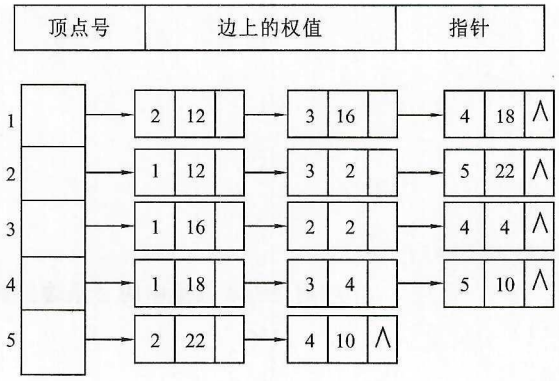

图 7-14　图的邻接表

答：该图 G 如图 7-15 所示，从顶点 1 出发，深度优先遍历序列为 1，2，3，4，5；广度优先遍历序列为 1，2，3，4，5；最小生成树如图 7-16 所示。

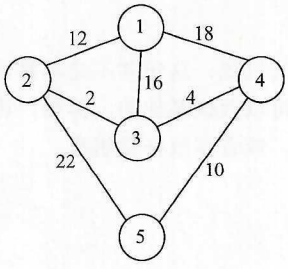

图 7-15　无向图

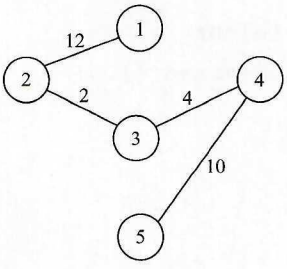

图 7-16　最小生成树

7.5　最短路径

7.5.1　迪杰斯特拉算法

通常采用迪杰斯特拉算法求图中**某一顶点到其余各顶点的最短路径**。

1．迪杰斯特拉算法思想

设有两个顶点集合 S 和 T，集合 S 中存放图中已找到最短路径的顶点，集合 T 存放图中剩余顶点。初始状态时，集合 S 中只包含源点 v_0，然后不断从集合 T 中选取到顶点 v_0 路径长度最短的顶点 v_u 并入到集合 S 中。**集合 S 每并入一个新的顶点 v_u，都要修改顶点 v_0 到集合 T 中顶点的最短路径长度值**。不断重复此过程，直到集合 T 的顶点全部并入到 S 中为止。

在理解“集合 S 每并入一个新的顶点 v_u，都要修改顶点 v_0 到集合 T 中顶点的最短路径长度值”的时候需要注意，在 v_u 被选入 S 中后，v_u 被确定为最短路径上的顶点，此时 v_u 就像 v_0 到达 T 中顶点的中转站，多了一个中转站，就会多一些到达 T 中顶点的新的路径，而这些新的路径有可能比之前 v_0 到 T 中顶点的路径要短，因此需要修改原有 v_0 到 T 中其他顶点的路径长度。此时对于 T 中的一个顶点 v_k，有两种情况：一种是 v_0 不经过 v_u 到达 v_k 的路径长度为 a（旧的路径长度），另一种是 v_0 经过 v_u 到达 v_k 的路径长度为 b（新的路径长度）。如果 a≤b，则什么也不做；如果 a>b，则用 b 来代替 a。用同样的方法处理 T 中其他顶点。当 T 中所有顶点都被处理完之后，会出现一组新的 v_0 到 T 中各顶点的路径，这些路径中有一条最短的，对应了 T 中一个顶点，就是新的 v_u，将其并入 S。重复上述过程，最后 T 中所有的顶点都会被并入 S 中，此时就可以得到 v_0 到图中所有顶点的最短路径。

2．迪杰斯特拉算法执行过程

引进 3 个辅助数组 dist[]、path[]和 set[]。

dist[v_i]表示当前已找到的从 v_0 到每个终点 v_i 的最短路径的长度。它的初态为：若从 v_0 到 v_i 有边，则 dist[v_i]为边上的权值，否则置 dist[v_i]为∞。

path[v_i]中保存从 v_0 到 v_i 最短路径上 v_i 的前一个顶点，假设最短路径上的顶点序列为 v_0，v_1，v_2，…，v_{i-1}，v_i，则 path[v_i]=v_{i-1}。path[]的初态为：如果 v_0 到 v_i 有边，则 path[v_i]=v_0，否则 path[v_i]=-1。

set[]为标记数组，set[v_i]=0 表示 v_i 在 T 中，即没有被并入最短路径；set[v_i]=1 表示 v_i 在 S 中，即已经被并入最短路径。set[]初态为：set[v_0]=1，其余元素全为 0。

迪杰斯特拉算法执行过程如下：

1）从当前 dist[]数组中选出最小值，假设为 dist[v_u]，将 set[v_u]设置为 1，表示当前新并入的顶点为 v_u。

2）循环扫描图中顶点，对每个顶点进行以下检测：

假设当前顶点为 v_j，检测 v_j 是否已经被并入 S 中，即看是否 set[v_j]=1。如果 set[v_j]=1，则什么都不做；如果 set[v_j]=0，则比较 dist[v_j]和 dist[v_u]+w 的大小，其中 w 为边<v_u，v_j>的权值。这个比较就是要看 v_0 经过旧的最短路径到达 v_j 和 v_0 经过含有 v_u 的新的最短路径到达 v_j 哪个更短，如果 dist[v_j]>dist[v_u]+w，则用新的路径长度来更新旧的，并把顶点 v_u 加入路径中，且作为路径上 v_j 之前的那个顶点，否则什么都不做。

3）对 1）和 2）循环执行 n-1 次（n 为图中顶点个数），即可得到 v_0 到其余所有顶点的最短路径。

迪杰斯特拉算法比较复杂，为便于理解，下面通过一个例子来体会一下用迪杰斯特拉算法求解最短路径的过程。

对图 7-17 所示的有向图，用迪杰斯特拉算法求从顶点 0 到其余各顶点最短路径的过程如下。

初始态：dist[0]置 0，dist[1]置 4，dist[2]置 6，dist[3]置 6，其余元素置∞。

path[1]置 0，path[2]置 0，path[3]置 0，其余元素置-1。

set[0]置 1，其余元素置 0。

假设图的邻接矩阵用二维数组 g[][]表示，则 g[i][j]为边<i，j>的权值。

1）从通往当前剩余顶点的路径中选出长度最短的，是 0→1，长度为 dist[1]=4，因此将顶点 1 并入最短路径中，set[1]置 1，结果如图 7-18 所示。

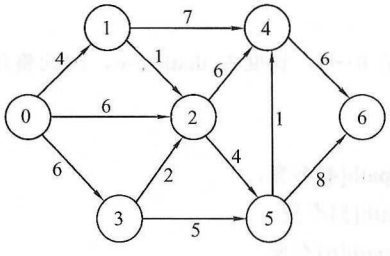

图 7-17 初始态

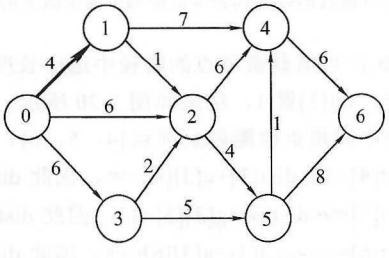

图 7-18 结果 1

以 1 为中间点检测剩余顶点{2，3，4，5，6}：

① dist[2]=6>dist[1]+g[1][2]=5，因此 dist[2]重置 5，path[2]重置 1。

② dist[3]=6<dist[1]+g[1][3]=∞，因此 dist[3]不变，path[3]不变。

③ dist[4]=∞>dist[1]+g[1][4]=11，因此 dist[4]重置 11，path[4]重置 1。

④ dist[5]=∞=dist[1]+g[1][5]=∞，因此 dist[5]不变，path[5]不变。

⑤ dist[6]=∞=dist[1]+g[1][6]=∞，因此 dist[6]不变，path[6]不变。

此时各数组值见表 7-1。

2）从通往当前剩余顶点的路径中选出长度最短的，是 0→1→2，长度为 dist[2]=5，因此将顶点 2 并入最短路径中，set[2]置 1，结果如图 7-19 所示。

以 2 为中间点检测剩余顶点{3，4，5，6}：

表 7-1　以 1 为中间点检测剩余顶点时各数组值

数组 \ 下标	0	1	2	3	4	5	6
dist[]	0	4	[5]	6	[11]	∞	∞
path[]	-1	0	[1]	0	[1]	-1	-1
set[]	1	[1]	0	0	0	0	0

注：表中方框内的数据代表本次检测中发生改变的数据。

① dist[3]=6<dist[2]+g[2][3]=∞，因此 dist[3]不变，path[3]不变。

② dist[4]=11=dist[2]+g[2][4]=11，因此 dist[4]不变，path[4]不变。

③ dist[5]=∞>dist[2]+g[2][5]=9，因此 dist[5]重置 9，path[5]重置 2。

④ dist[6]=∞=dist[2]+g[2][6]=∞，因此 dist[6]不变，path[6]不变。

此时各数组值见表 7-2。

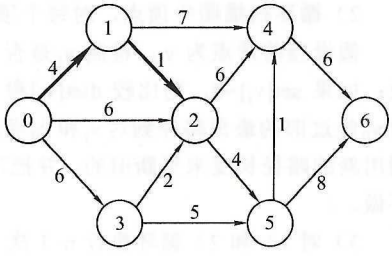

图 7-19　结果 2

表 7-2　以 2 为中间点检测剩余顶点时各数组值

数组 \ 下标	0	1	2	3	4	5	6
dist[]	0	4	5	6	11	[9]	∞
path[]	-1	0	1	0	1	[2]	-1
set[]	1	1	[1]	0	0	0	0

注：表中方框内的数据代表本次检测中发生改变的数据。

3）从通往当前剩余顶点的路径中选出长度最短的，是 0→3，长度为 dist[3]=6，因此将顶点 3 并入最短路径中，set[3]置 1，结果如图 7-20 所示。

以 3 为中间顶点检测剩余顶点{4，5，6}：

① dist[4]=11<dist[3]+g[3][4]=∞，因此 dist[4]不变，path[4]不变。

② dist[5]=9<dist[3]+g[3][5]=11，因此 dist[5]不变，path[5]不变。

③ dist[6]=∞=dist[3]+g[3][6]=∞，因此 dist[6]不变，path[6]不变。

此时各数组值见表 7-3。

表 7-3　以 3 为中间顶点检测剩余顶点时各数组值

数组 \ 下标	0	1	2	3	4	5	6
dist[]	0	4	5	6	11	9	∞
path[]	-1	0	1	0	1	2	-1
set[]	1	1	1	[1]	0	0	0

注：表中方框内的数据代表本次检测中发生改变的数据。

4）从通往当前剩余顶点的路径中选出长度最短的，是 0→1→2→5，长度为 dist[5]=9，因此将顶点 5 并入最短路径中，set[5]置 1，结果如图 7-21 所示。

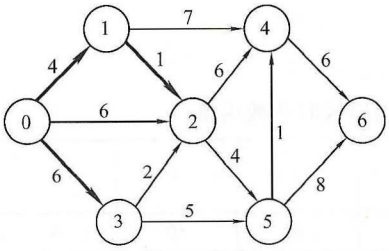

图 7-20 结果 3

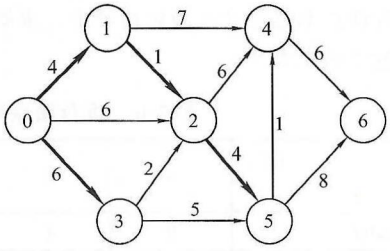

图 7-21 结果 4

以 5 为中间顶点检测剩余顶点{4，6}：

① dist[4]=11>dist[5]+g[5][4]=10，因此 dist[4]重置 10，path[4]重置 5。

② dist[6]=∞>dist[5]+g[5][6]=17，因此 dist[6]重置 17，path[6]重置 5。

此时各数组值见表 7-4。

表 7-4 以 5 为中间顶点检测剩余顶点时各数组值

数组 \ 下标	0	1	2	3	4	5	6
dist[]	0	4	5	6	[10]	9	[17]
path[]	-1	0	1	0	[5]	2	[5]
set[]	1	1	1	1	0	[1]	0

注：表中方框内的数据代表本次检测中发生改变的数据。

5）从通往当前剩余顶点的路径中选出长度最短的，是 0→1→2→5→4，长度为 dist[4]=10，因此将顶点 4 并入最短路径中，set[4]重置 1，结果如图 7-22 所示。

以 4 为中间顶点检测剩余顶点{6}：

dist[6]=17>dist[4]+g[4][6]=16，因此 dist[6]重置 16，path[6]重置 4。

此时各数组值见表 7-5。

表 7-5 以 4 为中间顶点检测剩余顶点时各数组值

数组 \ 下标	0	1	2	3	4	5	6
dist[]	0	4	5	6	10	9	[16]
path[]	-1	0	1	0	5	2	[4]
set[]	1	1	1	1	[1]	1	0

注：表中方框内的数据代表本次检测中发生改变的数据。

6）从通往当前剩余顶点的路径中选出长度最短的，是 0→1→2→5→4→6，长度为 dist[6]=16，因此将顶点 6 并入最短路径中，set[6]重置 1，结果如图 7-23 所示。

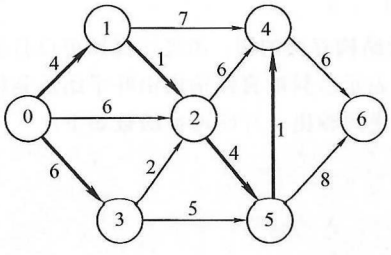

图 7-22 结果 5

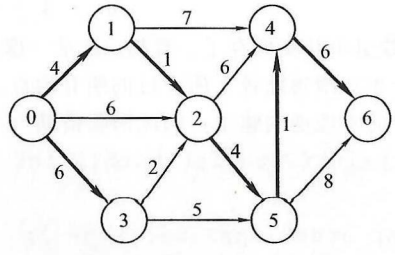

图 7-23 结果 6

此时所有顶点都已经并入最短路径中，求解过程结束。

此时各数组值见表 7-6。

表 7-6　所有顶点都并入最短路径时各数组值

数组＼下标	0	1	2	3	4	5	6
dist[]	0	4	5	6	10	9	16
path[]	-1	0	1	0	5	2	4
set[]	1	1	1	1	1	1	1

注：表中方框内的数据代表本次检测中发生改变的数据。

由表 7-6 可知：

顶点 0 到顶点 1 的最短路径为 0→1，长度为 4。

顶点 0 到顶点 2 的最短路径为 0→1→2，长度为 5。

顶点 0 到顶点 3 的最短路径为 0→3，长度为 6。

顶点 0 到顶点 4 的最短路径为 0→1→2→5→4，长度为 10。

顶点 0 到顶点 5 的最短路径为 0→1→2→5，长度为 9。

顶点 0 到顶点 6 的最短路径为 0→1→2→5→4→6，长度为 16。

以上就是用迪杰斯特拉算法求最短路径的标准过程，考生务必在理解的基础上熟练掌握，这是考研中的重点。考研中很容易出像上边例子中的手工求解最短路径的题目，过程写得很详细，是为了方便考生理解。如果考研中遇到这样的题目，答卷的时候不必写得这么烦琐，你可以根据自己的理解提取出要点，总结出应对这类题目的答题模板。对于以上例子，下面这种写法是一种适合作为答案的简写方法。

解：

已并入的顶点	剩余的顶点	dist[]	path[]
		0 1 2 3 4 5 6	0 1 2 3 4 5 6
0	1，2，3，4，5，6	0，4，6，6，∞，∞，∞	-1，0，0，0，-1，-1，-1
0，1	2，3，4，5，6	0，4，5，6，11，∞，∞	-1，0，1，0，1，-1，-1
0，1，2	3，4，5，6	0，4，5，6，11，9，∞	-1，0，1，0，1，2，-1
0，1，2，3	4，5，6	0，4，5，6，11，9，∞	-1，0，1，0，1，2，-1
0，1，2，3，5	4，6	0，4，5，6，10，9，17	-1，0，1，0，5，2，5
0，1，2，3，5，4	6	0，4，5，6，10，9，16	-1，0，1，0，5，2，4
0，1，2，3，5，4，6	无顶点剩余	0，4，5，6，10，9，16	-1，0，1，0，5，2，4

由上表可知，从顶点 0 到顶点 1～6 的最短路径长度分别为 4，5，6，10，9，16。

此时的 path[]数组为：

-1	0	1	0	5	2	4
0	1	2	3	4	5	6

path[]数组中其实保存了一棵树，这是一棵用**双亲存储结构**存储的树，通过这棵树可以打印出从源点到任何一个顶点最短路径上所经过的所有顶点。树的双亲表示法只能直接输出由叶子结点到根结点路径上的结点，而不能逆向输出，因此需要借助一个栈来实现逆向输出，打印路径函数如下：

```
void printfPath(int path[],int a)
{
    int stack[maxSize],top=-1;
    /*这个循环以由叶子结点到根结点的顺序将其入栈*/
```

```
    while(path[a]!=-1)
    {
        stack[++top]=a;
        a=path[a];
    }
    stack[++top]=a;
    while(top!=-1)
        cout<<stack[top--]<<" ";//出栈并打印出栈元素，实现了顶点的逆序打印
    cout<<endl;
}
```

由上述讲解可以写出如下迪杰斯特拉算法代码：

```
void Dijkstra(MGraph g,int v,int dist[],int path[])
{
    int set[maxSize];
    int min,i,j,u;
    /*从这句开始对各数组进行初始化*/
    for(i=0;i<g.n;++i)
    {
        dist[i]=g.edges[v][i];
        set[i]=0;
        if(g.edges[v][i]<INF)
            path[i]=v;
        else
            path[i]=-1;
    }
    set[v]=1;path[v]=-1;
    /*初始化结束*/
    /*关键操作开始*/
    for(i=0;i<g.n-1;++i)
    {
        min=INF;
        /*这个循环每次从剩余顶点中选出一个顶点，通往这个顶点的路径在通往所有剩余顶点的
        路径中是长度最短的*/
        for(j=0;j<g.n;++j)
            if(set[j]==0&&dist[j]<min)
            {
                u=j;
                min=dist[j];
            }
        set[u]=1;  //将选出的顶点并入最短路径中
        /*这个循环以刚并入的顶点作为中间点，对所有通往剩余顶点的路径进行检测*/
        for(j=0;j<g.n;++j)
        {
            /*这个 if 语句判断顶点 u 的加入是否会出现通往顶点 j 的更短的路径，如果出现，则
```

```
                改变原来路径及其长度，否则什么都不做*/
                if(set[j]==0&&dist[u]+g.edges[u][j]<dist[j])
                {
                    dist[j]=dist[u]+g.edges[u][j];
                    path[j]=u;
                }
            }
        }
        /*关键操作结束*/
    }
    /*函数结束时，dist[]数组中存放了 v 点到其余顶点的最短路径长度，path[]中存放 v 点到其余各顶点的最短路径*/
```

3．迪杰斯特拉算法时间复杂度分析

由算法代码可知，本算法主要部分为一个双重循环，外层循环内部有两个并列的单层循环，可以任取一个循环内的操作作为基本操作，基本操作执行的总次数即为双重循环执行的次数，为 n^2 次，因此本算法的时间复杂度为 $O(n^2)$。

7.5.2 弗洛伊德算法

迪杰斯特拉算法是求图中**某一顶点到其余各顶点的最短路径**，如果求图中**任意一对顶点间**的最短路径，则通常用弗洛伊德算法。

考试中涉及最多的是求用四阶方阵表示的图中每两点之间的最短路径的过程，方阵的阶数高了，计算量就比较大，考试中不会涉及太多。本算法的代码写起来要比迪杰斯特拉算法简单得多，因此在考试中，如果不对时间复杂度要求过于苛刻，尽量写本算法的代码，以减少错误。

对于本算法，需要掌握它的求解过程和程序代码，这两点比较简单，记住即可。对于考研要求，不需要钻研太多。下面就通过一个例子来总结用本算法求解最短路径的一般方法。

图 7-24 所示为一个有向图，各边的权值如图所示，用弗洛伊德算法求解其最短路径的过程如下。

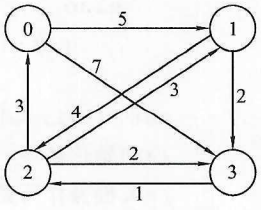

图 7-24 有向图

对于图 7-24，对应的邻接矩阵如下：

$$\begin{pmatrix} 0 & 5 & \infty & 7 \\ \infty & 0 & 4 & 2 \\ 3 & 3 & 0 & 2 \\ \infty & \infty & 1 & 0 \end{pmatrix}$$

初始时要设置两个矩阵 **A** 和 **Path**，**A** 用来记录当前已经求得的任意两个顶点最短路径的长度，**Path** 用来记录当前两顶点间最短路径上要经过的中间顶点。

1）初始时有：$\mathbf{A}_{-1}=\begin{pmatrix} 0 & 5 & \infty & 7 \\ \infty & 0 & 4 & 2 \\ 3 & 3 & 0 & 2 \\ \infty & \infty & 1 & 0 \end{pmatrix}$ $\mathbf{Path}_{-1}=\begin{pmatrix} -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \end{pmatrix}$（矩阵名的下标代表每一步中所选的中间顶点，图的顶点编号从 0 开始，初始的时候没有中间点，因此下标设为-1）。

2）以 0 为中间点，参照上一步矩阵中的结果，检测所有顶点对：{0，1}，{0，2}，{0，3}，{1，0}，{1，2}，{1，3}，{2，0}，{2，1}，{2，3}，{3，0}，{3，1}，{3，2}，假设当前所检测的顶点对为{i，j}，如果 A[i][j]>A[i][0]+A[0][j]，则将 A[i][j]改为 A[i][0]+A[0][j]的值，并且将 Path[i][j]改为 0。

经本次检测与修改，所得矩阵如下：