

# 数据结构与算法（14）：最短路算法

最短路径问题是图论研究中的一个经典算法问题，旨在寻找图（由结点和路径组成的）中两结点之间的最短路径。算法具体的形式包括：

- 确定起点的最短路径问题 - 即已知起始结点，求最短路径的问题。适合使用Dijkstra算法。
- 确定终点的最短路径问题 - 与确定起点的问题相反，该问题是已知终结结点，求最短路径的问题。在无向图中该问题与确定起点的问题完全等同，在有向图中该问题等同于把所有路径方向反转的确定起点的问题。
- 确定起点终点的最短路径问题 - 即已知起点和终点，求两结点之间的最短路径。
- 全局最短路径问题 - 求图中所有的最短路径。适合使用Floyd-Warshall算法。

## 一、Dijkstra算法

### 1.1 算法思想

Dijkstra(迪杰斯特拉)算法是典型的单源最短路径算法，用于计算一个节点到其他所有节点的最短路径。主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。Dijkstra算法是很有代表性的最短路径算法，在很多专业课程中都作为基本内容有详细的介绍，如数据结构，图论，运筹学等等。注意该算法要求图中不存在负权边。

问题描述：

在无向图  $G=(V,E)$  中，假设每条边  $E[i]$  的长度为  $w[i]$ ，找到由顶点  $V_0$  到其余各点的最短路径。（单源最短路径）

算法思想：

设 $G=(V,E)$ 是一个带权有向图，把图中顶点集合 $V$ 分成两组，第一组为已求出最短路径的顶点集合（用 $S$ 表示，初始时 $S$ 中只有一个源点，以后每求得一条最短路径，就将加入到集合 $S$ 中，直到全部顶点都加入到 $S$ 中，算法就结束了），第二组为其余未确定最短路径的顶点集合（用 $U$ 表示），按最短路径长度的递增次序依次把第二组的顶点加入 $S$ 中。在加入的过程中，总保持从源点 $v$ 到 $S$ 中各顶点的最短路径长度不大于从源点 $v$ 到 $U$ 中任何顶点的最短路径长度。此外，每个顶点对应一个距离， $S$ 中的顶点的距离就是从 $v$ 到此顶点的最短路径长度， $U$ 中的顶点的距离，是从 $v$ 到此顶点只包括 $S$ 中的顶点为中间顶点的当前最短路径长度。

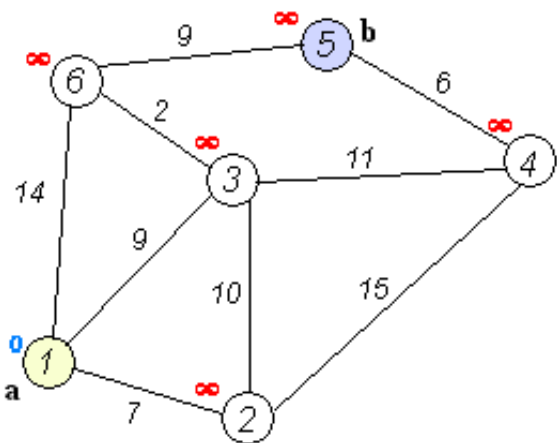
适用条件与限制

- 有向图和无向图都可以使用本算法，无向图中的每条边可以看成相反的两条边。
- 用来求最短路的图中不能存在负权边。(可以利用拓扑排序检测)

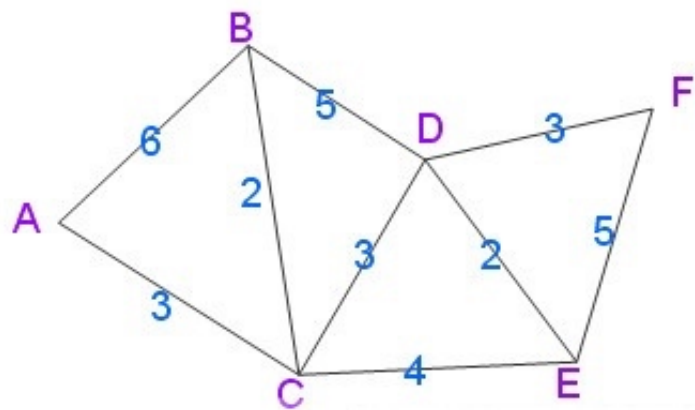
## 1.2 算法步骤

1. 初始时，S只包含源点，即 $S = \{v\}$ ，v的距离为0。U包含除v外的其他顶点，即： $U = \{\text{其余顶点}\}$ ，若v与U中顶点u有边，则正常有权值，若u不是v的出边邻接点，则权值为 $\infty$ 。
2. 从U中选取一个距离v最小的顶点k，把k，加入S中（该选定的距离就是v到k的最短路径长度）。
3. 以k为新考虑的中间点，修改U中各顶点的距离；若从源点v到顶点u的距离（经过顶点k）比原来距离（不经过顶点k）短，则修改顶点u的距离值，修改后的距离值的顶点k的距离加上边上的权。
4. 重复步骤b和c直到所有顶点都包含在S中。

步骤动画如下：



实例如下



用Dijkstra算法找出以A为起点的单源最短路径步骤如下

步骤	S 集合中	U 集合中
1	选入 A, 此时 S = <A> 此时最短路径 A → A = 0 以 A 为中间点, 从 A 开始找	U = <B、C、D、E、F> A → B = 6 A → C = 3 A → 其他 U 中的顶点 = ∞ 发现 A → C = 3 权值为最短
2	选入 C, 此时 S = <A、C> 此时最短路径 A → A = 0, A → C = 3 以 C 为中间点, 从 A → C = 3 这条最短路径开始找	U = <B、D、E、F> A → C → B = 5 (比上面第一步的 A → B = 6 要短) 此时到 B 权值为 A → C → B = 5 A → C → D = 6 A → C → E = 7 A → C → 其他 U 中的顶点 = ∞ 发现 A → C → B = 5 权值为最短
3	选入 B, 此时 S = <A、C、B> 此时最短路径 A → A = 0, A → C = 3, A → C → B = 5 以 B 为中间点, 从 A → C → B = 5 这条最短路径开始找	U = <D、E、F> A → C → B → D = 10 (比上面第二步的 A → C → D = 6 要长) 此时到 D 权值更改为 A → C → D = 6 A → C → B → 其他 U 中的顶点 = ∞ 发现 A → C → D = 6 权值为最短
4	选入 D, 此时 S = <A、C、B、D> 此时最短路径 A → A = 0, A → C = 3, A → C → B = 5, A → C → D = 6 以 D 为中间点, 从 A → C → D 这条最短路径开始找	U = <E、F> A → C → D → E = 8 (比上面第二步的 A → C → E = 7 要长) 此时到 E 权值更改为 A → C → E = 7 A → C → D → F = 9 发现 A → C → E = 7 权值为最短
5	选入 E, 此时 S = <A、C、B、D、E> 此时最短路径 A → A = 0, A → C = 3, A → C → B = 5, A → C → D = 6, A → C → E = 7 以 E 为中间点, 从 A → C → E = 7 这条最短路径开始找	U = <F> A → C → E → F = 12 (比上面第四步的 A → C → D → F = 9 要长) 此时到 F 权值更改为 A → C → D → F = 9 发现 A → C → D → F = 9 权值为最短
6	选入 F, 此时 S = <A、C、B、D、E、F> 此时最短路径 A → A = 0, A → C = 3, A → C → B = 5, A → C → D = 6, A → C → E = 7, A → C → D → F = 9	U 集合已空, 查找完毕。

## 1.3 代码实现

以"邻接矩阵"为例对迪杰斯特拉算法进行说明。

```

public class MatrixUDG {

    private int mEdgNum;           // 边的数量
    private char[] mVexs;          // 顶点集合
    private int[][] mMatrix;       // 邻接矩阵
    private static final int INF = Integer.MAX_VALUE; // 最大值

    ...

}

```

MatrixUDG是邻接矩阵对应的结构体。mVexs用于保存顶点，mEdgNum用于保存边数，mMatrix则是用于保存矩阵信息的二维数组。例如，mMatrix[i][j]=1，则表示"顶点i(即

mVexs[i]"和"顶点j(即mVexs[j])"是邻接点；mMatrix[i][j]=0，则表示它们不是邻接点。

```
/*
 * Dijkstra最短路径。
 * 即，统计图中"顶点vs"到其它各个顶点的最短路径。
 *
 * 参数说明：
 *     vs -- 起始顶点(start vertex)。即计算"顶点vs"到其它顶点的最短路径。
 *     prev -- 前驱顶点数组。即，prev[i]的值是"顶点vs"到"顶点i"的最短路径所经历的全部顶点
 * 中，位于"顶点i"之前的那个顶点。
 *     dist -- 长度数组。即，dist[i]是"顶点vs"到"顶点i"的最短路径的长度。
 */
public void dijkstra(int vs, int[] prev, int[] dist) {
    // flag[i]=true表示"顶点vs"到"顶点i"的最短路径已成功获取
    boolean[] flag = new boolean[mVexs.length];

    // 初始化
    for (int i = 0; i < mVexs.length; i++) {
        flag[i] = false;           // 顶点i的最短路径还没获取到。
        prev[i] = 0;               // 顶点i的前驱顶点为0。
        dist[i] = mMatrix[vs][i]; // 顶点i的最短路径为"顶点vs"到"顶点i"的权。
    }

    // 对"顶点vs"自身进行初始化
    flag[vs] = true;
    dist[vs] = 0;

    // 遍历mVexs.length-1次；每次找出一个顶点的最短路径。
    int k=0;
    for (int i = 1; i < mVexs.length; i++) {
        // 寻找当前最小的路径；
        // 即，在未获取最短路径的顶点中，找到离vs最近的顶点(k)。
        int min = INF;
        for (int j = 0; j < mVexs.length; j++) {
            if (flag[j]==false && dist[j]<min) {
                min = dist[j];
                k = j;
            }
        }
        // 标记"顶点k"为已经获取到最短路径
        flag[k] = true;

        // 修正当前最短路径和前驱顶点
        // 即，当已经"顶点k的最短路径"之后，更新"未获取最短路径的顶点的最短路径和前驱顶点"
        for (int j = 0; j < mVexs.length; j++) {
            int tmp = (mMatrix[k][j]==INF ? INF : (min + mMatrix[k][j]));
        }
    }
}
```

```

        if (flag[j]==false && (tmp<dist[j])) {
            dist[j] = tmp;
            prev[j] = k;
        }
    }
}

// 打印dijkstra最短路径的结果
System.out.printf("dijkstra(%c): \n", mVexs[vs]);
for (int i=0; i < mVexs.length; i++)
    System.out.printf("    shortest(%c, %c)=%d\n", mVexs[vs], mVexs[i], dist[i]);
}

```

## 1.4 时间复杂度

我们可以用大O符号将该算法的运行时间表示为边数 $m$ 和顶点数 $n$ 的函数。

对于基于顶点集 $Q$ 的实现，算法的运行时间是 $O(|E| \cdot dk_Q + |V| \cdot em_Q)$ ，其中 $dk_Q$ 和 $em_Q$ 分别表示完成键的降序排列时间和从 $Q$ 中提取最小键值的时间。

Dijkstra算法最简单的实现方法是用一个链表或者数组来存储所有顶点的集合 $Q$ ，所以搜索 $Q$ 中最小元素的运算（Extract-Min( $Q$ ））只需要线性搜索  $Q$ 中的所有元素。这样的话算法的运行时间是 $O(n^2)$ 。

对于边数少于 $n^2$ 的稀疏图来说，我们可以用邻接表来更有效的实现该算法。同时需要将一个二叉堆或者斐波纳契堆用作优先队列来寻找最小的顶点（Extract-Min）。当用到二叉堆的时候，算法所需的时间为 $O((m + n)\log n)$ ，斐波纳契堆能稍微提高一些性能，让算法运行时间达到 $O(m + n\log n)$ 。然而，使用斐波纳契堆进行编程，常常会由于算法常数过大而导致速度没有显著提高。

## 二、Floyd算法

Dijkstra很优秀，但是使用Dijkstra有一个最大的限制，就是不能有负权边。而Bellman-Ford适用于权值可以为负、无权值为负的回路的图。这比Dijkstra算法的使用范围要广。

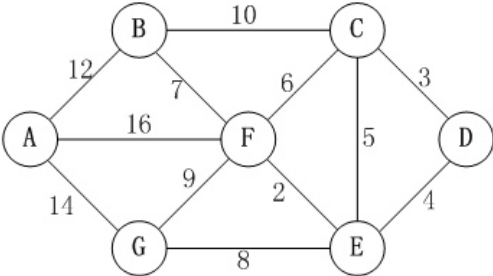
### 2.1 算法思想

Floyd算法是一个经典的动态规划算法。用通俗的语言来描述的话，首先我们的目标是寻找从点 $i$ 到点 $j$ 的最短路径。从动态规划的角度看问题，我们需要为这个目标重新做一个诠释（这个诠释正是动态规划最富创造力的精华所在）

从任意节点*i*到任意节点*j*的最短路径不外乎2种可能，1是直接从*i*到*j*，2是从*i*经过若干个节点*k*到*j*。所以，我们假设Dis(*i*,*j*)为节点*u*到节点*v*的最短路径的距离，对于每一个节点*k*，我们检查Dis(*i*,*k*) + Dis(*k*,*j*) < Dis(*i*,*j*)是否成立，如果成立，证明从*i*到*k*再到*j*的路径比*i*直接到*j*的路径短，我们便设置Dis(*i*,*j*) = Dis(*i*,*k*) + Dis(*k*,*j*)，这样一来，当我们遍历完所有节点*k*，Dis(*i*,*j*)中记录的便是*i*到的最短路径的距离。

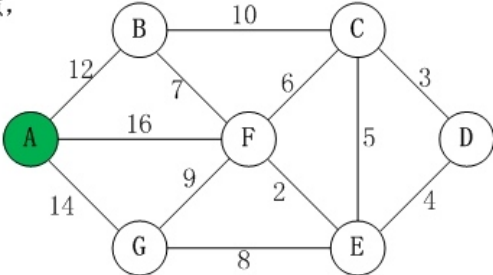
## 2.2 算法步骤

第1步：  
初始化矩阵S



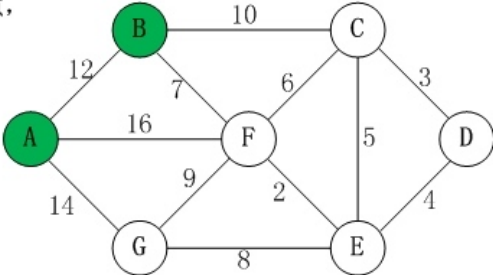
	A	B	C	D	E	F	G
A	0	12	INF	INF	INF	16	14
B	12	0	10	INF	INF	7	INF
C	INF	10	0	3	5	6	INF
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	INF	INF	INF	8	9	0

第2步：  
以顶点A为中介点，  
更新矩阵S。



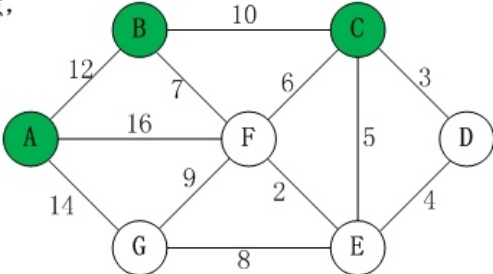
	A	B	C	D	E	F	G
A	0	12	INF	INF	INF	16	14
B	12	0	10	INF	INF	7	26
C	INF	10	0	3	5	6	INF
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	26	INF	INF	8	9	0

第3步：  
以顶点B为中介点，  
更新矩阵S。



	A	B	C	D	E	F	G
A	0	12	22	INF	INF	16	14
B	12	0	10	INF	INF	7	26
C	22	10	0	3	5	6	36
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	26	36	INF	8	9	0

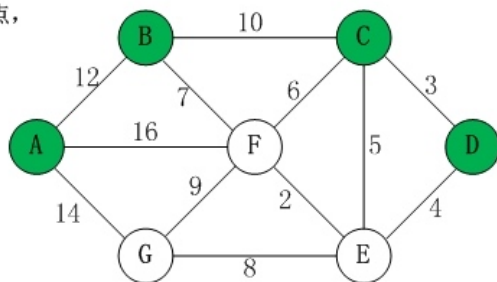
第4步：  
以顶点C为中介点，  
更新矩阵S。



	A	B	C	D	E	F	G
A	0	12	22	25	27	16	14
B	12	0	10	13	15	7	26
C	22	10	0	3	5	6	36
D	25	13	3	0	4	9	39
E	27	15	5	4	0	2	8
F	16	7	6	9	2	0	9
G	14	26	36	39	8	9	0

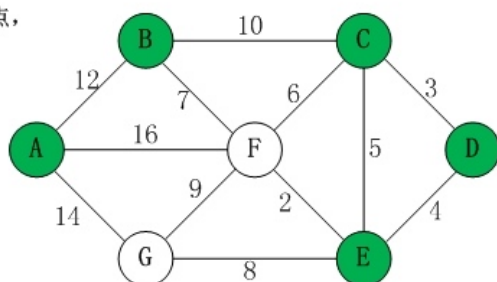


第5步：  
以顶点D为中介点，  
更新矩阵S。



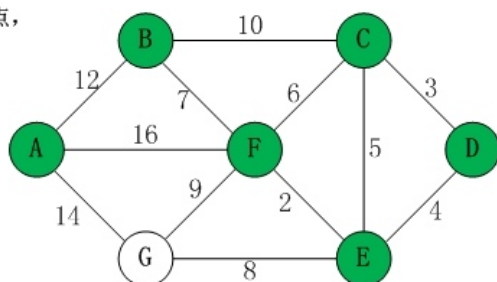
	A	B	C	D	E	F	G
A	0	12	22	25	27	16	14
B	12	0	10	13	15	7	26
C	22	10	0	3	5	6	36
D	25	13	3	0	4	9	39
E	27	15	5	4	0	2	8
F	16	7	6	9	2	0	9
G	14	26	36	39	8	9	0

第6步：  
以顶点E为中介点，  
更新矩阵S。



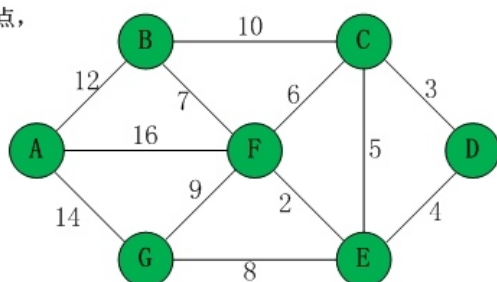
	A	B	C	D	E	F	G
A	0	12	22	25	27	16	14
B	12	0	10	13	15	7	23
C	22	10	0	3	5	6	13
D	25	13	3	0	4	6	12
E	27	15	5	4	0	2	8
F	16	7	6	6	2	0	9
G	14	23	13	12	8	9	0

第7步：  
以顶点F为中介点，  
更新矩阵S。



	A	B	C	D	E	F	G
A	0	12	22	22	18	16	14
B	12	0	10	13	9	7	16
C	22	10	0	3	5	6	13
D	22	13	3	0	4	6	12
E	18	9	5	4	0	2	8
F	16	7	6	6	2	0	9
G	14	16	13	12	8	9	0

第8步：  
以顶点G为中介点，  
更新矩阵S。



	A	B	C	D	E	F	G
A	0	12	22	22	18	16	14
B	12	0	10	13	9	7	16
C	22	10	0	3	5	6	13
D	22	13	3	0	4	6	12
E	18	9	5	4	0	2	8
F	16	7	6	6	2	0	9
G	14	16	13	12	8	9	0

初始状态：S是记录各个顶点间最短路径的矩阵。

1. 初始化S：矩阵S中顶点 $a[i][j]$ 的距离为顶点i到顶点j的权值；如果i和j不相邻，则 $a[i][j]=\infty$ 。实际上，就是将图的原始矩阵复制到S中。

注： $a[i][j]$ 表示矩阵S中顶点i(第i个顶点)到顶点j(第j个顶点)的距离。

2. 以顶点A(第1个顶点)为中介点，若 $a[i][j] > a[i][0] + a[0][j]$ ，则设置 $a[i][j] = a[i][0] + a[0][j]$ 。以顶点A[1]，上一步操作之后， $a[1][6] = \infty$ ；而将A作为中介点时， $(B,A)=12$ ， $(A,G)=14$ ，因此B和G之间的距离可以更新为26。

3. 同理，依次将顶点B,C,D,E,F,G作为中介点，并更新a[i][j]的大小。

## 2.3 代码实现

以"邻接矩阵"为例对弗洛伊德算法进行说明, 对于"邻接表"实现的图在后面会给出相应的源码。

```
public class MatrixUDG {

    private int mEdgNum;           // 边的数量
    private char[] mVexs;          // 顶点集合
    private int[][] mMatrix;       // 邻接矩阵
    private static final int INF = Integer.MAX_VALUE; // 最大值

    ...
}
```

MatrixUDG是邻接矩阵对应的结构体。mVexs用于保存顶点，mEdgNum用于保存边数，mMatrix则是用于保存矩阵信息的二维数组。例如，mMatrix[i][j]=1，则表示"顶点i(即mVexs[i])"和"顶点j(即mVexs[j])"是邻接点；mMatrix[i][j]=0，则表示它们不是邻接点。

```

/*
 * floyd最短路径。
 * 即，统计图中各个顶点间的最短路径。
 *
 * 参数说明：
 *     path -- 路径。path[i][j]=k表示，"顶点i"到"顶点j"的最短路径会经过顶点k。
 *     dist -- 长度数组。即，dist[i][j]=sum表示，"顶点i"到"顶点j"的最短路径的长度是sum。
 */
public void floyd(int[][] path, int[][] dist) {

    // 初始化
    for (int i = 0; i < mVexs.length; i++) {
        for (int j = 0; j < mVexs.length; j++) {
            dist[i][j] = mMatrix[i][j];    // "顶点i"到"顶点j"的路径长度为"i到j的权值"
            path[i][j] = j;                // "顶点i"到"顶点j"的最短路径是经过顶点j。
        }
    }

    // 计算最短路径
    for (int k = 0; k < mVexs.length; k++) {
        for (int i = 0; i < mVexs.length; i++) {
            for (int j = 0; j < mVexs.length; j++) {

                // 如果经过下标为k顶点路径比原两点间路径更短，则更新dist[i][j]和path[i][j]
            }
        }
    }
}

```



```

j]
        int tmp = (dist[i][k]==INF || dist[k][j]==INF) ? INF : (dist[i][k]
+ dist[k][j]);
        if (dist[i][j] > tmp) {
            // "i到j最短路径"对应的值设, 为更小的一个(即经过k)
            dist[i][j] = tmp;
            // "i到j最短路径"对应的路径, 经过k
            path[i][j] = path[i][k];
        }
    }
}

// 打印floyd最短路径的结果
System.out.printf("floyd: \n");
for (int i = 0; i < mVexs.length; i++) {
    for (int j = 0; j < mVexs.length; j++)
        System.out.printf("%2d ", dist[i][j]);
    System.out.printf("\n");
}
}

```

## 2.4 时间复杂度

Floyd-Warshall算法的时间复杂度为 $O(N^3)$ , 空间复杂度为 $O(N^2)$ 。