```
            printf("%d %d %d\n",u[k],v[k],w[k]);
            k=next[k];
        }
    }
```

可以发现使用邻接表来存储图的时间空间复杂度是 $O(M)$，遍历每一条边的时间复杂度也是 $O(M)$。如果一个图是稀疏图的话，M 要远小于 N^2。因此稀疏图选用邻接表来存储要比用邻接矩阵来存储好很多。

最后，本节介绍的求最短路径的算法是一种基于贪心策略的算法。每次新扩展一个路程最短的点，更新与其相邻的点的路程。当所有边权都为正时，由于不会存在一个路程更短的没扩展过的点，所以这个点的路程永远不会再被改变，因而保证了算法的正确性。不过根据这个原理，用本算法求最短路径的图是不能有负权边的，因为扩展到负权边的时候会产生更短的路程，有可能就破坏了已经更新的点路程不会改变的性质。既然用这个算法求最短路径的图不能有负权边，那有没有可以求带有负权边的指定顶点到其余各个顶点的最短路径算法呢？请看下一节。

最后说一下，这个算法的名字叫做 Dijkstra。该算法是由荷兰计算机科学家 Edsger Wybe Dijkstra（这神犇的中文译名的版本太多了，有上十种不重样的，这里就不写中文译名了）于 1959 年提出的。其实这个算法 Edsger Wybe Dijkstra 在 1956 年就发现了，当时他正与夫人在一家咖啡厅的阳台上晒太阳喝咖啡。因为当时没有专注于离散算法的专业期刊，直到 1959 年，他才把这个算法发表在 *Numerische Mathematik* 的创刊号上。

第 3 节　Bellman-Ford——解决负权边

Dijkstra 算法虽然好，但是它不能解决带有负权边（边的权值为负数）的图。本节我要来介绍一个无论是思想上还是代码实现上都堪称完美的最短路算法：Bellman-Ford。Bellman-Ford 算法非常简单，核心代码只有 4 行，并且可以完美地解决带有负权边的图，先来看看它长啥样：

```
for(k=1;k<=n-1;k++)
    for(i=1;i<=m;i++)
        if( dis[v[i]] > dis[u[i]] + w[i] )
            dis[v[i]] = dis[u[i]] + w[i];
```

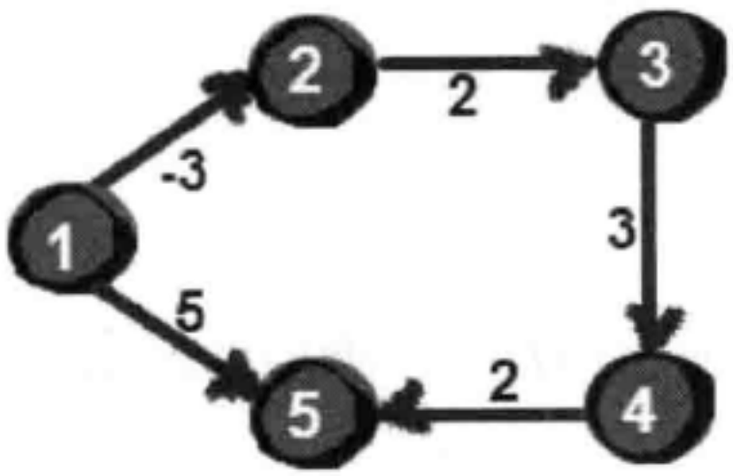
上面的代码中，外循环一共循环了 n–1 次（n 为顶点的个数），内循环循环了 m 次（m 为边的个数），即枚举每一条边。dis 数组的作用与 Dijkstra 算法一样，是用来记录源点到其余各个顶点的最短路径。u、v 和 w 三个数组是用来记录边的信息。例如第 i 条边存储在 $u[i]$、$v[i]$和 $w[i]$中，表示从顶点 $u[i]$到顶点 $v[i]$这条边（$u[i]$→$v[i]$）权值为 $w[i]$。

```
if( dis[v[i]] > dis[u[i]] + w[i] )
     dis[v[i]] = dis[u[i]] + w[i];
```

上面这两行代码的意思是：看看能否通过 $u[i]$→$v[i]$（权值为 $w[i]$）这条边，使得 1 号顶点到 $v[i]$号顶点的距离变短。即 1 号顶点到 $u[i]$号顶点的距离（dis[$u[i]$]）加上 $u[i]$→$v[i]$这条边（权值为 $w[i]$）的值是否会比原先 1 号顶点到 $v[i]$号顶点的距离（dis[$v[i]$]）要小。这一点其实与 Dijkstra 的“松弛”操作是一样的。现在我们要把所有的边都松弛一遍，代码如下。

```
for(i=1;i<=m;i++)
    if( dis[v[i]] > dis[u[i]] + w[i] )
         dis[v[i]] = dis[u[i]] + w[i];
```

那把每一条边都“松弛”一遍后，究竟会有什么效果呢？现在来举个具体的例子。求下图 1 号顶点到其余所有顶点的最短路径。



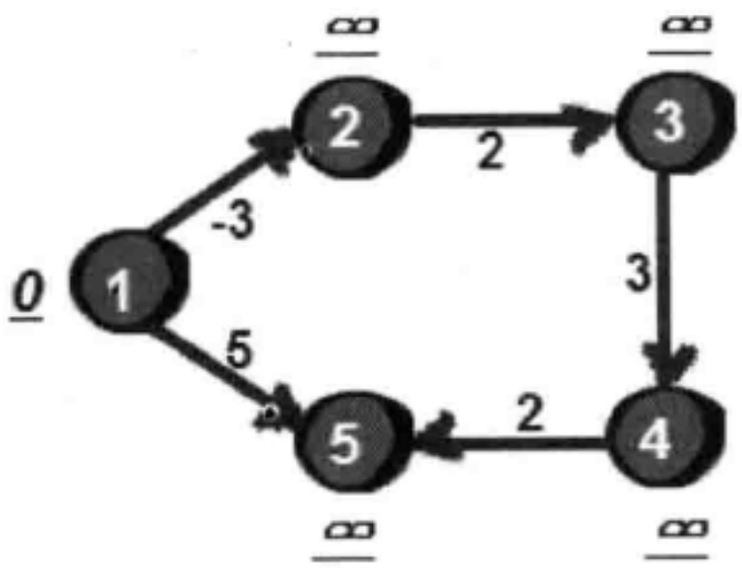
边给出的顺序是

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

我们还是用一个 dis 数组来存储 1 号顶点到所有顶点的距离。

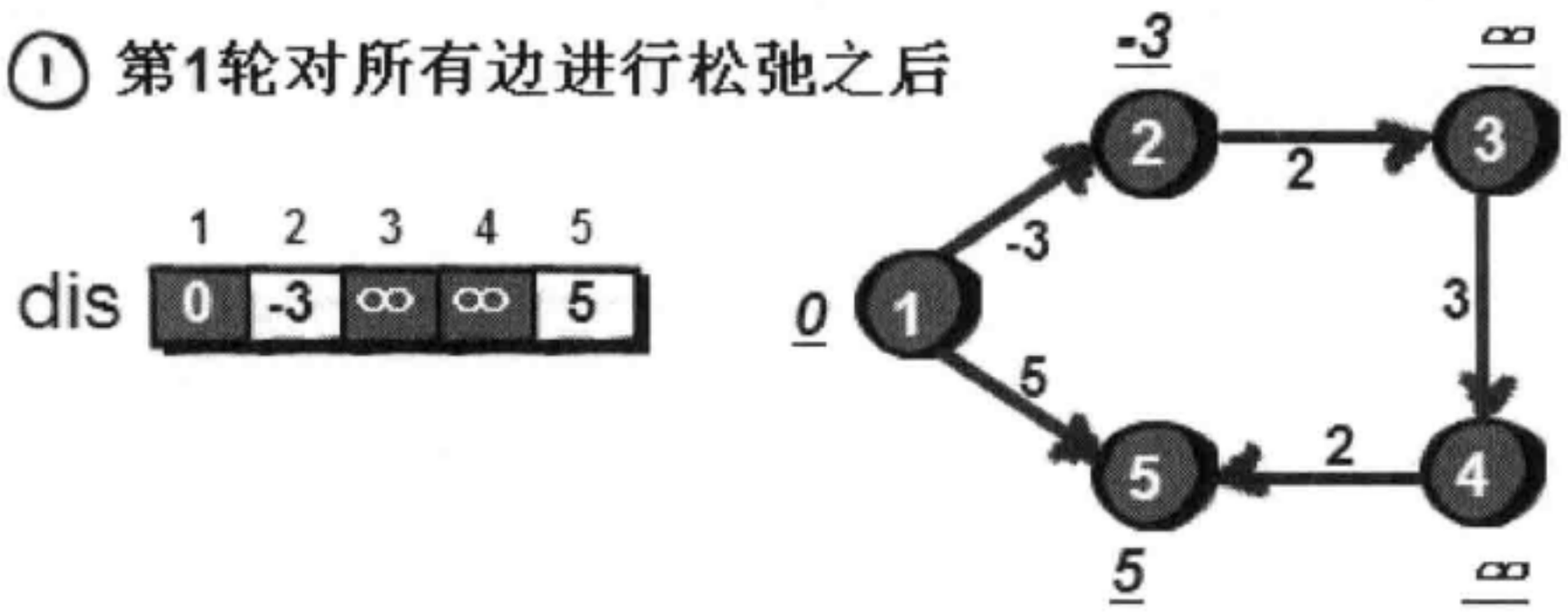
初始化

	1	2	3	4	5
dis	0	∞	∞	∞	∞



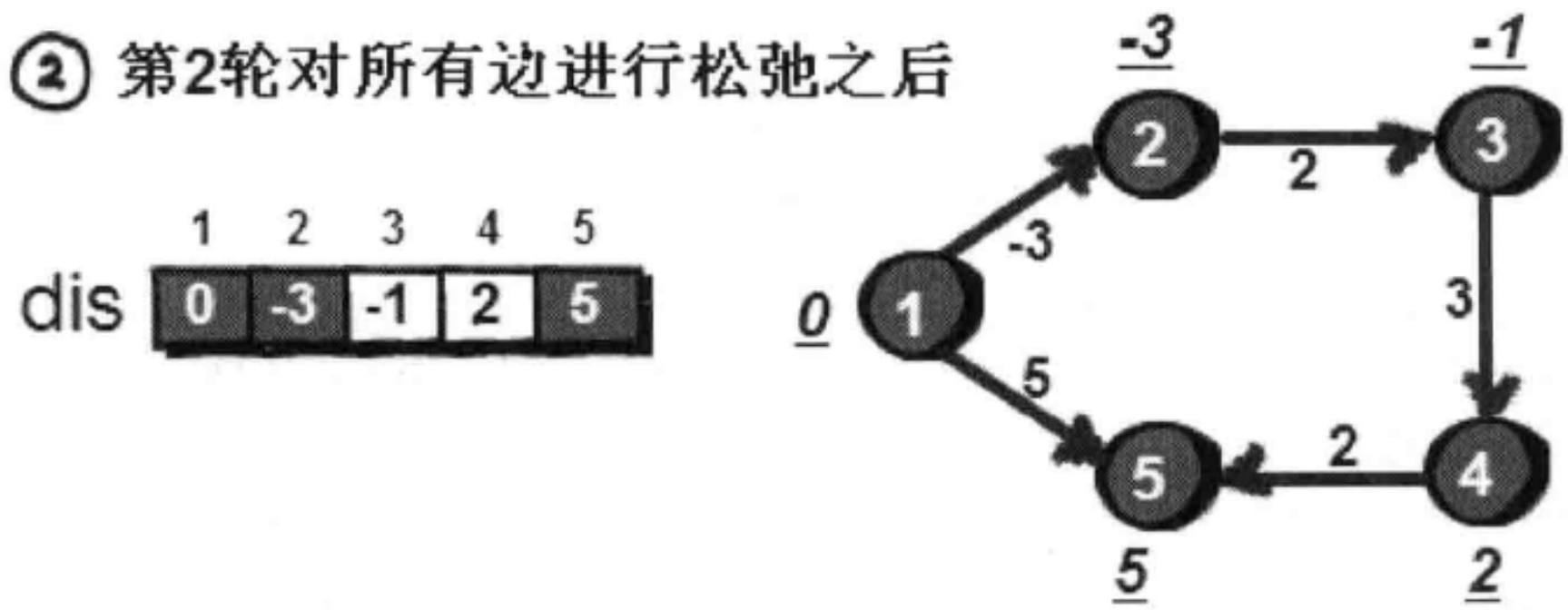
上方右图中每个顶点旁的值（带下划线的数字）为该顶点的最短路“估计值”（当前 1 号顶点到该顶点的距离），即数组 dis 中对应的值。根据边给出的顺序，先来处理第 1 条边“2 3 2”（$2\xrightarrow{2}3$，通过这条边进行松弛），即判断 dis[3] 是否大于 dis[2]+2。此时 dis[3]是∞，dis[2]是∞，因此 dis[2]+2 也是∞，所以通过“2 3 2”这条边不能使 dis[3]的值变小，松弛失败。

同理，继续处理第 2 条边“1 2 -3”（$1\xrightarrow{-3}2$），我们发现 dis[2]大于 dis[1]+(-3)，通过这条边可以使 dis[2]的值从∞变为-3，因此松弛成功。用同样的方法处理剩下的每一条边。对所有的边松弛一遍后的结果如下。



我们发现，在对每条边都进行一次松弛后，已经使得 dis[2]和 dis[5]的值变小，即 1 号顶到 2 号顶点的距离和 1 号顶点到 5 号顶点的距离都变短了。

接下来我们需要对所有的边再进行一轮松弛，操作过程与上一轮一样，再来看看又会发生什么变化。



在这一轮松弛时，我们发现，现在通过“2 3 2”（$2\xrightarrow{2}3$）这条边，可以使 1 号顶点到 3 号顶点的距离（dis[3]）变短了。爱思考的同学就会问了，这条边在上一轮也松弛过啊，为什么上一轮松弛失败了，这一轮却成功了呢？因为在第一轮松弛过后，1 号顶点到 2 号顶点的距离（dis[2]）已经发生了变化，这一轮再通过“2 3 2”（$2\xrightarrow{2}3$）这条边进行松弛的时候，已经可以使 1 号顶点到 3 号顶点的距离（dis[3]）的值变小。

换句话说，第 1 轮在对所有的边进行松弛之后，得到的是从 1 号顶点“只能经过一条边”到达其余各顶点的最短路径长度。第 2 轮在对所有的边进行松弛之后，得到的是从 1 号顶点“最多经过两条边”到达其余各顶点的最短路径长度。如果进行 k 轮的话，得到的就是 1 号顶点“最多经过 k 条边”到达其余各顶点的最短路径长度。现在又有一个新问题：需要进行多少轮呢？

只需要进行 $n-1$ 轮就可以了。因为在一个含有 n 个顶点的图中，任意两点之间的最短路径最多包含 $n-1$ 边。

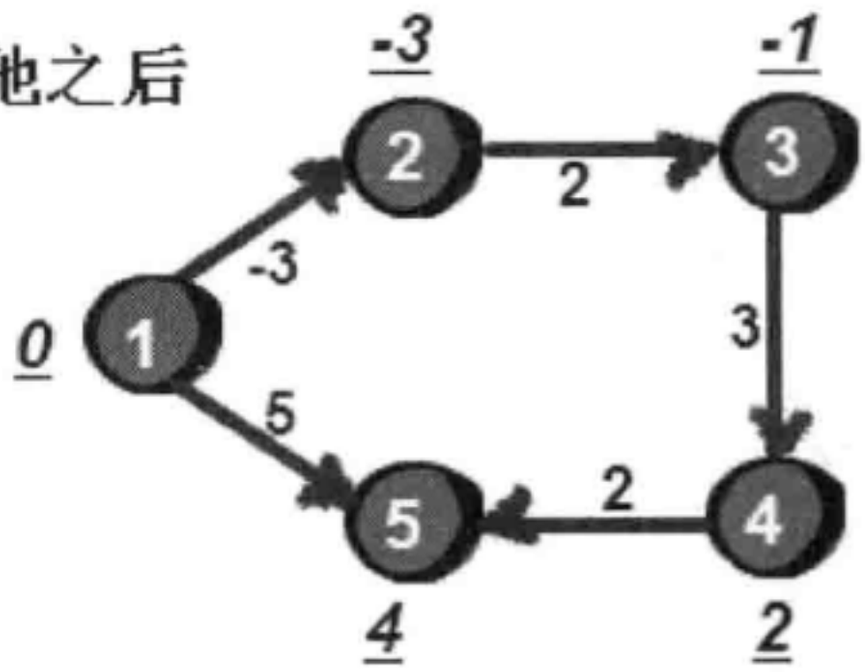
有些特别爱思考的同学又会发出一个疑问：真的最多只能包含 $n-1$ 条边？最短路径中不可能包含回路吗？

答案是：不可能！最短路径肯定是一个不包含回路的简单路径。回路分为正权回路（即回路权值之和为正）和负权回路（即回路权值之和为负）。我们分别来讨论一下为什么这两种回路都不可能有。如果最短路径中包含正权回路，那么去掉这个回路，一定可以得到更短的路径。如果最短路径中包含负权回路，那么肯定没有最短路径，因为每多走一次负权回路就可以得到更短的路径。因此，最短路径肯定是一个不包含回路的简单路径，即最多包含 $n-1$ 条边，所以进行 $n-1$ 轮松弛就可以了。

扯了半天，回到之前的例子，继续进行第 3 轮和第 4 轮松弛操作，这里只需进行 4 轮就可以了，因为这个图一共只有 5 个顶点。

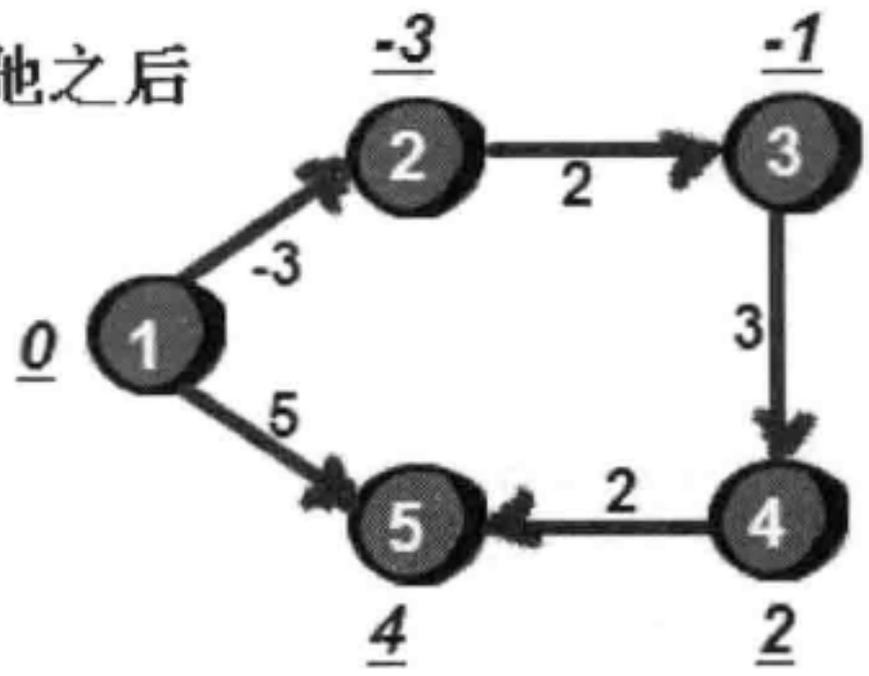
③ 第3轮对所有边进行松弛之后

	1	2	3	4	5
dis	0	-3	-1	2	4



④ 第4轮对所有边进行松弛之后

	1	2	3	4	5
dis	0	-3	-1	2	4



有些特别特别爱思考的同学又会有一个疑问，这里貌似不用进行第 4 轮嘛，因为进行第 4 轮之后 dis 数组没有发生任何变化！没错！你说的真是太对了！其实是**最多**进行 $n-1$ 轮松弛。

整个 Bellman-Ford 算法用一句话概括就是：对所有的边进行 $n-1$ 次“松弛”操作。核心代码只有 4 行，如下。

```
for(k=1;k<=n-1;k++) //进行n-1轮松弛
    for(i=1;i<=m;i++) //枚举每一条边
        if( dis[v[i]] > dis[u[i]] + w[i] ) //尝试对每一条边进行松弛
            dis[v[i]] = dis[u[i]] + w[i];
```

Ok，总结一下。因为最短路径上最多有 $n-1$ 条边，因此 Bellman-Ford 算法最多有 $n-1$ 个阶段。在每一个阶段，我们对每一条边都要执行松弛操作。其实每实施一次松弛操作，就会有一些顶点已经求得其最短路，即这些顶点的最短路的“估计值”变为“确定值”。此后这些顶点的最短路的值就会一直保持不变，不再受后续松弛操作的影响（但是，每次还是会判断是否需要松弛，这里浪费了时间，是否可以优化呢？）。在前 k 个阶段结束后，就已经找出了从源点发出“最多经过 k 条边”到达各个顶点的最短路。直到进行完 $n-1$ 个阶段后，便得出了最多经过 $n-1$ 条边的最短路。

Bellman-Ford 算法的完整的代码如下。

```
#include <stdio.h>
int main()
{
    int dis[10],i,k,n,m,u[10],v[10],w[10];
    int inf=99999999; //用inf（infinity的缩写）存储一个我们认为的正无穷值
    //读入n和m，n表示顶点个数，m表示边的条数
    scanf("%d %d",&n,&m);

    //读入边
    for(i=1;i<=m;i++)
        scanf("%d %d %d",&u[i],&v[i],&w[i]);

    //初始化dis数组，这里是1号顶点到其余各个顶点的初始路程
    for(i=1;i<=n;i++)
        dis[i]=inf;
    dis[1]=0;

    //Bellman-Ford算法核心语句
    for(k=1;k<=n-1;k++)
        for(i=1;i<=m;i++)
            if( dis[v[i]] > dis[u[i]] + w[i] )
                dis[v[i]] = dis[u[i]] + w[i];

    //输出最终的结果
```

```
    for(i=1;i<=n;i++)
        printf("%d ",dis[i]);

    getchar();getchar();
    return 0;
}
```

可以输入以下数据进行验证。第一行两个整数 *n m*。*n* 表示顶点个数（顶点编号为 1~*N*），*m* 表示边的条数。接下来 *m* 行表示，每行有 3 个数 *x y z*。表示从顶点 *x* 到顶点 *y* 的边的权值为 z。

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

运行结果是

```
0 -3 -1 2 4
```

此外，Bellman-Ford 算法还可以检测一个图是否含有负权回路。如果在进行 *n*−1 轮松弛之后，仍然存在：

```
if( dis[v[i]] > dis[u[i]] + w[i] )
    dis[v[i]] = dis[u[i]] + w[i];
```

的情况，也就是说在进行 *n*−1 轮松弛后，仍然可以继续成功松弛，那么此图必然存在负权回路。在之前的证明中我们已经讨论过，如果一个图如果没有负权回路，那么最短路径所包含的边最多为 *n*−1 条，即进行 *n*−1 轮松弛之后最短路不会再发生变化。如果在 *n*−1 轮松弛之后最短路仍然会发生变化，则该图必然存在负权回路，关键代码如下：

```
//Bellman-Ford算法核心语句
for(k=1;k<=n-1;k++)
    for(i=1;i<=m;i++)
        if( dis[v[i]] > dis[u[i]] + w[i] )
            dis[v[i]] = dis[u[i]] + w[i];
//检测负权回路
```

```
flag=0;
for(i=1;i<=m;i++)
    if( dis[v[i]] > dis[u[i]] + w[i] )  flag=1;
if (flag==1) printf("此图含有负权回路");
```

显然（请原谅我使用如此恶劣的词汇），Bellman-Ford 算法的时间复杂度是 O(*NM*)，这个时间复杂度貌似比 Dijkstra 算法还要高，我们还可以对其进行优化。在实际操作中，Bellman-Ford 算法经常会在未达到 *n*–1 轮松弛前就已经计算出最短路，之前我们已经说过，*n*–1 其实是最大值。因此可以添加一个一维数组用来备份数组 dis。如果在新一轮的松弛中数组 dis 没有发生变化，则可以提前跳出循环，代码如下。

```
#include <stdio.h>
int main()
{
    int dis[10],bak[10],i,k,n,m,u[10],v[10],w[10],check,flag;
    int inf=99999999; //用inf（infinity的缩写）存储一个我们认为的正无穷值
    //读入n和m，n表示顶点个数，m表示边的条数
    scanf("%d %d",&n,&m);

    //读入边
    for(i=1;i<=m;i++)
        scanf("%d %d %d",&u[i],&v[i],&w[i]);

    //初始化dis数组，这里是1号顶点到其余各个顶点的初始路程
    for(i=1;i<=n;i++)
        dis[i]=inf;
    dis[1]=0;

    //Bellman-Ford算法核心语句
    for(k=1;k<=n-1;k++)
    {
        //将dis数组备份至bak数组中
        for(i=1;i<=n;i++) bak[i]=dis[i];
        //进行一轮松弛
        for(i=1;i<=m;i++)
            if( dis[v[i]] > dis[u[i]] + w[i] )
                dis[v[i]] = dis[u[i]] + w[i];
        //松弛完毕后检测dis数组是否有更新
        check=0;
```

```
        for(i=1;i<=n;i++) if( bak[i]!=dis[i] ){check=1;break;}
        if(check==0) break; //如果dis数组没有更新，提前退出循环结束算法
    }
    //检测负权回路
    flag=0;
    for(i=1;i<=m;i++)
        if( dis[v[i]] > dis[u[i]] + w[i] )  flag=1;

    if (flag==1) printf("此图含有负权回路");
    else
    {
        //输出最终的结果
        for(i=1;i<=n;i++)
            printf("%d ",dis[i]);
    }
    getchar(); getchar();
    return 0;
}
```

Bellman-Ford 算法的另外一种优化在文中已经有所提示：在每实施一次松弛操作后，就会有一些顶点已经求得其最短路，此后这些顶点的最短路的估计值就会一直保持不变，不再受后续松弛操作的影响，但是每次还要判断是否需要松弛，这里浪费了时间。这就启发我们：**每次仅对最短路估计值发生变化了的顶点的所有出边执行松弛操作**。详情请看下一节：Bellman-Ford 的队列优化。

美国应用数学家 Richard Bellman（理查德·贝尔曼）于 1958 年发表了该算法。此外 Lester Ford, Jr.在 1956 年也发表了该算法。因此这个算法叫做 Bellman-Ford 算法。其实 Edward F. Moore 在 1957 年也发表了同样的算法，所以这个算法也称为 Bellman-Ford-Moore 算法。Edward F. Moore 很熟悉对不对？就是那个在“如何从迷宫中寻找出路”问题中提出了广度优先搜索算法的那个家伙。

第 4 节　Bellman-Ford 的队列优化

在上一节，我们提到 Bellman-Ford 算法的另一种优化：每次仅对最短路程发生变化了的点的相邻边执行松弛操作。但是如何知道当前哪些点的最短路程发生了变化呢？这里可以用一个队列来维护这些点，算法大致如下。