最短路径

一、Dijkstra 算法

给定有向带权图 G=(V, E)，其中每条边的**权值都是非负实数**。此外，给定 V 中的一个节点，称之为源点。求解从源点到其他各个节点的最短路径长度，路径长度指路上各边权之和。

如何求源点到其他各个节点的最短路径长度呢？

荷兰计算机科学家迪科斯彻提出了著名的**单源最短路径求解算法——Dijkstra 算法**。Dijkstra 算法是解决单源最短路径问题的**贪心算法**，它先求出长度最短的一条路径，再参照该最短路径求出长度次短的一条路径，直到求出从源点到其他各个节点的最短路径。

Dijkstra 算法的基本思想：假定源点 u，节点集合 V 被划分为两部分：集合 S 和集合 V−S。初始时，在集合 S 中仅包含源点 u，S 中的节点到源点的最短路径已经确定。集合 V−S 所包含的节点到源点的最短路径的长度待定，称从源点出发只经过集合 S 中的节点到达集合 V−S 中的节点的路径为特殊路径，并用数组 **dist[ ]记录当前每个节点所对应的最短特殊路径长度。**

Dijkstra 算法采用的贪心策略是选择特殊路径长度最短的路径，将其连接的集合 V−S 中的节点加入集合 S 中，同时更新数组 dist[]。一旦集合 S 包含所有节点，dist[ ]就是从源点到所有其他节点的最短路径长度。

1. 算法步骤

（1）数据结构。设置地图的邻接矩阵为 G.Edge[ ][ ]，即如果从源点 u 到节点 i 有边，就令 G.Edge[u][i]等于<u,i>的权值，否则 G.Edge[u][i]=∞（无穷大）；采用一维数组 dist[i]记录从源点到节点 i 的最短路径长度；采用一维数组 p[i]记录最短路径上节点 i 的前驱（记录最短路径）。

（2）初始化。令集合 S={u}，对于集合 V−S 中的所有节点 i，都初始化 dist[i]=G.Edge[u][i]，如果从源点 u 到节点 i 有边相连，则初始化 p[i]=u，否则 p[i]= −1。

（3）找最小。在集合 V−S 中查找 dist[ ]最小的节点 t，即 dist[t]=min（dist[j] | j 属于集合 V−S），则节点 t 就是集合 V−S 中距离源点 u 最近的节点。

（4）加入集合 S 中。将节点 t 加入集合 S 中，同时更新集合 V−S。

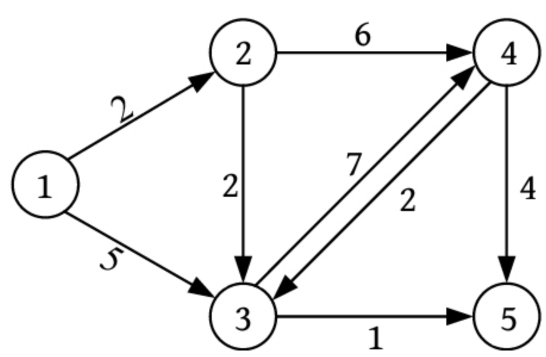
（5）判结束。如果集合 V−S 为空，则算法结束，否则转向步骤 6。

（6）借东风（松弛）。在步骤 3 中已经找到了从源点到节点 t 的最短路径，那么对集合 V−S 中节点 t 的所有邻接点 j，都可以借助 t 走捷径。如果 dist[j]>dist[t]+G.Edge[t][j]，则 dist[j]=dist[t]+G.Edge[t][j]，记录节点 j 的前驱为 t，有 p[j]=t，转向步骤 3。

由此，可求得从源点 u 到图 G 的其余各个节点的最短路径及长度，也可通过数组 p[ ]逆向找到最短路径上的节点。

2. 图解

有一个景点地图，如下图所示，假设从节点 1 出发，求到其他各个节点的最短路径。



（1）数据结构。设置地图的带权邻接矩阵为 G.Edge[ ][ ]，即如果从节点 i 到节点 j 有边，则 G.Edge[i][j]等于<i, j>的权值，否则 G.Edge[i][j]=∞（无穷大），如下图所示。

$$\begin{bmatrix} \infty & 2 & 5 & \infty & \infty \\ \infty & \infty & 2 & 6 & \infty \\ \infty & \infty & \infty & 7 & 1 \\ \infty & \infty & 2 & \infty & 4 \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

（2）初始化。令集合 S={1}，集合 V−S={2,3,4,5}，对于集合 V−S 中的所有节点 x，都初始化最短距离数组 dist[i]=G.Edge[1][i]，dist[u]=0。如果从源点 1 到节点 i 有边相连，则初始化前驱数组 p[i]=1，否则 p[i]= −1，如下图所示。

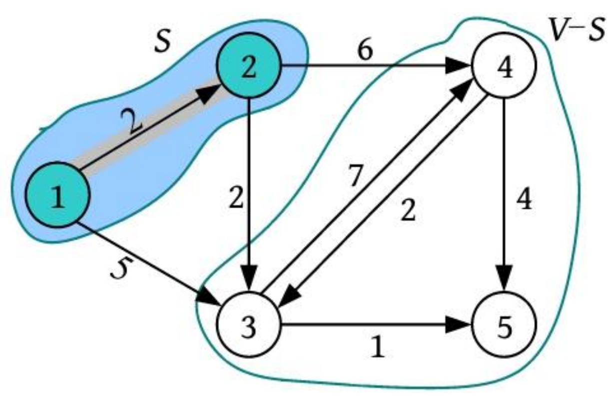
	1	2	3	4	5
dist[]	0	2	5	∞	∞

	1	2	3	4	5
p[]	-1	1	1	-1	-1

（3）找最小。在集合 V−S={2,3,4,5}中查找 dist[]最小的节点 t，找到的最小值为 2，对应的节点 t=2，如下图所示。

	1	2	3	4	5
dist[]	0	2	5	∞	∞

（4）加入集合 S 中。将节点 2 加入集合 S={1,2}中，同时更新集合 V−S={3,4,5}，如下图所示。



（5）借东风（**松弛**）。刚刚找到了从源点到节点 t=2 的最短路径，那么对集合 V−S 中节点 t 的所有邻接点 j，都可以借助节点 t 走捷径。节点 2 的邻接点是节点 3 和节点 4。先看节点 3 能否借助节点 2 走捷径：dist[2]+G.Edge[2][3]=2+2=4，而当前 dist[3]=5>4，因此可以走捷径，即 2-3，更新 dist[3]=4，记录节点 3 的前驱为节点 2，即 p[3]=2。再看节点 4 能否借助节点 2 走捷径：如果 dist[2]+G.Edge[2][4]=2+6=8，而当前 dist[4]=∞>8，因此可以走捷径，即 2-4，更新 dist[4]=8，记录节点 4 的前驱为节点 2，即 p[4]= 2。更新后如下图所示。

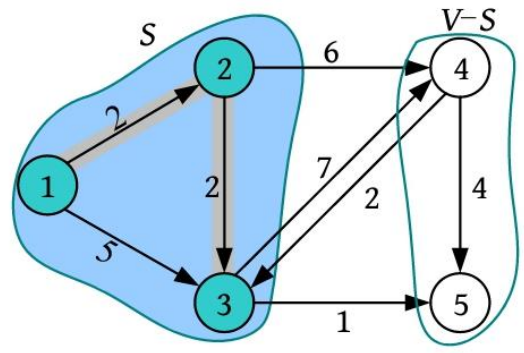
	1	2	3	4	5
dist[]	0	2	4	8	∞

	1	2	3	4	5
p[]	-1	1	2	2	-1

（6）找最小。在集合 V−S={3,4,5}中，查找 dist[ ]最小的节点 t，找到的最小值为 4，对应的节点 t=3。

	1	2	3	4	5
dist[]	0	2	4	8	∞

（7）加入集合 S 中。将节点 3 加入集合 S={1,2,3}中，同时更新集合 V−S={4,5}，如下图所示。



（8）借东风（松弛）。刚刚找到了从源点到节点 t=3 的最短路径，那么对集合 V−S 中节点 t 的所有邻接点 j，都可以借助 t 走捷径。节点 3 的邻接点是节点 4 和节点 5。先看节点 4 能否借助节点 3 走捷径：dist[3]+G.Edge[3][4]=4+7=11，而当前 dist[4]=8<11，比当前路径还长，因此不更新。再看节点 5 能否借助节点 3 走捷径：dist[3]+G.Edge[3][5]=4+1=5，而当前 dist[5]=∞>5，可以走捷径，即 3-5，更新 dist[5]=5，记录节点 5 的前驱为节点 3，即 p[5]=3。更新后如下图所示。

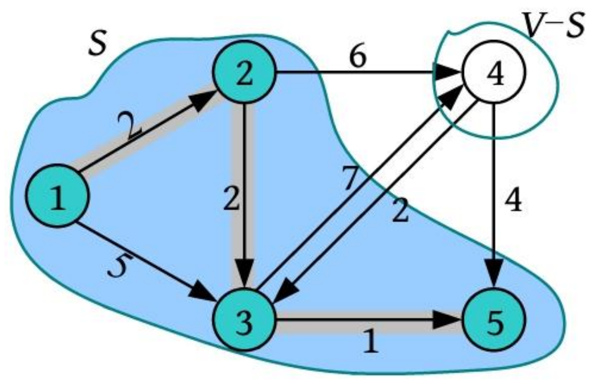
	1	2	3	4	5
dist[]	0	2	4	8	5

	1	2	3	4	5
p[]	-1	1	2	2	3

（9）找最小。在集合 V−S={4,5}中，查找 dist[]最小的节点 t，找到的最小值为 5，对应的节点 t=5，如下图所示。

	1	2	3	4	5
dist[]	0	2	4	8	5

（10）加入集合 S 中。将节点 5 加入集合 S={1,2,3,5}中，同时更新集合 V−S={4}，如下图所示。



（11）借东风（松弛）。刚刚找到了从源点到 t=5 的最短路径，那么对集合 V−S 中节点 t 的所有邻接点 j，都可以借助节点 t 走捷径。节点 5 没有邻接点，因此不更新，如下图所示。

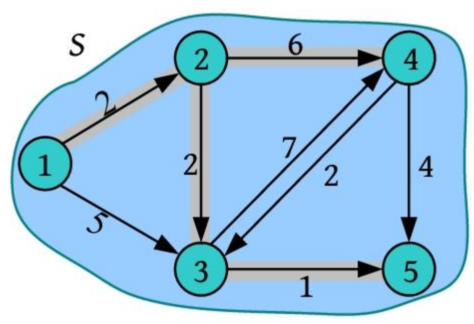
	1	2	3	4	5
dist[]	0	2	4	8	5

	1	2	3	4	5
p[]	-1	1	2	2	3

（12）找最小。在集合 V−S={4}中查找 dist[]最小的节点 t，找到的最小值为 8，对应的节点 t=4，如下图所示。

	1	2	3	4	5
dist[]	0	2	4	8	5

（13）加入集合 S 中。将节点 4 加入集合 S={1,2,3,5,4}中，同时更新集合 V−S={ }，如下图所示。



（14）算法结束。在集合 V−S={ }为空时算法停止。

由此，可求得从源点 u 到图 G 的其余各个节点的最短路径及长度，也可通过前驱数组 p[ ]逆向找到最短路径上的节点，如下图所示。

	1	2	3	4	5
p[]	−1	1	2	2	3

例如，p[5]=3，即节点 5 的前驱是节点 3；p[3]=2，即节点 3 的前驱是节点 2；p[2]=1，即节点 2 的前驱是节点 1；p[1]= −1，节点 1 没有前驱，那么从源点 1 到 5 的最短路径为 1-2-3-5。

3. 算法实现

```
void Dijkstra(AMGraph G,int u){
   for(int i=0;i<G.vexnum;i++){
      dist[i]=G.Edge[u][i]; //初始化源点 u 到其他各个节点的最短路径长度
      flag[i]=false;
      if(dist[i]==INF)
         p[i]=-1; //节点 i 与源点 u 不相邻
      else
         p[i]=u; //节点 i 与源点 u 相邻，设置节点 i 的前驱 p[i]=u
   }
   dist[u]=0;
   flag[u]=true; //初始时，在集合 S 中只有一个元素：源点 u
   for(int i=0;i<G.vexnum; i++){
      int temp=INF,t=u;
      for(int j=0;j<G.vexnum; j++) //在集合 V-S 中寻找距离源点 u 最近的节点 t
         if(!flag[j]&&dist[j]<temp){
            t=j;
            temp=dist[j];
         }
      if(t==u) return ; //找不到 t，跳出循环
      flag[t]=true;  //否则，将 t 加入集合
      for(int j=0;j<G.vexnum;j++)//更新与 t 相邻接的节点到源点 u 的距离
         if(!flag[j]&&G.Edge[t][j]<INF)
            if(dist[j]>(dist[t]+G.Edge[t][j])){
               dist[j]=dist[t]+G.Edge[t][j] ;
               p[j]=t ;
             }
   }
}
```

输出最短路径上的节点序列： p[ ]数组记录了最短路径上每一个节点的前驱，因此除了显示最短距离，还可以显示最短路径上的节点，可以增加一段程序逆向找到该最短路径上的节点序列。

```
void findpath(AMGraph G,VexType u){
    int x;
    stack<int>S;
    cout<<"源点为："<<u<<endl;
    for(int i=0;i<G.vexnum;i++){
        x=p[i];
        if(x==-1&&u!=G.Vex[i]){
            cout<<u<<"--"<<G.Vex[i]<<" 无路可达！"<<endl;
            continue;
        }
        while(x!=-1){
            S.push(x);
            x=p[x];
        }
        cout<<"从源点到其他各节点的最短路径为：";
        while(!S.empty()){
            cout<<G.Vex[S.top()]<<"--";
            S.pop();
        }
        cout<<G.Vex[i]<<"    最短距离为："<<dist[i]<<endl;
    }
}
```

4. 算法分析

时间复杂度：在 Dijkstra 算法描述中共有 4 个 for 语句，第 1 个 for 语句的执行次数为 n；在第 2 个 for 语句里面嵌套了两个 for 语句。这两个 for 语句在内层对算法的运行时间贡献最大，语句的执行次数为 n^2，算法的时间复杂度为 **$O(n^2)$。**

空间复杂度：辅助空间包含数组 flag[]及 i、j、t 和 temp 等变量，空间复杂度为 **O(n)。**

5. 算法优化

（1）优先队列优化。第 3 个 for 语句是在集合 V-S 中寻找距离源点 u 最近的节点 t，如果穷举，则需要 O(n)时间。如果采用优先队列，则寻找一个最近节点需要 O(logn)时间。**时间复杂度为 O(nlogn)。**

（2）数据结构优化。第 4 个 for 语句是松弛操作，采用邻接矩阵存储，访问一个节点的所有邻接点需要执行 n 次，总时间复杂度为 $O(n^2)$。如果采用**邻接表存储**，则访问一个节点的所有邻接点的执行次数为该节点的出度，所有节点的出度之和为 m（边数），总时间复杂度为 **O(m)。**对于**稀疏图，O(m)要比 $O(n^2)$小。**

二、Floyd 算法

Dijkstra 算法用于求从源点到其他各个节点的最短路径。如果求解任意两个节点之间的最短路径，则需要以每个节点为源点，重复调用 n 次 Dijkstra 算法。其实完全没必要这么麻烦，Floyd 算法可用于求解任意两个节点间的最短路径。Floyd 算法又被称为插点法，其算法核心是在节点 i 与节点 j 之间插入节点 k，看看是否可以缩短节点 i 与节点 j 之间的距离（松弛操作）。

1. 算法步骤

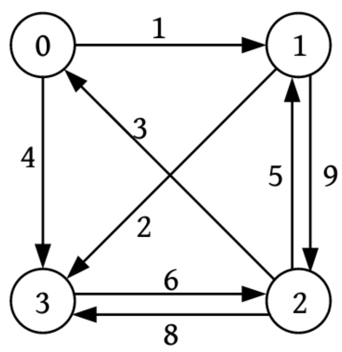
（1）数据结构。设置地图的带权邻接矩阵为 G.Edge[ ][ ]，即如果从节点 i 到节点 j 有边，则 G.Edge[i][j]=<i,j>的权值，否则 G.Edge[i][j]=∞（无穷大）；采用两个辅助数组：最短距离数组 dist[i][j]，记录从节点 i 到节点 j 的最短路径长度；前驱数组 p[i][j]，记录从节点 i 到节点 j 的最短路径上节点 j 的前驱。

（2）初始化。初始化 dist[i][j]=G.Edge[i][j]，如果从节点 i 到节点 j 有边相连，则初始化 p[i][j]=i，否则 p[i][j]=-1。

（3）插点。其实就是在节点 i、j 之间插入节点 k，看是否可以缩短节点 i、j 之间的距离（松弛操作）。如果 dist[i][j]>dist[i][k]+dist[k][j]，则 dist[i][j]=dist[i][k]+dist[k][j]，记录节点 j 的前驱 p[i][j]=p[k][j]。

2. 图解

有一个景点地图，如下图所示，假设从节点 0 出发，求各个节点之间的最短路径。



（1）数据结构。地图采用邻接矩阵存储，如果从节点 i 到节点 j 有边，则 G.Edge[i][j]=<i, j>的权值；当 i=j 时，G.Edge[i][i]=0,否则 G.Edge[i][j]=∞（无穷大）。

$$\begin{bmatrix} 0 & 1 & \infty & 4 \\ \infty & 0 & 9 & 2 \\ 3 & 5 & 0 & 8 \\ \infty & \infty & 6 & 0 \end{bmatrix}$$

（2）初始化。初始化最短距离数组 dist[i][j]=G.Edge[i][j]，如果从节点 i 到节点 j 有边相连，则初始化前驱数组 p[i][j]=i，否则 p[i][j]=-1。初始化后的 dist[][]和 p[][]如下图所示。

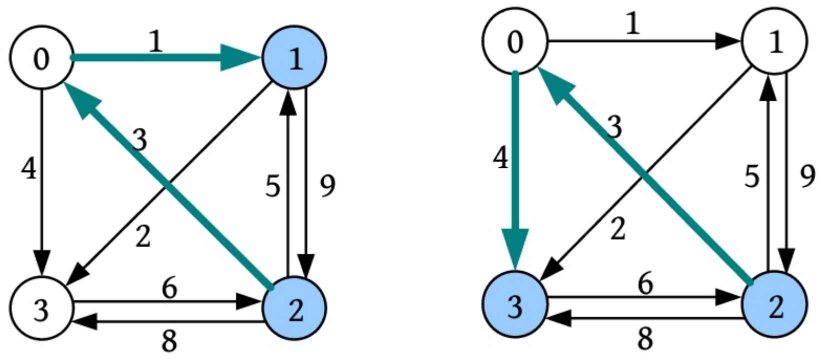
$$\text{dist}[i][j]=\begin{bmatrix} 0 & 1 & \infty & 4 \\ \infty & 0 & 9 & 2 \\ 3 & 5 & 0 & 8 \\ \infty & \infty & 6 & 0 \end{bmatrix} \qquad p[i][j]=\begin{bmatrix} -1 & 0 & -1 & 0 \\ -1 & -1 & 1 & 1 \\ 2 & 2 & -1 & 2 \\ -1 & -1 & 3 & -1 \end{bmatrix}$$

（3）插点（k=0）。其实就是“借点、借东风”，考查所有节点是否可以借助节点 0 更新最短距离。如果 dist[i][j]>dist[i][0]+dist[0][j]，则 dist[i][j]=dist[i][0]+dist[0][j]，记录节点 j 的前驱为 p[i][j]=p[0][j]。谁可以借节点 0 呢？看节点 0 的入边 2-0，也就是说节点 2 可以借节点 0，更新 2 到其他节点的最短距离（在程序中需要穷举所有节点是否可以借助节点 0）。

- dist[2][1]：dist[2][1]=5>dist[2][0]+dist[0][1]=4，更新 dist[2][1]=4，p[2][1]=0。在节点 2、1 之间插入节点 0。

- dist[2][3]：dist[2][3]=8>dist[2][0]+dist[0][3]=7，更新 dist[2][3]=7，p[2][3]=0。在节点 2、3 之间插入节点 0。

以上两个最短距离的更新如下图所示。



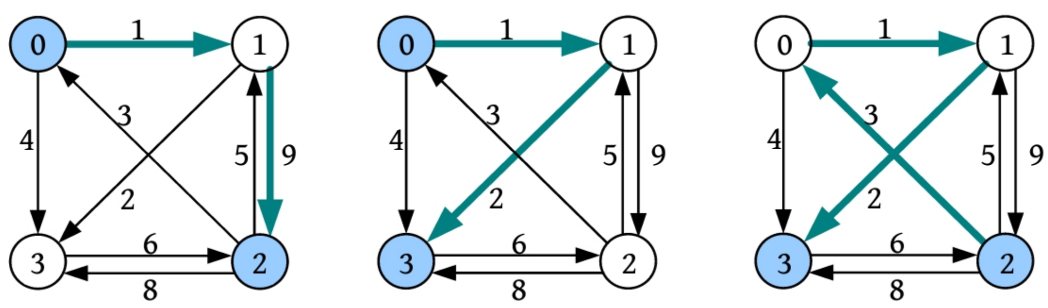
更新后的最短距离数组和前驱数组如下图所示。

$$\text{dist}[i][j]=\begin{bmatrix}0 & 1 & \infty & 4\\ \infty & 0 & 9 & 2\\ 3 & 4 & 0 & 7\\ \infty & \infty & 6 & 0\end{bmatrix}\qquad p[i][j]=\begin{bmatrix}-1 & 0 & -1 & 0\\ -1 & -1 & 1 & 1\\ 2 & 0 & -1 & 0\\ -1 & -1 & 3 & -1\end{bmatrix}$$

（4）插点（k=1）。考查所有节点是否可以借助节点 1 更新最短距离。看节点 1 的入边，节点 0、2 都可以借助节点 1 更新其到其他节点的最短距离。

- dist[0][2]：dist[0][2]=∞>dist[0][1]+dist[1][2]=10，更新 dist[0][2]=10，p[0][2]=1。在节点 0、2 之间插入节点 1。
- dist[0][3]：dist[0][3]=4>dist[0][1]+dist[1][3]=3，更新 dist[0][3]=3，p[0][3]=1。在节点 0、3 之间插入节点 1。
- dist[2][0]：dist[2][0]=3<dist[2][1]+dist[1][0]=∞，不更新。
- dist[2][3]：dist[2][3]=8>dist[2][1]+dist[1][3]=6，更新 dist[0][2]=6，p[2][3]=1。在节点 2、3 之间插入节点 1。

以上 3 个最短距离的更新如下图所示。



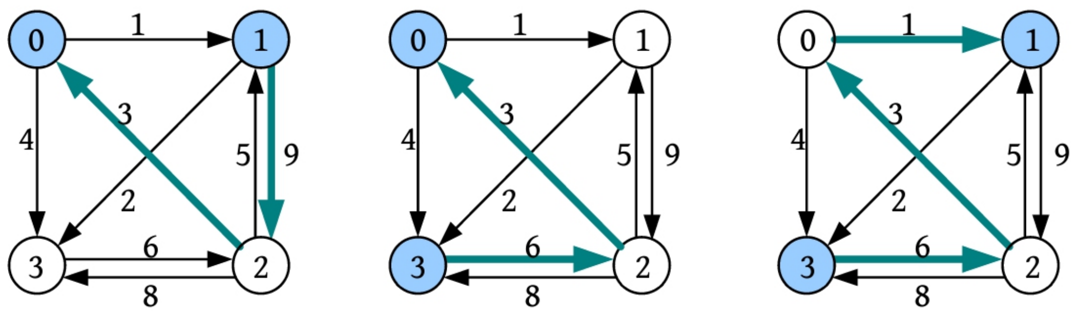
更新后的最短距离数组和前驱数组如下图所示。

$$\text{dist}[i][j]=\begin{bmatrix}0 & 1 & 10 & 3\\ \infty & 0 & 9 & 2\\ 3 & 4 & 0 & 6\\ \infty & \infty & 6 & 0\end{bmatrix}\qquad p[i][j]=\begin{bmatrix}-1 & 0 & 1 & 1\\ -1 & -1 & 1 & 1\\ 2 & 0 & -1 & 1\\ -1 & -1 & 3 & -1\end{bmatrix}$$

（5）插点（k=2）。考查所有节点是否可以借助节点 2 更新最短距离。看节点 2 的入边，节点 1、3 都可以借节点 2 更新其到其他节点的最短距离。

- dist[1][0]：dist[1][0]=∞>dist[1][2]+dist[2][0]=12，更新 dist[1][0]=12，p[1][0]=2。在节点 1、0 之间插入节点 2。
- dist[1][3]：dist[1][3]=2<dist[1][2]+dist[2][3]=15，不更新。
- dist[3][0]：dist[3][0]=∞>dist[3][2]+dist[2][1]=9，更新 dist[3][0]=9，p[3][0]=2。在节点 3、0 之间插入节点 2。
- dist[3][1]：dist[3][1]=∞>dist[3][2]+dist[2][1]=10，更新 dist[3][1]=10，p[3][1]=p[2][1]=0。在节点 3、1 之间插入节点 2。

以上 3 个最短距离的更新如下图所示。



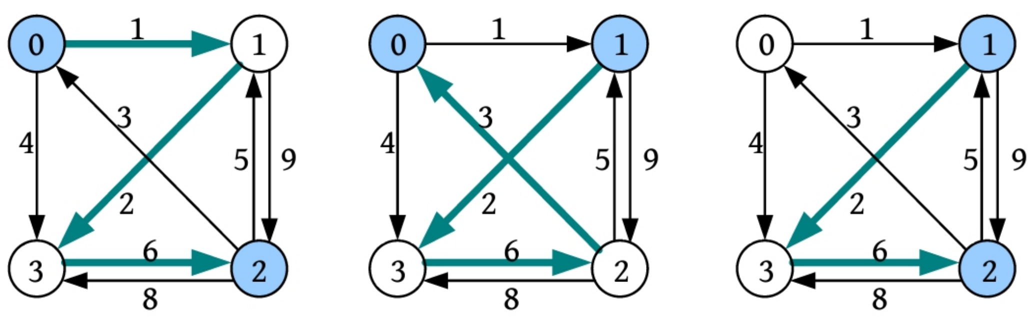
更新后的最短距离数组和前驱数组如下图所示。

$$\text{dist}[i][j]=\begin{bmatrix}0 & 1 & 10 & 3\\ 12 & 0 & 9 & 2\\ 3 & 4 & 0 & 6\\ 9 & 10 & 6 & 0\end{bmatrix}\qquad p[i][j]=\begin{bmatrix}-1 & 0 & 1 & 1\\ 2 & -1 & 1 & 1\\ 2 & 0 & -1 & 1\\ 2 & 0 & 3 & -1\end{bmatrix}$$

（6）插点（k=3）。考查所有节点是否可以借助节点 3 更新最短距离。看节点 3 的入边，节点 0、1、2 都可以借助节点 3 更新其到其他节点的最短距离。

- dist[0][1]：dist[0][1]=1<dist[0][3]+dist[3][1]=13，不更新。
- dist[0][2]：dist[0][2]=10>dist[0][3]+dist[3][2]=9，更新 dist[0][2]=9，p[0][2]=3。在节点 0、2 之间插入节点 3 点。
- dist[1][0]：dist[1][0]=12>dist[1][3]+dist[3][0]=11，更新 dist[1][0]=11，p[1][0]=p[3][0]= 2。在节点 1、0 之间插入节点 3。
- dist[1][2]：dist[1][2]=9>dist[1][3]+dist[3][2]=8，则更新 dist[1][2]=8，p[1][2]=3。在节节点 1、2 之间插入节点 3。
- dist[2][0]：dist[2][0]=3<dist[2][3]+dist[3][0]=15，不更新。
- dist[2][1]：dist[2][1]=4<dist[2][3]+dist[3][1]=16，不更新。

以上 3 个最短距离的更新如下图所示。



更新后的最短距离数组和前驱数组如下图所示。

$$\text{dist}[i][j]=\begin{bmatrix}0 & 1 & 9 & 3\\ 11 & 0 & 8 & 2\\ 3 & 4 & 0 & 6\\ 9 & 10 & 6 & 0\end{bmatrix}\qquad p[i][j]=\begin{bmatrix}-1 & 0 & 3 & 1\\ 2 & -1 & 3 & 1\\ 2 & 0 & -1 & 1\\ 2 & 0 & 3 & -1\end{bmatrix}$$

（7）插点结束。dist[ ][ ]数组包含了各节点之间的最短距离，如果想找从节点 i 到节点 j 的最短路径，则可以根据前驱数组 p[ ][ ]找到。例如，求从节点 1 到节点 2 的最短路径，首先读取 p[1][2]=3，说明节点 2 的前驱为节点 3，继续向前找，读取 p[1][3]=1，说明节点 3 的前驱为节点 1，得到从节点 1 到节点 2 的最短路径为 1-3-2。求从节点 1 到节点 0 的最短路径，首先读取 p[1][0]=2，说明节点 0 的前驱为节点 2，继续向前找，读取 p[1][2]=3，说明节点 2 的前驱为节点 3，继续向前找，读取 p[1][3]=1，得到从节点 1 到节点 0 的最短路径为 1-3-2-0。

3. 算法实现

```
void Floyd(AMGraph G){//用 Floyd 算法求有向网 G 中各对节点 i 和 j 之间的最短路径
    int i,j,k;
    for(i=0;i<G.vexnum;i++)//各对节点之间的初始距离及已知路径
      for(j=0;j<G.vexnum;j++){
         dist[i][j]=G.Edge[i][j];
         if(dist[i][j]<INF && i!=j)
            p[i][j]=i;   //如果在节点 i 和节点 j 之间有弧，则将节点 j 的前驱置为 i
         else p[i][j]=-1;  //如果在节点 i 和节点 j 之间无弧，则将节点 j 的前驱置为-1
      }
    for(k=0;k<G.vexnum; k++)
      for(i=0;i<G.vexnum; i++)
         for(j=0;j<G.vexnum; j++)
            if(dist[i][k]+dist[k][j]<dist[i][j]) {//从节点 i 经节点 k 到节点 j 的一条路径更短
               dist[i][j]=dist[i][k]+dist[k][j]; //更新 dist[i][j]
               p[i][j]=p[k][j];  //更改 j 的前驱为 k
            }
}
```

4. 算法分析

时间复杂度：三层 for 循环，时间复杂度为 **O(n³)**。

空间复杂度：采用最短距离数组 dist[ ][ ]和前驱数组 p[ ][ ]，空间复杂度为 **O(n²)**。

尽管 Floyd 算法的时间复杂度为 $O(n^3)$，但其代码简单，对于中等输入规模来说，仍然相当有效。如果用 Dijkstra 算法求解各个节点之间的最短路径，则需要以每个节点为源点都调用一次，共调用 n 次，其总的时间复杂度也为 $O(n^3)$。特别注意的是，**Dijkstra 算法无法处理带有负权边的图。**如果有**负权边，则可以采用 Bellman-Ford 算法或 SPFA 算法。**

三、Bellman-Ford 算法

如果遇到负权边，则在**没有负环（回路的权值之和为负）**存在时，可以采用 Bellman-Ford 算法求解最短路径。**Bellman-Ford 算法**用于**求解单源最短路径问题**，由理查德•贝尔曼和莱斯特•福特提出。该算法的优点是边的权值可以为负数、实现简单，缺点是时间复杂度过高。但是，对该算法可以进行若干种优化，以提高效率。

Bellman-Ford 算法与 Dijkstra 算法类似，都**以松弛操作为基础。Dijkstra 算法以贪心法**选取未被处理的具有最小权值的节点，然后**对其出边进行松弛**操作；而 **Bellman-Ford 算法对所有边都进行松弛**操作，共 **n-1 次。**因为负环可以无限制地减少最短路径长度，所以**如果发现第 n 次操作仍可松弛，则一定存在负环。**Bellman-Ford 算法的最长运行时间为 **O(nm)**，其中 n 和 m 分别是节点数和边数。

1. 算法步骤

（1）数据结构。因为需要利用边进行松弛，因此采用边集数组存储。每条边都有三个域：两个端点 a、b 和边权 w。

（2）松弛操作。对所有的边 j(a,b,w)，如果 dis[e[j].b]>dis[e[j].a]+e[j].w，则松弛，令 dis[e[j].b]=dis[e[j].a]+e[j].w。其中，dis[v]表示从源点到节点 v 的最短路径长度。

（3）重复松弛操作 n-1 次。

（4）**负环判定**（简称“判负环”）。**再执行一次松弛**操作，**如果仍然可以松弛，则说明有负环。**

2. 算法实现

```
bool bellman_ford(int u){//求从源点 u 到其他节点的最短路径长度，并判断是否有负环
   memset(dis,0x3f,sizeof(dis));
   dis[u]=0;
   for(int i=1;i<n;i++){//执行 n-1 次
      bool flag=0;
      for(int j=0;j<m;j++)//边数 m 或 cnt
         if(dis[e[j].b]>dis[e[j].a]+e[j].w){
            dis[e[j].b]=dis[e[j].a]+e[j].w;
            flag=true;
         }
      if(!flag)
         return false;
   }
   for(int j=0;j<m;j++)//再执行 1 次，还能松弛，说明有环
      if(dis[e[j].b]>dis[e[j].a]+e[j].w)
         return true;
   return false;
}
```

3. 算法优化

（1）提前退出循环。在实际操作中，Bellman-Ford 算法经常会在未达到 n-1 次时就求解完毕，可以在循环中设置判定，在某次循环不再进行松弛时，直接退出循环。通过上段代码中的 if(!flag)就可以提前退出循环。

（2）队列优化。松弛操作必定只会发生在最短路径松弛过的前驱节点上，用一个**队列记录松弛过的节点，可以避免冗余计算。**这就是队列优化的 Bellman-Ford 算法，又被称为 **SPFA 算法。**

四、SPFA 算法

SPFA（Shortest Path Faster Algorithm）算法是**Bellman-Ford 算法的队列优化算法**，通常用于**求解含负权边的单源最短路径，以及判负环。**在**最坏情况下**，SPFA 算法的时间复杂度和 Bellman-Ford 算法相同，为 **O(nm)**；但在**稀疏图上运行效率较高**，为 **O(km)**，其中 **k 是一个较小的常数。**

1. 算法步骤

（1）创建一个队列，首先源点 u 入队，标记 u 在队列中，u 的入队次数加 1。

（2）松弛操作。取出队头节点 x，标记 x 不在队列中。扫描 x 的所有出边 i(x,v,w)，如果 dis[v]>dis[x]+e[i].w，则松弛，令 dis[v]=dis[x]+e[i].w。如果节点 v 不在队列中，判断 v 的入队次数加 1 后大于或等于 n，则说明有负环，退出；否则 v 入队，标记 v 在队列中。

（3）重复松弛操作，直到队列为空。

2. 算法实现

```
bool spfa(int u){
    queue<int>q;
    memset(vis,0,sizeof(vis));    //标记是否在队列中
    memset(sum,0,sizeof(sum));    //统计入队的次数
    memset(dis,0x3f,sizeof(dis));
    vis[u]=1;
    dis[u]=0;
    sum[u]++;
    q.push(u);
    while(!q.empty()){
        int x=q.front();
        q.pop();
        vis[x]=0;
        for(int i=head[x];~i;i=e[i].next){//链式前向星存储图
            int v=e[i].to;
            if(dis[v]>dis[x]+e[i].w){
                dis[v]=dis[x]+e[i].w;
                if(!vis[v]){
                    if(++sum[v]>=n)
                        return true;
                    vis[v]=1;
                    q.push(v);
                }
            }
        }
    }
    return false;
}
```

3. 算法优化

SPFA 算法有两个优化策略：**SLF 和 LLL。**

（1）SLF（Small Label First）策略：如果待入队的节点是 j，队首元素为节点 i，若 dis[j]<dis[i]，则将 j 插入队首，否则插入队尾。

（2）LLL（Large Label Last）策略：设队首元素为节点 i，队列中所有 dis[ ]的平均值为 x，若 dis[i]>x，则将节点 i 插入队尾，查找下一元素，直到找到某一节点 i 满足 dis[i]≤x，将节点 i 出队，进行松弛操作。

SLF 和 LLL 在随机数据上表现优秀，但是在正权图上的最坏情况为 O(nm)，在负权图上的最坏情况为达到指数级复杂度。

如果**在图中没有负权边**，则可以**采用优先队列优化 SPFA**，每次都取出当前 dis[ ]最小的节点扩展，节点第 1 次被从优先队列中取出时，就得到了该节点的最短路径。**这与优先队列优化的 Dijkstra 算法类似**，时间复杂度均为 **O(mlogn)。**