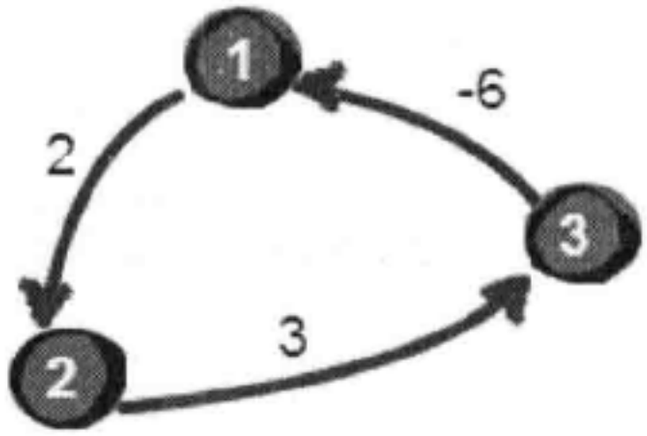
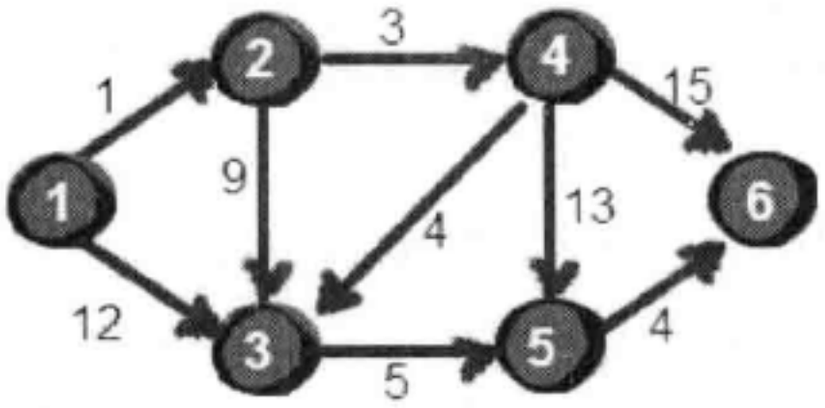


此算法由 Robert W. Floyd（罗伯特・弗洛伊德）于 1962 年发表在 *Communications of the ACM* 上。同年 Stephen Warshall（史蒂芬・沃舍尔）也独立发表了这个算法。Robert W．Floyd 这个牛人是朵奇葩，他原本在芝加哥大学读的文学，但是因为当时美国经济不太景气，找工作比较困难，无奈之下到西屋电气公司当了一名计算机操作员，在 IBM650 机房值夜班，并由此开始了他的计算机生涯。此外他还和 J.W.J. Williams（威廉姆斯）于 1964 年共同发明了著名的堆排序算法 HEAPSORT。堆排序算法我们将在第 7 章学习。Robert W. Floyd 在 1978 年获得了图灵奖。

第 2 节　Dijkstra 算法——通过边实现松弛

本节来学习指定一个点（源点）到其余各个顶点的最短路径，也叫做“单源最短路径”。例如求下图中的 1 号顶点到 2、3、4、5、6 号顶点的最短路径。



与 Floyd-Warshall 算法一样，这里仍然使用二维数组 e 来存储顶点之间边的关系，初始值如下。

e	1	2	3	4	5	6
1	0	1	12	∞	∞	∞
2	∞	0	9	3	∞	∞
3	∞	∞	0	∞	5	∞
4	∞	∞	4	0	13	15
5	∞	∞	∞	∞	0	4
6	∞	∞	∞	∞	∞	0

我们还需要用一个一维数组 dis 来存储 1 号顶点到其余各个顶点的初始路程，如下。

	1	2	3	4	5	6
dis	0	1	12	∞	∞	∞

我们将此时 dis 数组中的值称为最短路程的“估计值”。

既然是求 1 号顶点到其余各个顶点的最短路程，那就先找一个离 1 号顶点最近的顶点。通过数组 dis 可知当前离 1 号顶点最近的是 2 号顶点。当选择了 2 号顶点后，dis[2]的值就已经从“估计值”变为了“确定值”，即 1 号顶点到 2 号顶点的最短路程就是当前 dis[2]值。为什么呢？你想啊，目前离 1 号顶点最近的是 2 号顶点，并且这个图所有的边都是正数，那么肯定不可能通过第三个顶点中转，使得 1 号顶点到 2 号顶点的路程进一步缩短了。因为 1 号顶点到其他顶点的路程肯定没有 1 号到 2 号顶点短，对吧 O(∩_∩)O~

既然选了 2 号顶点，接下来再来看 2 号顶点有哪些出边呢。有 2→3 和 2→4 这两条边。先讨论通过 2→3 这条边能否让 1 号顶点到 3 号顶点的路程变短，也就是说现在来比较 dis[3]和 dis[2]+e[2][3]的大小。其中 dis[3]表示 1 号顶点到 3 号顶点的路程；dis[2]+e[2][3]中 dis[2]表示 1 号顶点到 2 号顶点的路程，e[2][3]表示 2→3 这条边。所以 dis[2]+e[2][3]就表示从 1 号顶点先到 2 号顶点，再通过 2→3 这条边，到达 3 号顶点的路程。

我们发现 dis[3]=12，dis[2]+e[2][3]=1+9=10，dis[3]>dis[2]+e[2][3]，因此 dis[3]要更新为 10。这个过程有个专业术语叫做“**松弛**”，1 号顶点到 3 号顶点的路程即 dis[3]，通过 2→3 这条边松弛成功。这便是 Dijkstra 算法的主要思想：通过“边”来松弛 1 号顶点到其余各个顶点的路程。

同理，通过 2→4（e[2][4]），可以将 dis[4]的值从∞松弛为 4（dis[4]初始为∞，dis[2]+e[2][4]=1+3=4，dis[4]>dis[2]+e[2][4]，因此 dis[4]要更新为 4）。

刚才我们对 2 号顶点所有的出边进行了松弛。松弛完毕之后 dis 数组为：

	1	2	3	4	5	6
dis	0	1	10	4	∞	∞

接下来，继续在剩下的 3、4、5 和 6 号顶点中，选出离 1 号顶点最近的顶点。通过上面更新过的 dis 数组，当前离 1 号顶点最近的是 4 号顶点。此时，dis[4]的值已经从“估计值”变为了“确定值”。下面继续对 4 号顶点的所有出边（4→3，4→5 和 4→6）用刚才的方法进行松弛。松弛完毕之后 dis 数组为：

	1	2	3	4	5	6
dis	0	1	8	4	17	19

继续在剩下的 3、5 和 6 号顶点中，选出离 1 号顶点最近的顶点，这次选择 3 号顶点。此时，dis[3]的值已经从“估计值”变为了“确定值”。对 3 号顶点的所有出边（3→5）进行松弛。松弛完毕之后 dis 数组为：

	1	2	3	4	5	6
dis	0	1	8	4	13	19

继续在剩下的 5 和 6 号顶点中，选出离 1 号顶点最近的顶点，这次选择 5 号顶点。此时，dis[5]的值已经从“估计值”变为了“确定值”。对 5 号顶点的所有出边（5→4）进行松弛。松弛完毕之后 dis 数组为：

	1	2	3	4	5	6
dis	0	1	8	4	13	17

最后对 6 号顶点的所有出边进行松弛。因为这个例子中 6 号顶点没有出边，因此不用处理。到此，dis 数组中所有的值都已经从“估计值”变为了“确定值”。

最终 dis 数组如下，这便是 1 号顶点到其余各个顶点的最短路径。

	1	2	3	4	5	6
dis	0	1	8	4	13	17

OK，现在来总结一下刚才的算法。算法的基本思想是：每次找到离源点（上面例子的源点就是 1 号顶点）最近的一个顶点，然后以该顶点为中心进行扩展，最终得到源点到其余所有点的最短路径。基本步骤如下：

1. 将所有的顶点分为两部分：已知最短路程的顶点集合 P 和未知最短路径的顶点集合 Q。最开始，已知最短路径的顶点集合 P 中只有源点一个顶点。我们这里用一个 book 数组来记录哪些点在集合 P 中。例如对于某个顶点 i，如果 book[i]为 1 则表示这个顶点在集合 P 中，如果 book[i]为 0 则表示这个顶点在集合 Q 中。
2. 设置源点 s 到自己的最短路径为 0 即 dis[s]=0。若存在有源点能直接到达的顶点 i，则把 dis[i]设为 e[s][i]。同时把所有其他（源点不能直接到达的）顶点的最短路径设为∞。
3. 在集合 Q 的所有顶点中选择一个离源点 s 最近的顶点 u（即 dis[u]最小）加入到集

合 P。并考察所有以点 *u* 为起点的边，对每一条边进行松弛操作。例如存在一条从 *u* 到 *v* 的边，那么可以通过将边 *u*→*v* 添加到尾部来拓展一条从 *s* 到 *v* 的路径，这条路径的长度是 dis[*u*]+e[*u*][*v*]。如果这个值比目前已知的 dis[*v*]的值要小，我们可以用新值来替代当前 dis[*v*]中的值。

4. 重复第 3 步，如果集合 Q 为空，算法结束。最终 dis 数组中的值就是源点到所有顶点的最短路径。

完整的 Dijkstra 算法代码如下：

```
#include <stdio.h>
int main()
{
    int e[10][10],dis[10],book[10],i,j,n,m,t1,t2,t3,u,v,min;
    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
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            if(i==j) e[i][j]=0;
              else e[i][j]=inf;

    //读入边
    for(i=1;i<=m;i++)
    {
        scanf("%d %d %d",&t1,&t2,&t3);
        e[t1][t2]=t3;
    }

    //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]=e[1][i];

    //book数组初始化
    for(i=1;i<=n;i++)
        book[i]=0;
    book[1]=1;
```

```
//Dijkstra算法核心语句
for(i=1;i<=n-1;i++)
{
    //找到离1号顶点最近的顶点
    min=inf;
    for(j=1;j<=n;j++)
    {
        if(book[j]==0 && dis[j]<min)
        {
            min=dis[j];
            u=j;
        }
    }
    book[u]=1;
    for(v=1;v<=n;v++)
    {
        if(e[u][v]<inf)
        {
            if(dis[v]>dis[u]+e[u][v])
                dis[v]=dis[u]+e[u][v];
        }
    }
}

//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 和 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 边的权值为 z。

```
6 9
1 2 1
1 3 12
2 3 9
```

```
2 4 3
3 5 5
4 3 4
4 5 13
4 6 15
5 6 4
```

运行结果是：

```
0 1 8 4 13 17
```

通过上面的代码我们可以看出，这个算法的时间复杂度是 $O(N^2)$。其中每次找到离 1 号顶点最近的顶点的时间复杂度是 $O(N)$，这里我们可以用“堆”（将在下一章学到）来优化，使得这一部分的时间复杂度降低到 $O(\log N)$。另外对于边数 M 少于 N^2 的稀疏图来说（我们把 M 远小于 N^2 的图称为稀疏图，而 M 相对较大的图称为稠密图），我们可以用**邻接表**（这是个神马东西？不要着急，待会再仔细讲解）来代替邻接矩阵，使得整个时间复杂度优化到 $O(M+N)\log N$。请注意！在最坏的情况下 M 就是 N^2，这样的话 $(M+N)\log N$ 要比 N^2 还要大。但是大多数情况下并不会有那么多边，因此 $(M+N)\log N$ 要比 N^2 小很多。

这里我们主要来讲解如何使用邻接表来存储一个图，先上数据。

```
4 5
1 4 9
2 4 6
1 2 5
4 3 8
1 3 7
```

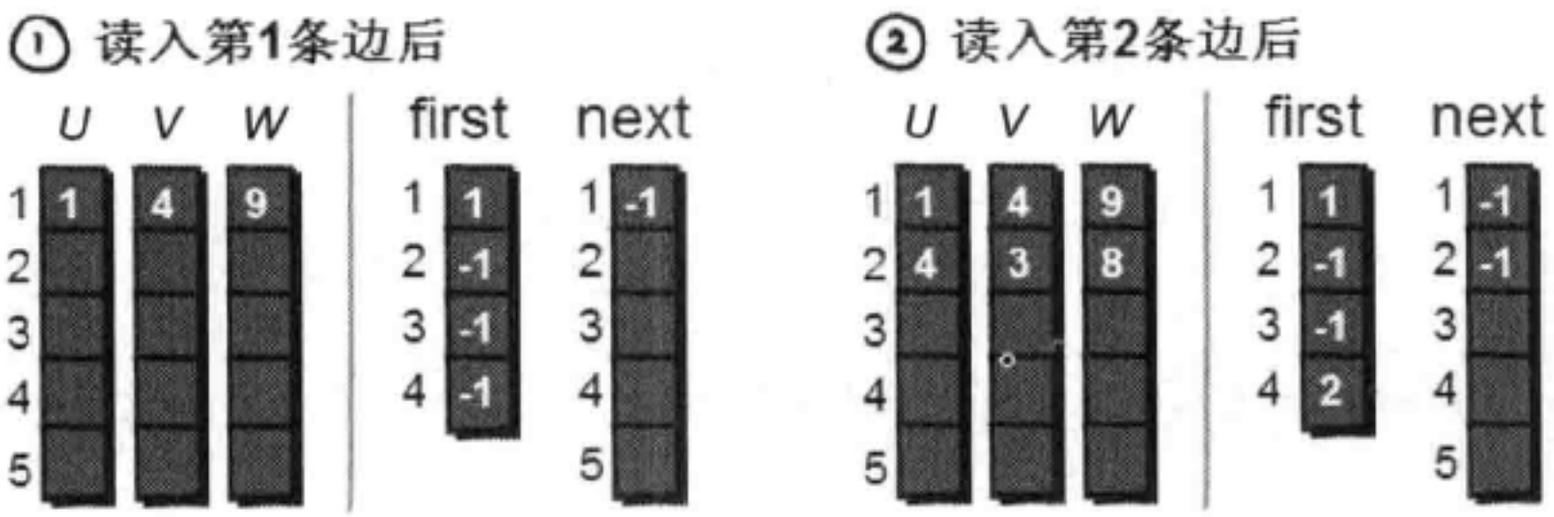
第一行两个整数 n m。n 表示顶点个数（顶点编号为 1~n），m 表示边的条数。接下来 m 行，每行有 3 个数 x y z。表示顶点 x 到顶点 y 的边的权值为 z。

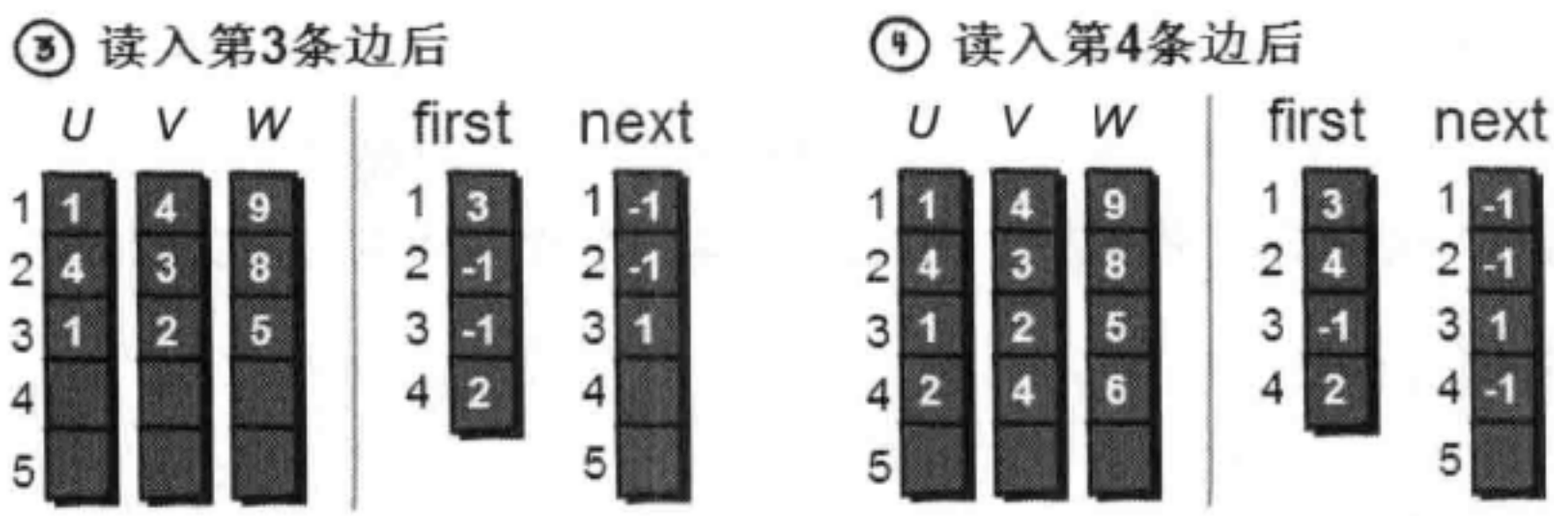
现在用邻接表来存储这个图，先给出代码如下。

```
int n,m,i;
//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[6],v[6],w[6];
//first和next的数组大小要根据实际情况来设置，要比n的最大值要大1
int first[5],next[5];
scanf("%d %d",&n,&m);
//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
```

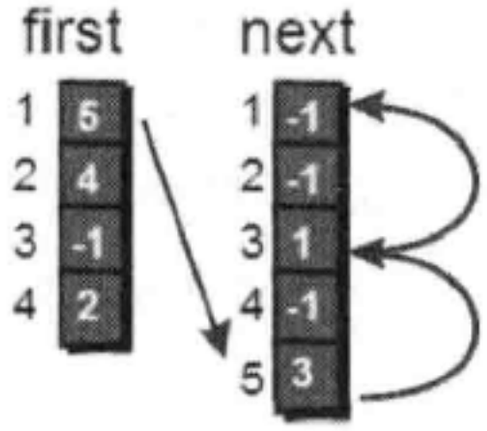
```
for(i=1;i<=n;i++)
    first[i]=-1;
for(i=1;i<=m;i++)
{
    scanf("%d %d %d",&u[i],&v[i],&w[i]);    //读入每一条边
    //下面两句是关键啦
    next[i]=first[u[i]];
    first[u[i]]=i;
}
```

这里为大家介绍的是使用数组来实现邻接表，而没有使用真正的指针链表，这是一种在实际应用中非常容易实现的方法。这种方法为每个顶点 i（i 从 1~n）都设置了一个链表，里面保存了从顶点 i 出发的所有的边（这里用 first 和 next 数组来实现，待会再来详细介绍）。首先我们需要为每一条边进行 1~m 的编号。用 u、v 和 w 三个数组来记录每条边的信息，即 $u[i]$、$v[i]$和 $w[i]$表示第 i 条边是从第 $u[i]$号顶点到 $v[i]$号顶点（$u[i]$→$v[i]$），且权值为 $w[i]$。first 数组的 1~n 号单元格分别用来存储 1~n 号顶点的第一条边的编号，初始的时候因为没有边加入所以都是-1。即 first[u[i]]保存顶点 u[i]的第一条边的编号，next[i]存储“编号为 i 的边”的“下一条边”的编号。





接下来如何遍历每一条边呢？我们之前说过其实 first 数组存储的就是每个顶点 i（i 从 1~n）的第一条边。比如 1 号顶点的第一条边是编号为 5 的边（1 3 7），2 号顶点的第一条边是编号为 4 的边（2 4 6），3 号顶点没有出向边，4 号顶点的第一条边是编号为 2 的边（2 4 6）。那么如何遍历 1 号顶点的每一条边呢？也很简单。请看下图：



在找到 1 号顶点的第一条边之后，剩下的边都可以在 next 数组中依次找到。

```
k=first[1];
while(k!=-1)
{
    printf("%d %d %d\n",u[k],v[k],w[k]);
    k=next[k];
}
```

细心的同学会发现，此时遍历某个顶点的边的时候的遍历顺序正好与读入时候的顺序相反。因为在为每个顶点插入边的时候都是直接插入“链表”的首部而不是尾部。不过这并不会产生任何问题，这正是这种方法的奇妙之处。遍历每个顶点的边，其代码如下。

```
for(i=1;i<=n;i++)
{
    k=first[i];
    while(k!=-1)
    {
```

```
            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];
```