```
        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

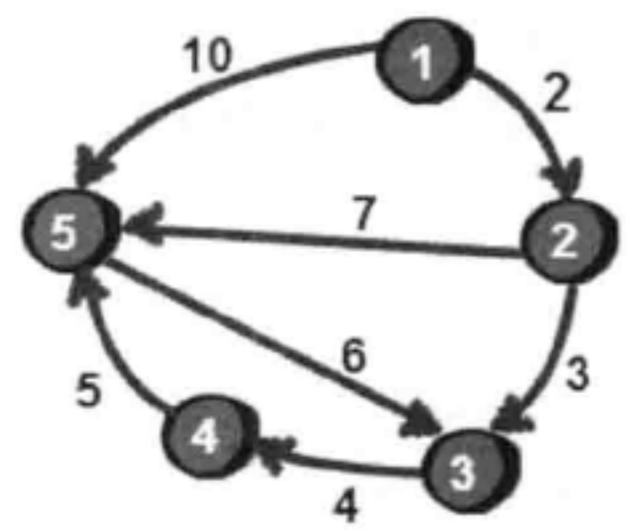
在上一节，我们提到 Bellman-Ford 算法的另一种优化：每次仅对最短路程发生变化了的点的相邻边执行松弛操作。但是如何知道当前哪些点的最短路程发生了变化呢？这里可以用一个队列来维护这些点，算法大致如下。

每次选取队首顶点 u，对顶点 u 的所有出边进行松弛操作。例如有一条 $u \rightarrow v$ 的边，如果通过 $u \rightarrow v$ 这条边使得源点到顶点 v 的最短路程变短（$dis[u]+e[u][v]<dis[v]$），且顶点 v 不在当前的队列中，就将顶点 v 放入队尾。需要注意的是，同一个顶点同时在队列中出现多次是毫无意义的，所以我们需要一个数组来判重（判断哪些点已经在队列中）。在对顶点 u 的所有出边松弛完毕后，就将顶点 v 出队。接下来不断从队列中取出新的队首顶点再进行如上操作，直至队列空为止。

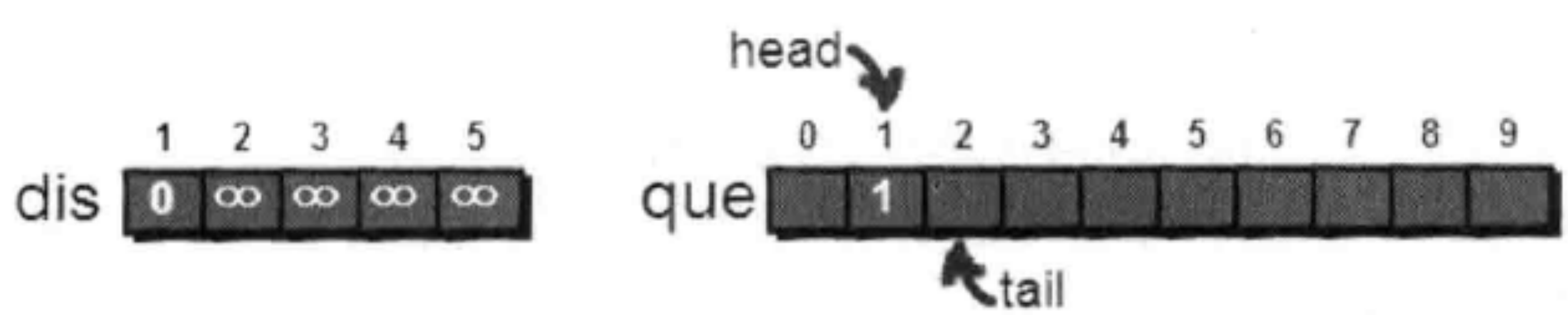
下面我们用一个具体的例子来详细讲解。

```
5 7
1 2 2
1 5 10
2 3 3
2 5 7
3 4 4
4 5 5
5 3 6
```

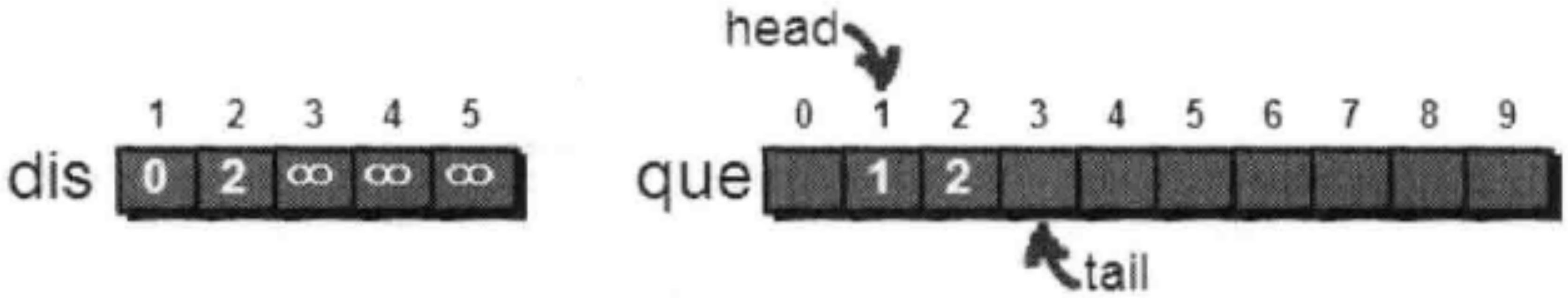
第一行两个整数 n m。n 表示顶点个数（顶点编号为 1~N），m 表示边的条数。接下来 m 行，每行有 3 个数 x y z。表示顶点 x 到顶点 y 的边权值为 z。



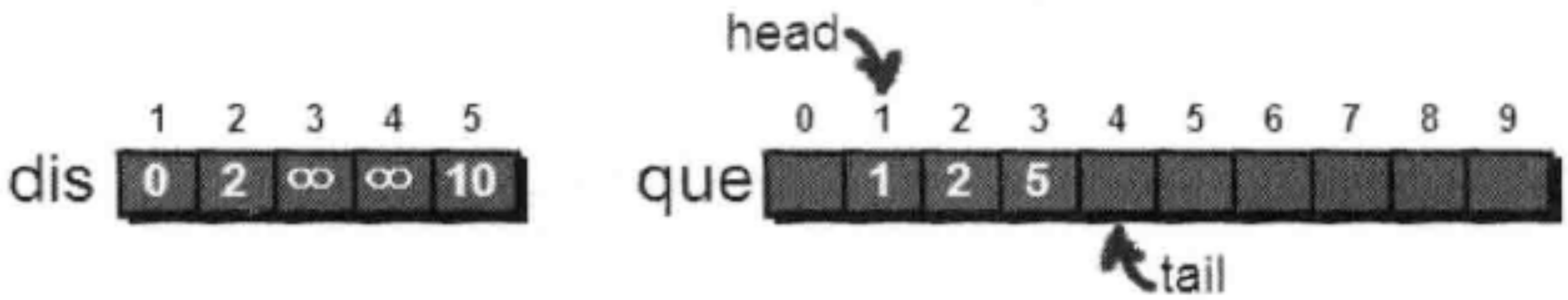
我们用数组 dis 来存放 1 号顶点到其余各个顶点的最短路径。初始时 dis[1]为 0，其余为无穷大。接下来将 1 号顶点入队。队列这里用一个数组 que 以及两个分别指向队列头和尾的变量 head 和 tail 来实现（队列的实现我们在第 2 章已经掌握）。



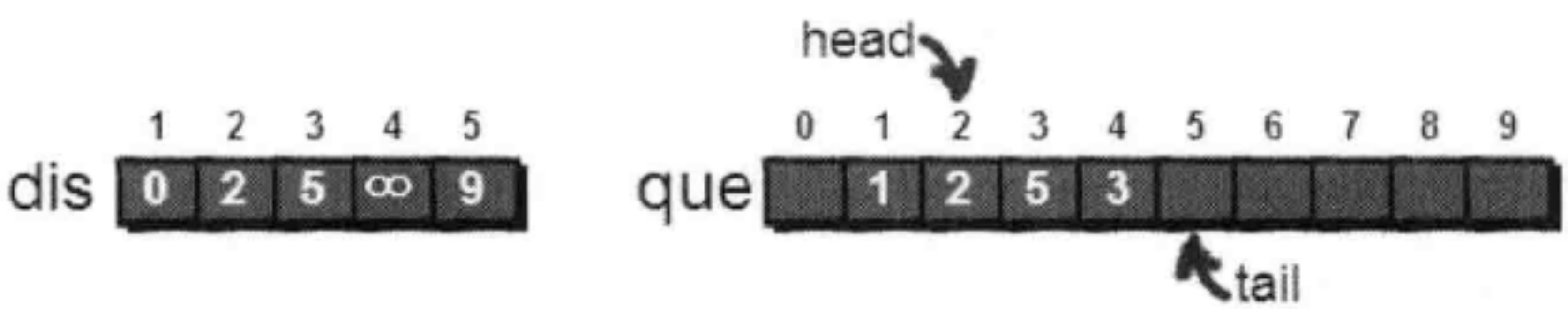
先来看当前队首 1 号顶点的边 1→2，看通过 1→2 能否让 1 号顶点到 2 号顶点的路程（即 dis[2]）变短，也就是说先来比较 dis[2]和 dis[1]+(1→2)的大小。dis[2]原来的值为∞，dis[1]+(1→2)的值为 2，因此松弛成功，dis[2]的值从∞更新为 2。并且当前 2 号顶点不在队列中，因此将 2 号顶点入队。



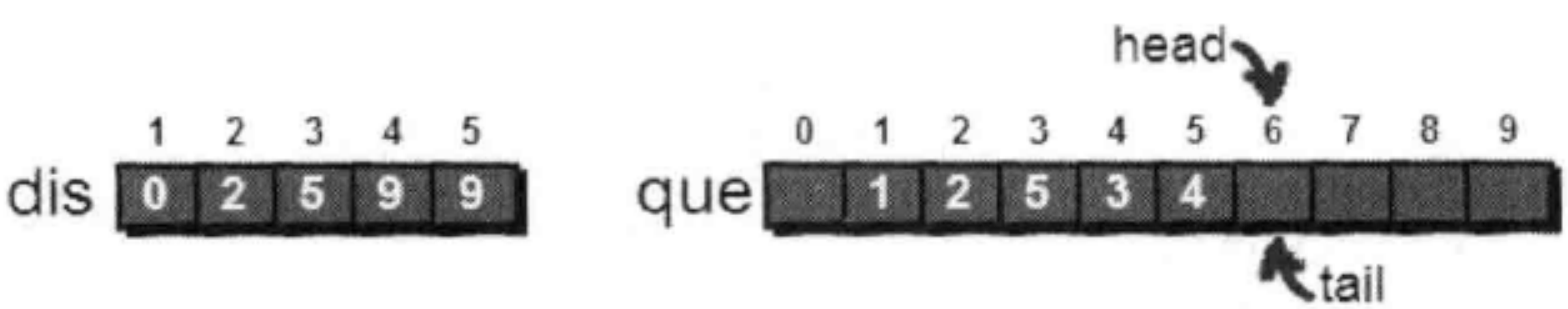
同样，对 1 号顶点剩余的出边进行如上操作，处理完毕后数组 dis 和队列 que 状态如下：



对 1 号顶点处理完毕后，就将 1 号顶点出队（head++即可），再对新队首 2 号顶点进行如上处理。在处理 2→5 这条边时需要特别注意一下，2→5 这条边虽然可以让 1 号顶点到 5 号顶点的路程变短（dis[5]的值从 10 更新为 9），但是 5 号顶点**已经**在队列中了，因此 5 号顶点**不能**再次入队。对 2 号顶点处理完毕后数组 dis 和队列 que 状态如下：



在对 2 号顶点处理完毕后，需要将 2 号顶点出队，并依次对剩下的顶点做相同的处理，直到队列为空为止。最终数组 dis 和队列 que 状态如下：



下面是代码实现，我们还是用邻接表来存储这个图，具体如下。

```
#include <stdio.h>
int main()
{
    int n,m,i,j,k;
    //u、v和w的数组大小要根据实际情况来设置，要比m的最大值要大1
    int u[8],v[8],w[8];
    //first要比n的最大值要大1，next要比m的最大值要大1
    int first[6],next[8];
    int dis[6]={0},book[6]={0};//book数组用来记录哪些顶点已经在队列中
    int que[101]={0},head=1,tail=1; //定义一个队列，并初始化队列
    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);

    //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;

    //初始化book数组，初始化为0，刚开始都不在队列中
    for(i=1;i<=n;i++)  book[i]=0;

    //初始化first数组下标1~n的值为-1，表示1~n顶点暂时都没有边
    for(i=1;i<=n;i++)  first[i]=-1;

    for(i=1;i<=m;i++)
    {
        //读入每一条边
        scanf("%d %d %d",&u[i],&v[i],&w[i]);
        //下面两句是建立邻接表的关键
        next[i]=first[u[i]];
        first[u[i]]=i;
    }

    //1号顶点入队
    que[tail]=1; tail++;
    book[1]=1;//标记1号顶点已经入队
```

```
    while(head<tail)//队列不为空的时候循环
    {
        k=first[que[head]];//当前需要处理的队首顶点
        while(k!=-1)//扫描当前顶点所有的边
        {
            if(dis[v[k]]>dis[u[k]]+w[k])//判断是否松弛成功
            {
                dis[v[k]]=dis[u[k]]+w[k];//更新顶点1到顶点v[k]的路程
                //这的book数组用来判断顶点v[k]是否在队列中
                //如果不使用一个数组来标记的话，判断一个顶点是否在队列中每次都需要从
队列的head到tail扫一遍，很浪费时间
                if(book[v[k]]==0)//0表示不在队列中，将顶点v[k]加入队列中
                {
                    //下面两句是入队操作
                    que[tail]=v[k];
                    tail++;
                    book[v[k]]=1;//同时标记顶点v[k]已经入队
                }
            }
            k=next[k];
        }
        //出队
        book[que[head]]=0;
        head++;
    }

    //输出1号顶点到其余各个顶点的最短路径
    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。

```
5 7
1 2 2
1 5 10
2 3 3
```

```
2 5 7
3 4 4
4 5 5
5 3 6
```

运行结果是：

```
0 2 5 9 9 5 3 6
```

下面来总结一下。初始时将源点加入队列。每次从队首（head）取出一个顶点，并对与其相邻的所有顶点进行松弛尝试，若某个相邻的顶点松弛成功，且这个相邻的顶点不在队列中（不在 head 和 tail 之间），则将它加入到队列中。对当前顶点处理完毕后立即出队，并对下一个新队首进行如上操作，直到队列为空时算法结束。这里用了一个数组 book 来记录每个顶点是否处在队列中。其实也可以不要 book 数组，检查一个顶点是否在队列中，只需要把 que[head]到 que[tail]依次判断一遍就可以了，但是这样做的时间复杂度是 $O(N)$，而使用 book 数组来记录的话时间复杂度会降至 $O(1)$。

使用队列优化的 Bellman-Ford 算法在形式上和广度优先搜索非常类似，不同的是在广度优先搜索的时候一个顶点出队后通常就不会再重新进入队列。而这里一个顶点很可能在出队列之后再次被放入队列，也就是当一个顶点的最短路程估计值变小后，需要对其所有出边进行松弛，但是如果这个顶点的最短路程估计值再次变小，仍需要对其所有出边再次进行松弛，这样才能保证相邻顶点的最短路程估计值同步更新。需要特别说明一下的是，使用队列优化的 Bellman-Ford 算法的时间复杂度在最坏情况下也是 $O(NM)$。通过队列优化的 Bellman-Ford 算法如何判断一个图是否有负环呢？如果某个点进入队列的次数超过 n 次，那么这个图则肯定存在负环。

用队列优化的 Bellman-Ford 算法的关键之处在于：只有那些在前一遍松弛中改变了最短路程估计值的顶点，才可能引起它们邻接点最短路程估计值发生改变。因此，用一个队列来存放被成功松弛的顶点，之后只对队列中的点进行处理，这就降低了算法的时间复杂度。另外说一下，西南交通大学段凡丁在 1994 年发表的关于最短路径的 SPFA 快速算法（SPFA，Shortest Path Faster Algorithm），也是基于队列优化的 Bellman-Ford 算法的。中国人在学习使用队列来优化 Bellman-Ford 算法时，段凡丁的这篇文章起到了不小的推广作用。令我感到很奇怪的是——为什么直到 1994 年才有人去发表这篇论文呢？

第 5 节　最短路径算法对比分析

	Floyd	Dijkstra	Bellman-Ford	队列优化的 Bellman-Ford
空间复杂度	$O(N^2)$	$O(M)$	$O(M)$	$O(M)$
时间复杂度	$O(N^3)$	$O((M+N)\log N)$	$O(NM)$	最坏也是 $O(NM)$
适用情况	稠密图 和顶点关系密切	稠密图 和顶点关系密切	稀疏图 和边关系密切	稀疏图 和边关系密切
负权	可以解决负权	不能解决负权	可以解决负权	可以解决负权

Floyd 算法虽然总体时间复杂度高，但是可以解决负权边，并且均摊到每一点对上，在所有的算法中还是属于较优的。另外，Floyd 算法较小的编码复杂度也是它的一大优势。所以，如果要求的是所有点对间的最短路径，或者如果数据范围较小，则 Floyd 算法比较适合。Dijkstra 算法最大的弊端是它无法适应有负权边的图。但是 Dijkstra 具有良好的可扩展性，扩展后可以适应很多问题。另外用堆优化的 Dijkstra 算法的时间复杂度可以达到 $O(M\log N)$。当边有负权时，需要使用 Bellman-Ford 算法或者队列优化的 Bellman-Ford 算法。因此我们选择最短路径算法时，要根据实际需求和每一种算法的特性，选择适合的算法。