可以看作是在没有负边情况下的一个特殊处理，而且堆优化版本的`Dijkstra`的时间复杂度可以达到`nlog(m)`

对于`spfa`算法，他的数据结构中出现一个顶点后这个顶点在后面的情况下还是可能出现，所以它的时间复杂度不是`O(n)`，而且在每次遍历这个顶点（出队列）的时候它都会遍历这个顶点的边，所以当数据结构中不会重复进入顶点的情况下（而且没有负环），它最少遍历了顶点的所有的边，也就是说时间复杂度最少是`O(m)`

时间复杂度最多是`O(nm)`，复杂度`O(|V||E|)`。

`OI`比赛中，如果图的`|V||E|`不大（例如`|V| <= 1000`，`|E| <= 10000`），使用`SPFA`是靠谱的，而如果`|V||E|`较大（如`|V| <= 50000`，`|E| <= 100000`），那么`SPFA`可能就`TLE`了。

如果边权非负，建议使用复杂度稳定的`Dijkstra`。但有负权边的时候`Dijkstra`不能用，只能用`SPFA`。

至于时间复杂度为什么最多是这样的，有兴趣可以看看这个[解释

5. **他的负环和重边的处理过程是什么?**

负环的处理如上的例子分析所示，对于重边的处理，其实跟前面的几种算法的处理时一样的，比如重边`a -> b(权值：1)`，`a -> b(权值：2)`遍历`a`的所有邻接点的时候（其实时遍历所有的边），`j`会取到两次点`b`，每次都会比较`dsit[j]`与`dsit[a] + w`，这两个重边必然是选择一个权值小的更新`dsit[j]`

6. **他的自环(正值）的处理过程是什么?**

自环的处理跟重边类似，自环必然不满足`dist[j] > dist[t] + w[i]`这个条件，因为自环`t == j`，所以`dist[j] < dist[t] + w[i]`，自环会被忽略

代码实现：

给定一个 n 个点 m 条边的有向图，图中可能存在重边和自环， **边权可能为负数**。

请你求出 1 号点到 n 号点的最短距离，如果无法从 1 号点走到 n 号点，则输出 `impossible`。

数据保证不存在负权回路。

输入格式

第一行包含整数 n 和 m。

接下来 m 行每行包含三个整数 x, y, z，表示存在一条从点 x 到点 y 的有向边，边长为 z。

输出格式

输出一个整数，表示 1 号点到 n 号点的最短距离。

如果路径不存在，则输出 `impossible`。

数据范围

$1 \leq n, m \leq 10^5$,
图中涉及边长绝对值均不超过 10000。

输入样例：

```
3 3
1 2 5
2 3 -3
1 3 4
```

输出样例：

```
2
```

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

const int N = 100010;

int n, m;
int h[N], e[N], ne[N], w[N], idx;
int dist[N];
bool st[N];

void add(int a, int b, int c)
{
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx;
    w[idx] = c;
    idx ++;
}

int spfa()
```

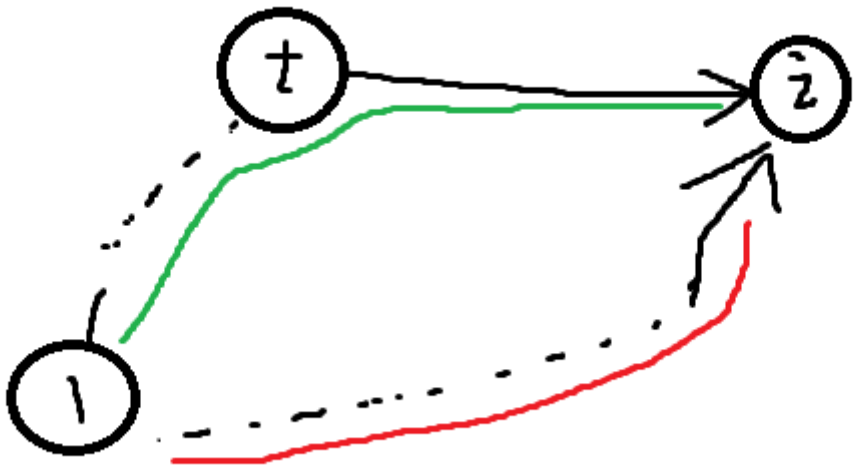
```
{
    memset(dist, 0x3f, sizeof dist);

    dist[1] = 0;
    queue<int> q;
    q.push(1);
    //st数组表示当前这个点是不是在队列当中
    //防止队列中存重复的点
    st[1] = true;

    while(q.size())
    {
        int t = q.front();
        q.pop();
        //表示这个点从队列中出来了
        st[t] = false;
       // cout <<"t:" << t << endl;
        for(int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if(dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                //假如队列中已经存在这个点了
                //我们只用更新一下这个点的dist[]就可以，不用再将这个点加入到队列中
                //加入队中没有这个点，我们需要把这个点加进去
                if(!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }

    }

    return dist[n];

}



int main()
{
    cin >> n >> m;
    memset(h, -1, sizeof h);

    while(m --)
    {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c);
```

```
    }

    int t = spfa();
    if(t == 0x3f3f3f3f) puts("impossible");
    else cout << t;

    return 0;
}


```

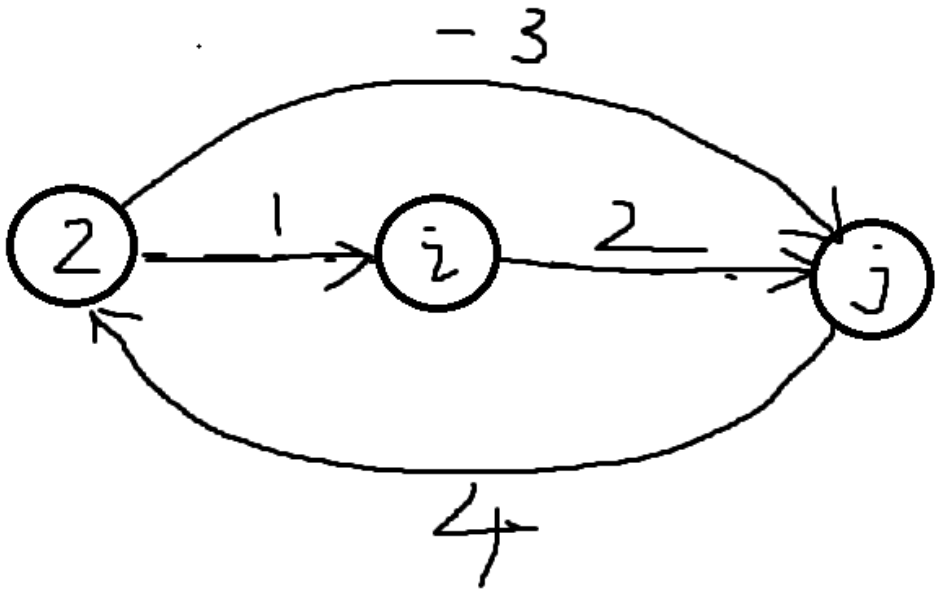
spfa判断负环：

基本思想：

我们进加入一个数组`cnt[]`，`cnt[i]`表示从源点到点`i`经过的边的个数，当我们更新`j`的`dist[j] = dsit[t] + w[i]`的时候还要更新一下`cnt[j] = cnt[t] + 1`，表示下图从第一个路径（红色）更新到第二个路径（绿色）

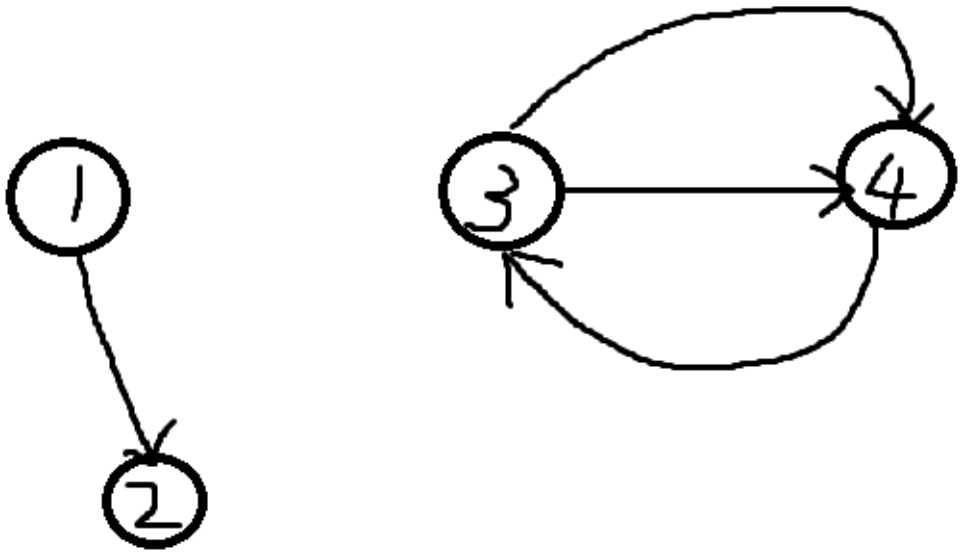


而对于，某一个点`j`，如果有`cnt[j] >= n`，因为有`n`个点，所以不经过环的情况下路径上的边的个数必然不大于`n - 1`，而`cnt[j] >= n`说明源点到`j`这条路必然经过了环，由于我们每次更新边的时候都会减小，如果这个环是正环的话是肯定能不会再往回绕的，如下图：



`2` 经过 `-3` 这条边到达 `j`，很显然 `dist[2] < dist[j](dist[2] - 3) + w[i]`，所以不往回绕，所以成环的话必然是负环

另外我们需要注意的是，这里判断是否有负环，但是这个负环图中的源点不一定可达，比如：



按照 `y` 老师原话就是，此时需要在原图的基础上新建一个虚拟源点，从该点向其他所有点连一条权值为 `0` 的有向边。那么原图有负环等价于新图有负环。此时在新图上做 `spfa`，将虚拟源点加入队列中。然后进行 `spfa` 的第一次迭代，这时会将所有点的距离更新并将所有点插入队列中。执行到这一步，就等价于直接将所有的点加入到队列中，并且所有的 `dist[]`，的值为 `0`。那么视频中的做法可以找到负环，等价于这次 `spfa` 可以找到负环，等价于新图有负环，等价于原图有负环.

代码实现：

给定一个 n 个点 m 条边的有向图，图中可能存在重边和自环， **边权可能为负数。**

请你判断图中是否存在负权回路。

输入格式

第一行包含整数 n 和 m。

接下来 m 行每行包含三个整数 x, y, z，表示存在一条从点 x 到点 y 的有向边，边长为 z。

输出格式

如果图中**存在**负权回路，则输出 `Yes`，否则输出 `No`。

数据范围

$1 \leq n \leq 2000,$
$1 \leq m \leq 10000,$
图中涉及边长绝对值均不超过 10000。

输入样例：

```
3 3
1 2 -1
2 3 4
3 1 -4
```

输出样例：

```
Yes
```

```
1
```

```
#include<iostream>
#include<cstring>
#include<algorithm>
#include<queue>
using namespace std;
const int N = 100010;

int h[N], e[N], ne[N], w[N], idx;
int n, m;
int st[N];
int dist[N];
int cnt[N];
void add(int a, int b, int c)
{
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx;
    w[idx] = c;
    idx ++;

```

```cpp
}

int spfa()
{
    queue<int> q;

    //等价于虚拟源点进行第一次迭代的结果
    for(int i = 1; i <= n; i ++)
    {
        q.push(i);
        st[i] = true;
        cnt[i] += 1;
    }

    //继续求所有的点到虚拟源点的距离
    //由于此时所有的点都相对于虚拟源点可达，所以此时算法可行

    while(q.size())
    {
        int t = q.front();
        q.pop();
        st[t] = false;

        for(int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if(dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                cnt[j] = cnt[t] + 1;
                //这里也可以用j >= n, 然后删掉上面的cnt[i] += 1, 两种写法是等价的
                if(cnt[j] >= n + 1) return true;
                if(!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    return false;

}

int main()
{
    cin >> n >> m;

    memset(h, -1, sizeof h);

    while(m --)
```

```
    {
        int a, b, c;
        cin >> a >> b >> c;
        add(a, b, c);
    }

    if(spfa()) puts("Yes");
    else puts("No");


    return 0;
}
```

还有一个问题就是关于无向图，其实无向图就等价于一条边有两个方向，我们在加入无向图的边的时候需要加入两条边，用两次`add`函数，然后用有向图的最短路算法计算即可