8.6.3 每对顶点之间的最短路径

问题描述：对于一个各边权值均大于零的有向图，对每一对顶点$i \neq j$，求出顶点i与顶点j之间的最短路径和最短路径长度。

多源最短路径问题： **Floyd算法**

1936~2001

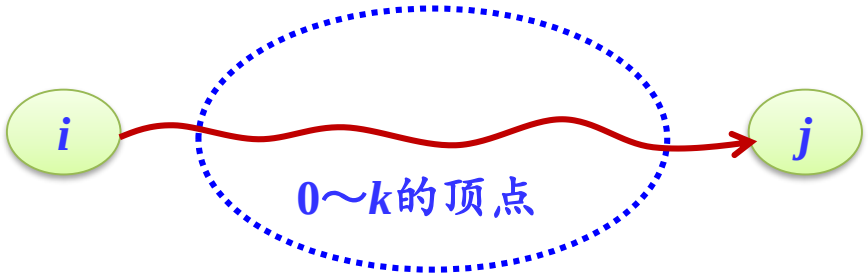
算法：迭代（递推）思路

假设有向图G=(V，E)采用邻接矩阵存储。设置一个二维数组A用于存放当前顶点之间的最短路径长度，分量$A[i][j]$表示当前顶点i ⇨ j的最短路径长度。

递推产生一个矩阵序列：

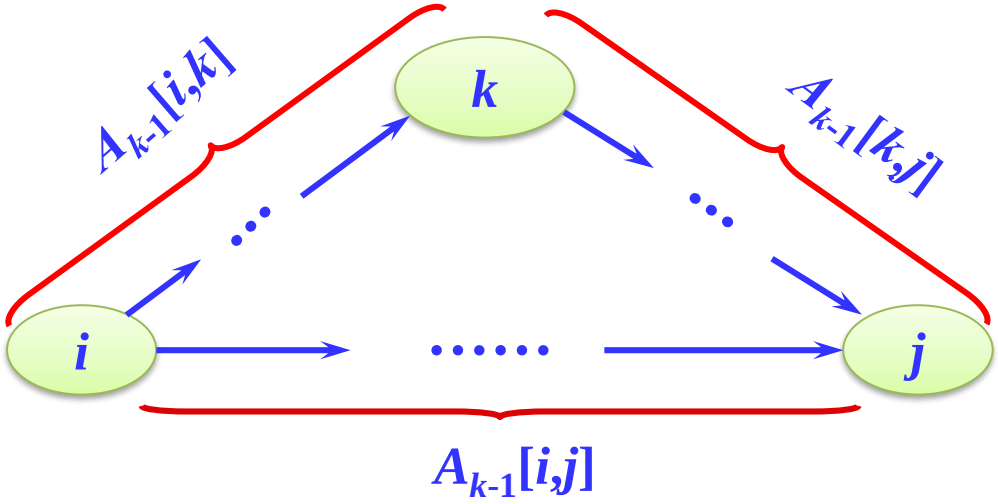
A_0 ⇨ A_1 ⇨ … ⇨ A_k ⇨ … ⇨ A_{n-1}。

$A_k[i][j]$：i ⇨ j的路径上所经过的顶点编号不大于k的最短路径长度。



❶ 初始时，有$A_{-1}[i][j]$=g.edges[i][j]。

❷ 考虑从i ⇨ j的最短路径经过编号为k顶点的情况：



$$A_k[i,j]=\text{MIN}\{\ A_{k-1}[i,j],\ \ A_{k-1}[i,k]+A_{k-1}[k,j]\ \}$$

$A_{-1}[i][j]$=g.edges[i][j]

$A_k[i,j]=\text{MIN}\{\ A_{k-1}[i,j],\ \ A_{k-1}[i,k]+A_{k-1}[k,j]\ \}$　$0 \leqslant k \leqslant n-1$

算法设计（解决2个问题）

（1）用二维数组A存储最短路径长度：

- $A_k[i][j]$表示考虑顶点0~k后得出的i ⇨ j的最短路径长度。
- $A_{n-1}[i][j]$表示最终的i ⇨ j的最短路径长度。

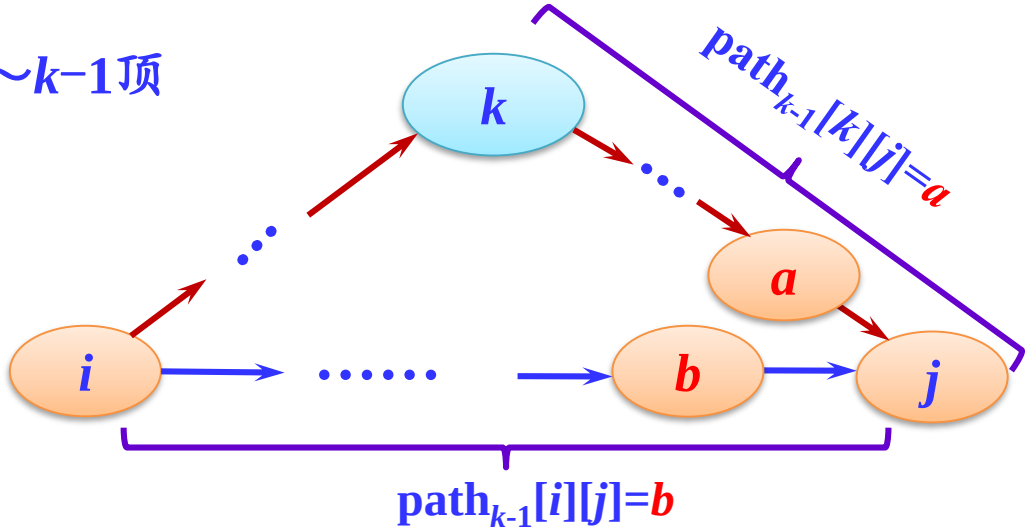
（2）用二维数组path存放最短路径：

- $path_k[i][j]$表示考虑顶点0~k后得出的i ⇨ j的最短路径。
- $path_{n-1}[i][j]$表示最终i ⇨ j的最短路径。

如何用path存放最短路径？

$path_x[i][j]$表示考虑过0～x的顶点得到i ⇨ j的最短路径，该路径上顶点j的前一个顶点。

❶ 已经考虑过0～$k-1$顶点的情况

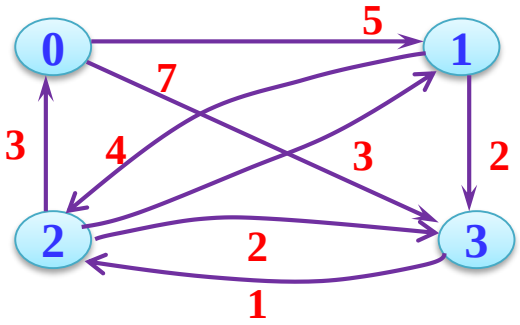


❷ 现在考虑顶点k

若经过顶点k的路径更短： ☞ $path_k[i][j] = a = path_{k-1}[k][j]$

否则： ☞ $path_k[i][j] = b = path_{k-1}[i][j]$ 不改变

Floyd算法示例演示



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

A_{-1}

	0	1	2	3
0	0	5	∞	7
1	∞	0	4	2
2	3	3	0	2
3	∞	∞	1	0

求path

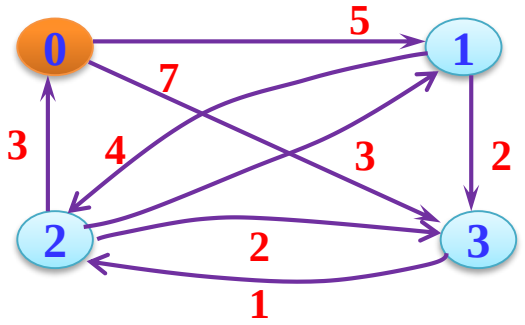
∞和i⇨i：−1

(i,j)有边：i

$path_{-1}$

	0	1	2	3
0	−1	0	−1	0
1	−1	−1	1	1
2	2	2	−1	2
3	−1	−1	3	−1

Floyd算法示例演示



考虑顶点0：

- 没有任何路径修改

$A_0=A_{-1}$，$path_0=path_{-1}$

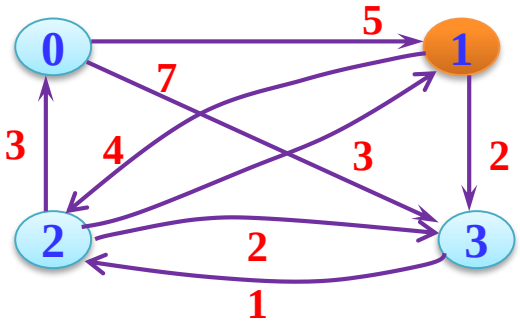
A_0

	0	1	2	3
0	0	5	∞	7
1	∞	0	4	2
2	3	3	0	2
3	∞	∞	1	0

$path_0$

	0	1	2	3
0	−1	0	−1	0
1	−1	−1	1	1
2	2	2	−1	2
3	−1	−1	3	−1

Floyd算法示例演示



考虑顶点1：

- 0→2：由无路径改为0→1 →2，长度为9，path[0][2]改为1

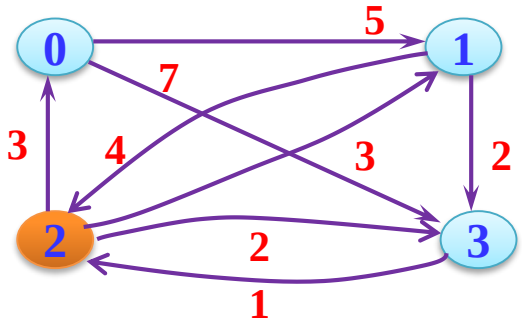
A_1

	0	1	2	3
0	0	5	9	7
1	∞	0	4	2
2	3	3	0	2
3	∞	∞	1	0

$path_1$

	0	1	2	3
0	−1	0	1	0
1	−1	−1	1	1
2	2	2	−1	2
3	−1	−1	3	−1

Floyd算法示例演示



考虑顶点2：

- 1→0：由无路径改为1→2 →0，长度为7，path[1][0]改为2
- 3→0：由无路径改为3→2 →0，长度为4，path[3][0]改为2
- 3→1：由无路径改为3→2 →1，长度为4，path[3][1]改为2

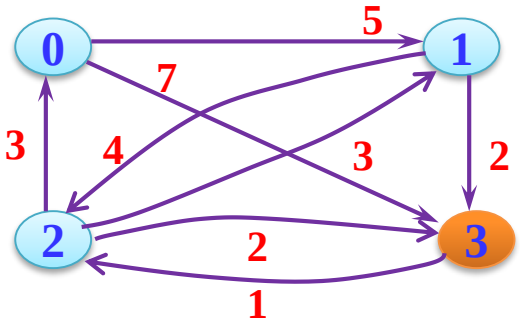
A_2

	0	1	2	3
0	0	5	9	7
1	7	0	4	2
2	3	3	0	2
3	4	4	1	0

$path_2$

	0	1	2	3
0	−1	0	1	0
1	2	−1	1	1
2	2	2	−1	2
3	2	2	3	−1

Floyd算法示例演示



考虑顶点3：

- 0→2：由0→1 →2改为0→3 →2，长度为8，path[0][2]改为3
- 1→0：由1→2 →0改为1→3 →2 →0，长度为6，path[1][0]改为2
- 1→2：由1→2改为1→3 →2，长度为3，path[1][2]改为3

A_3

	0	1	2	3
0	0	5	**8**	7
1	**6**	0	**3**	2
2	3	3	0	2
3	4	4	1	0

$path_3$

	0	1	2	3
0	−1	0	**3**	0
1	**2**	−1	**3**	1
2	2	2	−1	2
3	2	2	3	−1

求最终结果

A_3

	0	1	2	3
0	0	5	8	7
1	6	0	3	2
2	7	3	0	2
3	4	4	1	0

$path_3$

	0	1	2	3
0	−1	0	3	0
1	2	−1	3	1
2	2	2	−1	2
3	2	2	3	−1

❶ 求最短路径长度：

由A_3数组可以直接得到两个顶点之间的最短路径长度。

如$A_3[1][0]=6$

说明顶点1到0的最短路径长度为6。

A_3

	0	1	2	3
0	0	5	8	7
1	6	0	3	2
2	3	3	0	2
3	4	4	1	0

$path_3$

	0	1	2	3
0	−1	0	3	0
1	2	−1	3	1
2	2	2	−1	2
3	2	2	3	−1

❷ 求最短路径：

求顶点1 ⇨ 0的最短路径：

$path_3[1][0]=2$

$path_3[1][2]=3$

$path_3[1][3]=1$

顶点序列为0、2、3、1，则顶点1 ⇨0的最短路径为1→3→2→0。

弗洛伊德算法如下：

```
void Floyd(MatGraph g)                    //求每对顶点之间的最短路径
{    int A[MAXVEX][MAXVEX];               //建立A数组
     int path[MAXVEX][MAXVEX];            //建立path数组
     int i, j, k;
     for (i=0;i<g.n;i++)
         for (j=0;j<g.n;j++)
         {       A[i][j]=g.edges[i][j];
                 if (i!=j && g.edges[i][j]<INF)
                       path[i][j]=i;              //i和j顶点之间有一条边时
                 else                             //i和j顶点之间没有一条边时
                       path[i][j]=-1;
          }
```

*A*和path数组初始化

```
    for (k=0;k<g.n;k++)                          //求A_k[i][j]
    {   for (i=0;i<g.n;i++)
            for (j=0;j<g.n;j++)
                if (A[i][j]>A[i][k]+A[k][j])     //找到更短路径
                {     A[i][j]=A[i][k]+A[k][j];   //修改路径长度
                      path[i][j]=path[k][j];     //修改最短路径为经过顶点k
                }
    }
}
```

调整

本算法的时间复杂度为$O(n^3)$。

思考题

求所有顶点之间的最短路径可以对每个顶点调用一次Dijkstra算法，总共调用n次即可，其时间复杂度为$O(n^3)$。

而Floyd算法的时间复杂度也为$O(n^3)$。两者有什么不同？

数据结构经典算法的启示

用Dijkstra求所有顶点之间的最短路径

共享前面路径比较所得到的信息*A*

Floyd算法

——本讲完——