**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。

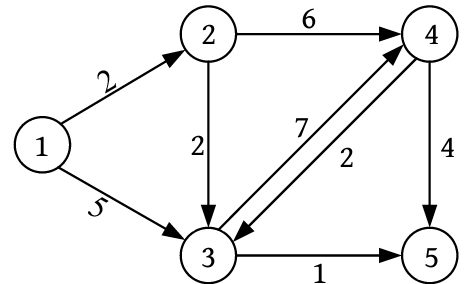
（5）判结束。如果集合V−S为空，则算法结束，否则转向步骤6。

（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。



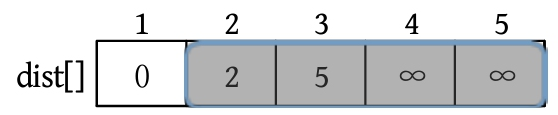
（1）数据结构。设置地图的带权邻接矩阵为G.Edge[ ][ ]，即如果从节点i到节点j有边，则G.Edge[i][j]等于<i, j>的权值，否则G.Edge[i][j]=∞（无穷大），如下图所示。



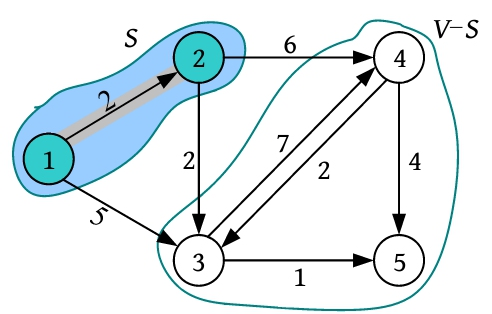
（2）初始化。令集合S={1}，集合V−S={2,3,4,5}，对于集合V−S中的所有节点x，都初始化最短距离数组dist[i]=G.Edge[1][i]，dist[u]=0。如果从源点1到节点i有边相连，则初始化前驱数组p[i]=1，否则p[i]= −1，如下图所示。



（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。



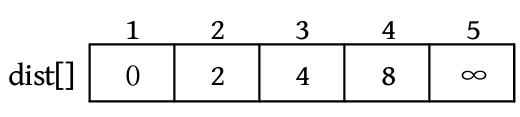
（4）加入集合S中。将节点2加入集合S={1,2}中，同时更新集合V−S={3,4,5}，如下图所示。



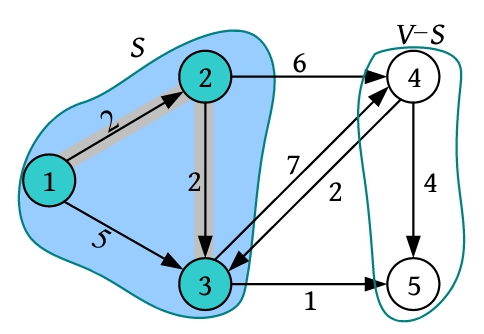
（5）借东风（**松弛**）。刚刚找到了从源点到节点t=2的最短路径，那么对集合V−S中节点t的所有邻接点j，都可以借助节点t走捷径。节点2的邻接点是节点3和节点4。先看节点3能否借助节点2走捷径：dist[2]+G.Edge[2][3]=2+2=4，而当前dist[3]=5>4，因此可以走捷径，即2-3，更新dist[3]=4，记录节点3的前驱为节点2，即p[3]=2。再看节点4能否借助节点2走捷径：如果dist[2]+G.Edge[2][4]=2+6=8，而当前dist[4]=∞>8，因此可以走捷径，即2-4，更新dist[4]=8，记录节点4的前驱为节点2，即p[4]= 2。更新后如下图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



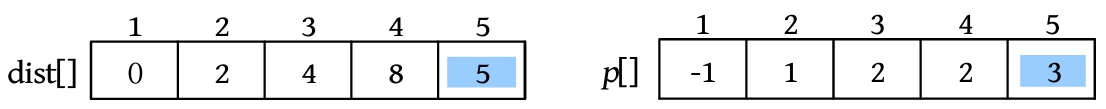
（6）找最小。在集合V−S={3,4,5}中，查找dist[ ]最小的节点t，找到的最小值为4，对应的节点t=3。



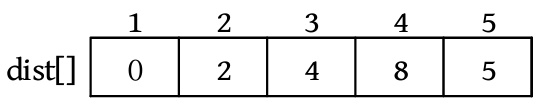
（7）加入集合S中。将节点3加入集合S={1,2,3}中，同时更新集合V−S={4,5}，如下图所示。



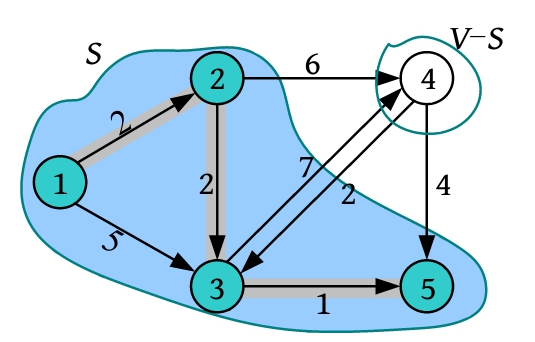
（8）借东风（松弛）。刚刚找到了从源点到节点t=3的最短路径，那么对集合V−S中节点t的所有邻接点j，都可以借助t走捷径。节点3的邻接点是节点4和节点5。先看节点4能否借助节点3走捷径：dist[3]+G.Edge[3][4]=4+7=11，而当前dist[4]=8<11，比当前路径还长，因此不更新。再看节点5能否借助节点3走捷径：dist[3]+G.Edge[3][5]=4+1=5，而当前dist[5]=∞>5，可以走捷径，即3-5，更新dist[5]=5，记录节点5的前驱为节点3，即p[5]=3。更新后如下图所示。



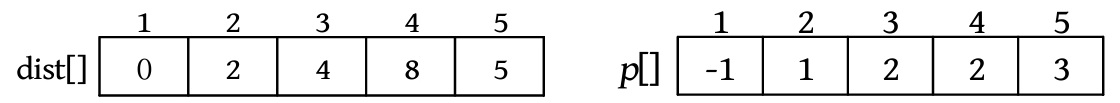
（9）找最小。在集合V−S={4,5}中，查找dist[]最小的节点t，找到的最小值为5，对应的节点t=5，如下图所示。



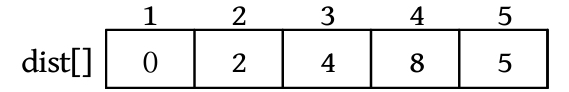
（10）加入集合S中。将节点5加入集合S={1,2,3,5}中，同时更新集合V−S={4}，如下图所示。



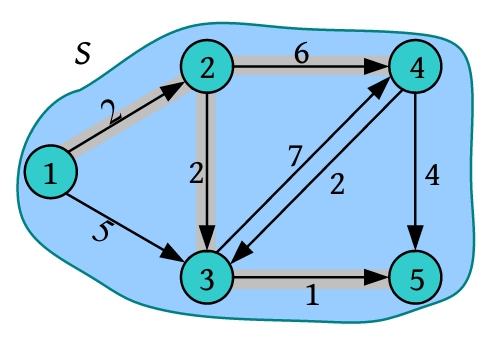
（11）借东风（松弛）。刚刚找到了从源点到t=5的最短路径，那么对集合V−S中节点t的所有邻接点j，都可以借助节点t走捷径。节点5没有邻接点，因此不更新，如下图所示。



（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。

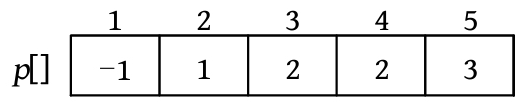


（13）加入集合S中。将节点4加入集合S={1,2,3,5,4}中，同时更新集合V−S={ }，如下图所示。



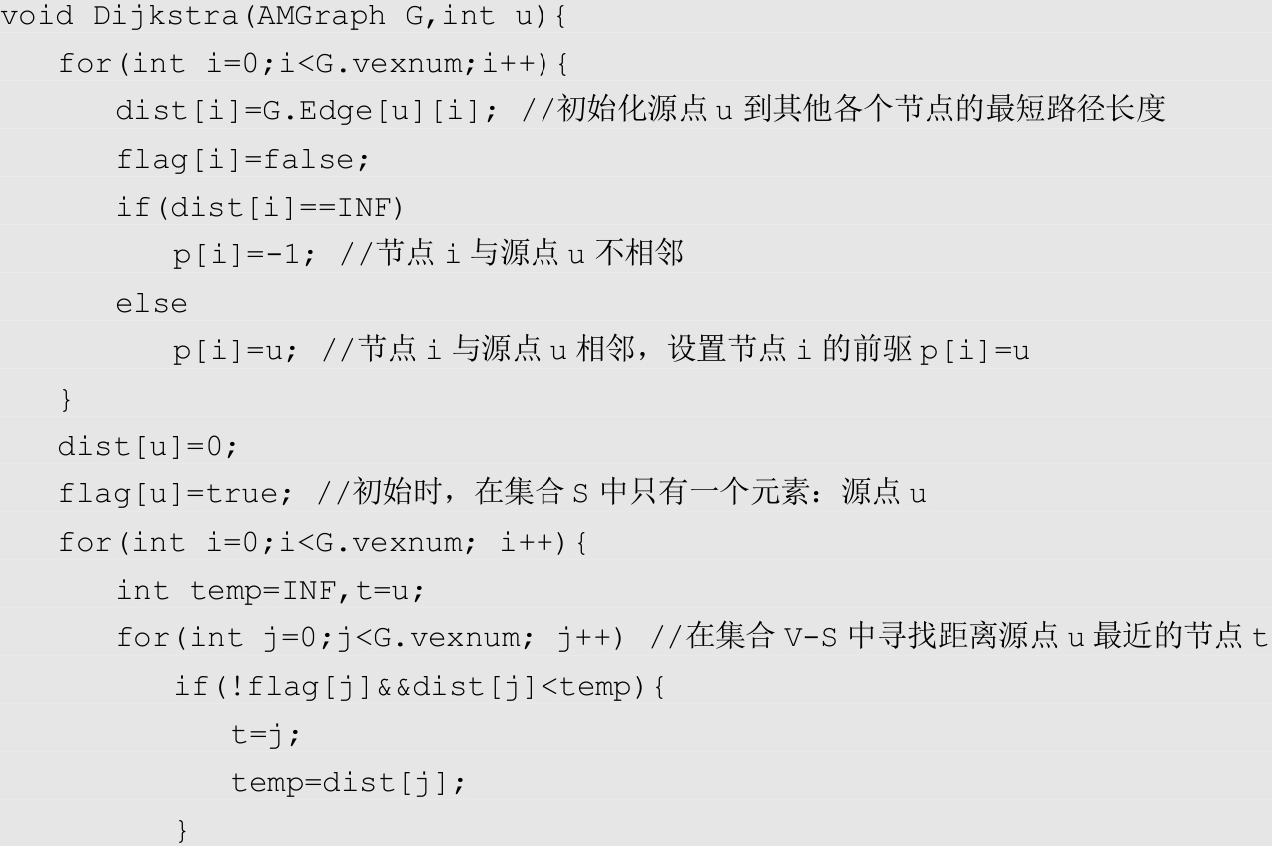
（14）算法结束。在集合V−S={ }为空时算法停止。

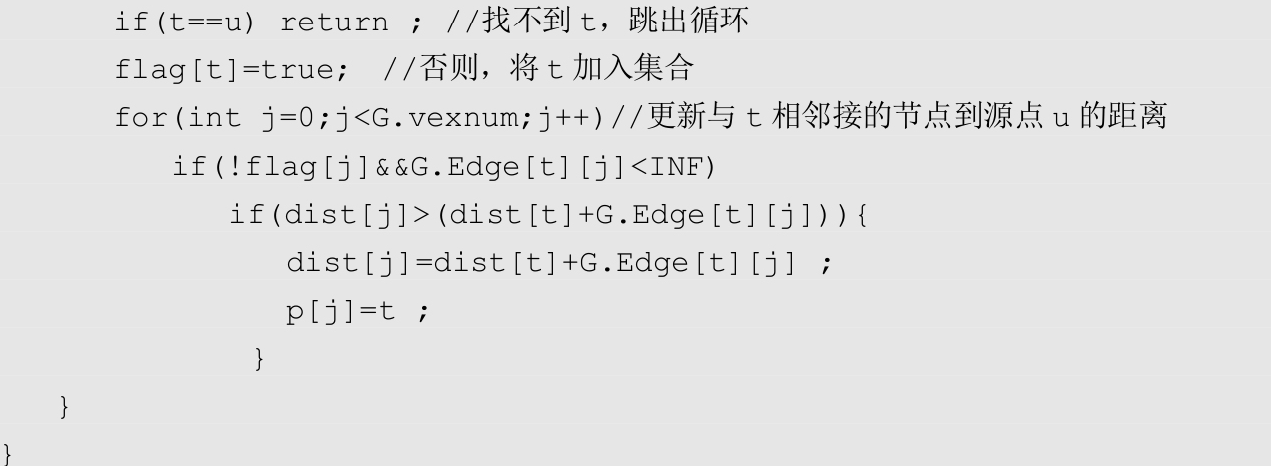
由此，可求得从源点u到图G的其余各个节点的最短路径及长度，也可通过前驱数组p[ ]逆向找到最短路径上的节点，如下图所示。



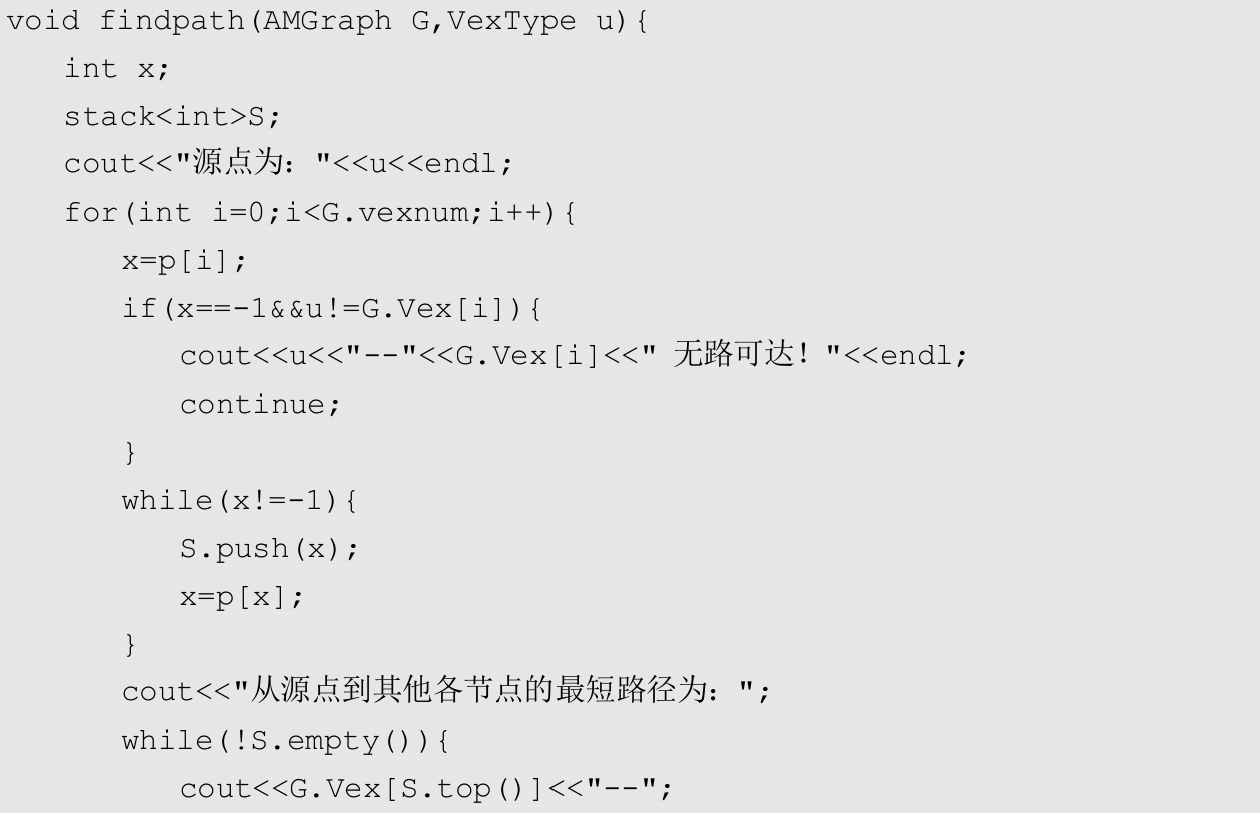
例如，p[5]=3，即节点5的前驱是节点3；p[3]=2，即节点3的前驱是节点2；p[2]=1，即节点2的前驱是节点1；p[1]= −1，节点1没有前驱，那么从源点1到5的最短路径为1-2-3-5。

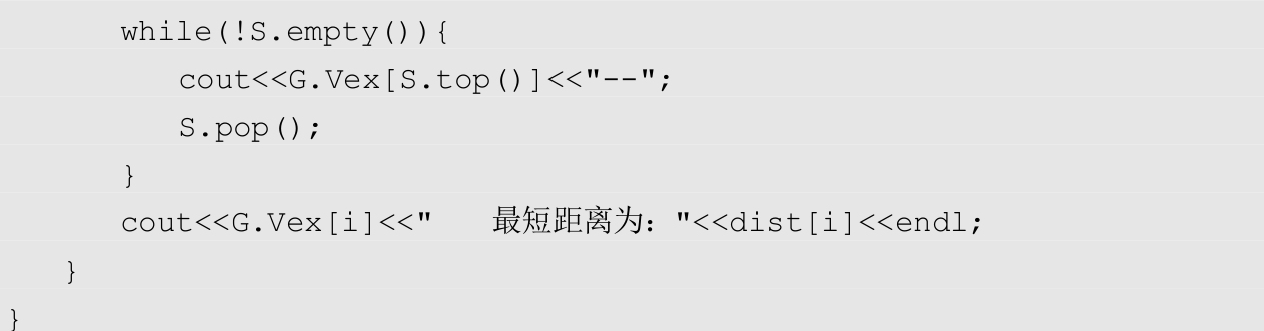
**3. 算法实现**





**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。





**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n2，算法的时间复杂度为**O(n2)**。

**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(n2)。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(n2)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**

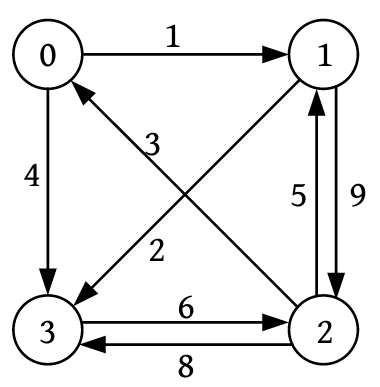
（1）数据结构。设置地图的带权邻接矩阵为G.Edge[ ][ ]，即如果从节点i到节点j有边，则G.Edge[i][j]=<i,j>的权值，否则G.Edge[i][j]=∞（无穷大）；采用两个辅助数组：最短距离数组dist[i][j]，记录从节点i到节点j的最短路径长度；前驱数组p[i][j]，记录从节点i到节点j的最短路径上节点j的前驱。

（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**

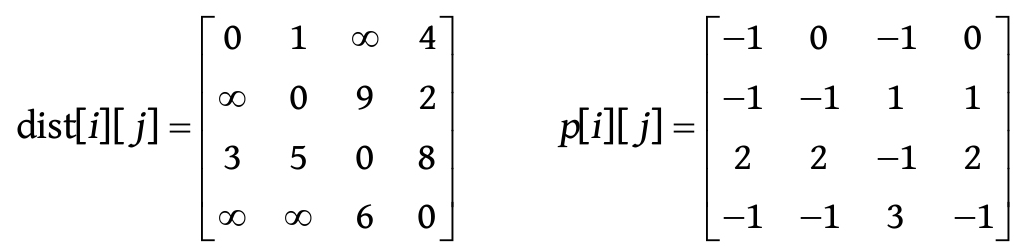
有一个景点地图，如下图所示，假设从节点0出发，求各个节点之间的最短路径。



（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）。



（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。

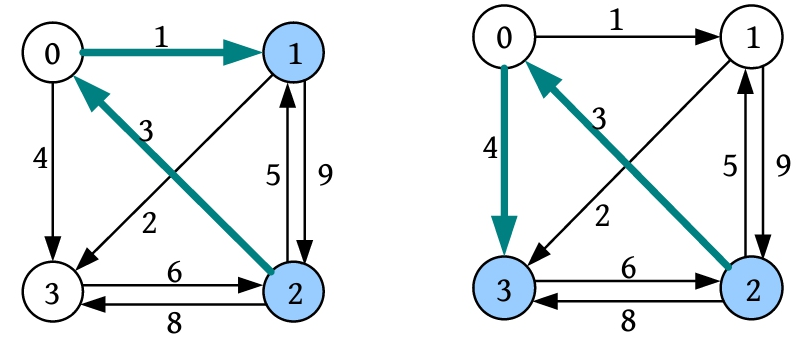


（3）插点（k=0）。其实就是“借点、借东风”，考查所有节点是否可以借助节点0更新最短距离。如果dist[i][j]>dist[i][0]+dist[0][j]，则dist[i][j]=dist[i][0]+dist[0][j]，记录节点j的前驱为p[i][j]=p[0][j]。谁可以借节点0呢？看节点0的入边2-0，也就是说节点2可以借节点0，更新2到其他节点的最短距离（在程序中需要穷举所有节点是否可以借助节点0）。

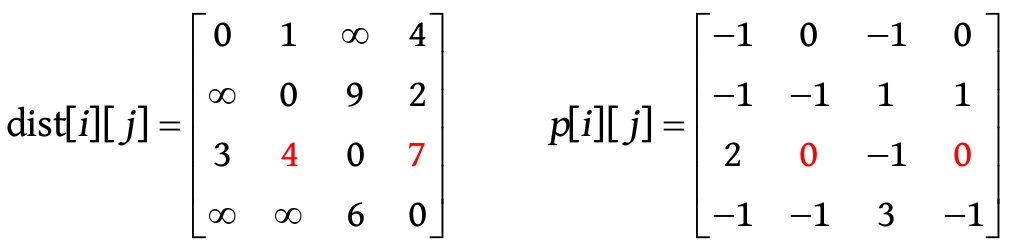
• dist[2][1]：dist[2][1]=5>dist[2][0]+dist[0][1]=4，更新dist[2][1]=4，p[2][1]=0。在节点2、1之间插入节点0。

• 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。



（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。

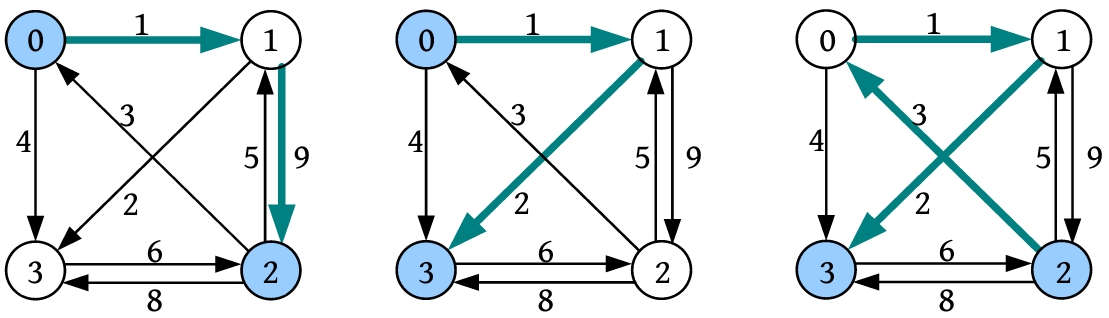
• dist[0][2]：dist[0][2]=∞>dist[0][1]+dist[1][2]=10，更新dist[0][2]=10，p[0][2]=1。在节点0、2之间插入节点1。

• 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。

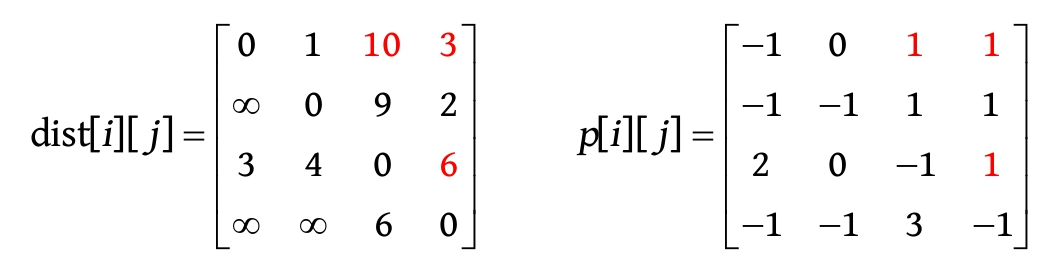
• dist[2][0]：dist[2][0]=3<dist[2][1]+dist[1][0]=∞，不更新。

• 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。



（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。

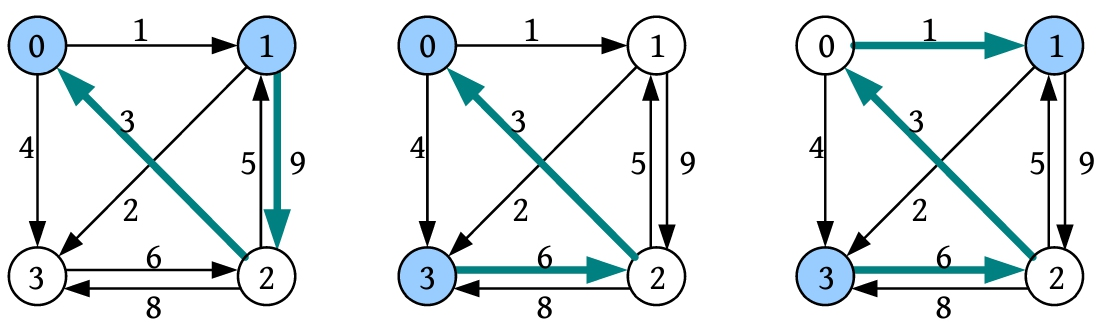
• dist[1][0]：dist[1][0]=∞>dist[1][2]+dist[2][0]=12，更新dist[1][0]=12，p[1][0]=2。在节点1、0之间插入节点2。

• dist[1][3]：dist[1][3]=2<dist[1][2]+dist[2][3]=15，不更新。

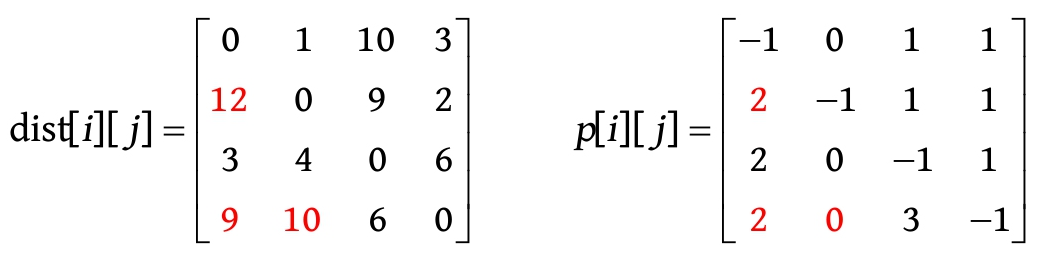
• dist[3][0]：dist[3][0]=∞>dist[3][2]+dist[2][1]=9，更新dist[3][0]=9，p[3][0]=2。在节点3、0之间插入节点2。

• 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。



（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。

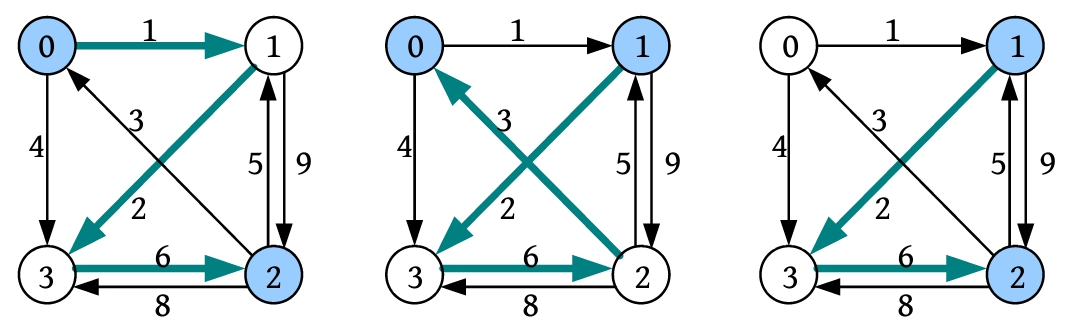
• dist[1][0]：dist[1][0]=12>dist[1][3]+dist[3][0]=11，更新dist[1][0]=11，p[1][0]=p[3][0]= 2。在节点1、0之间插入节点3。

• 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。

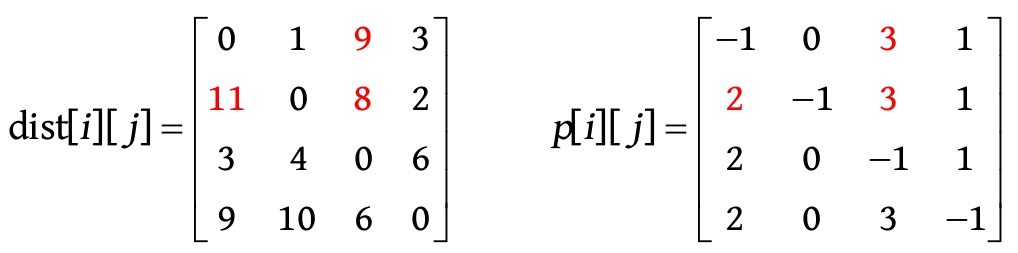
• dist[2][0]：dist[2][0]=3<dist[2][3]+dist[3][0]=15，不更新。

• 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。

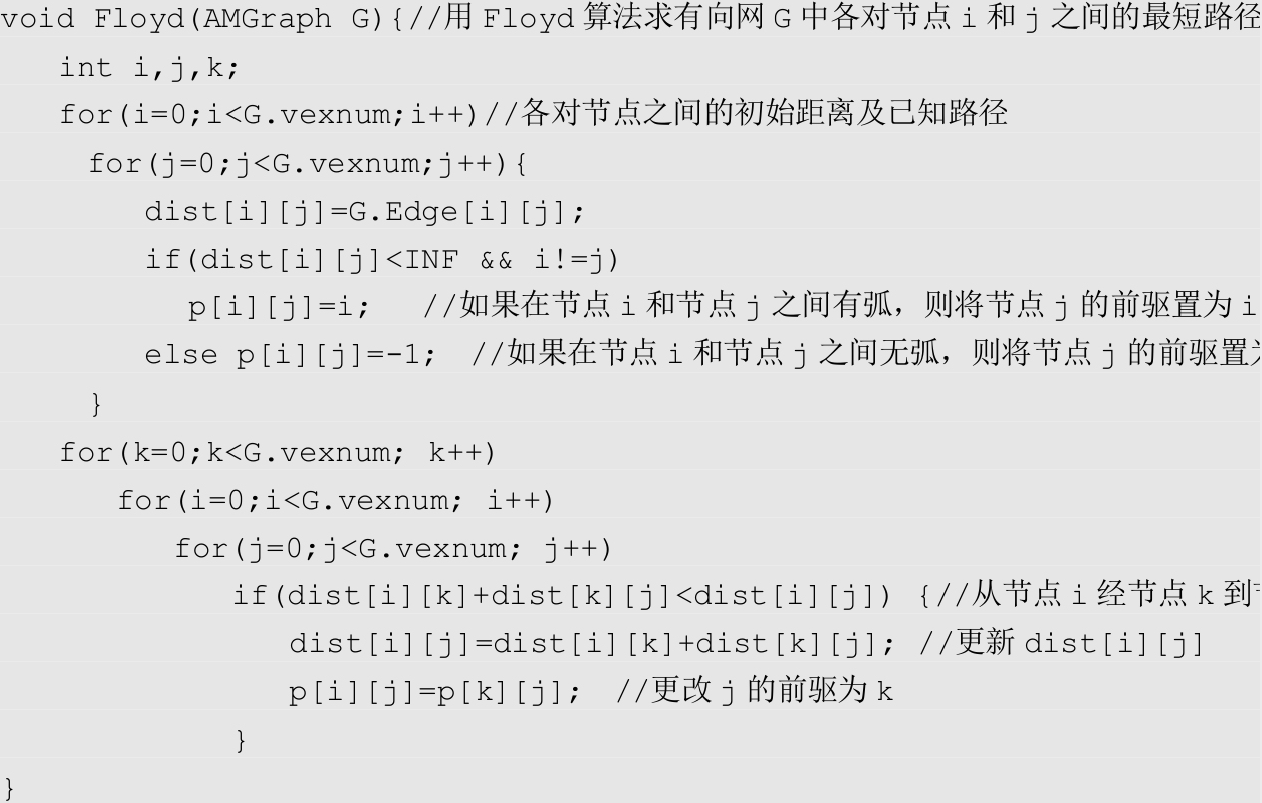
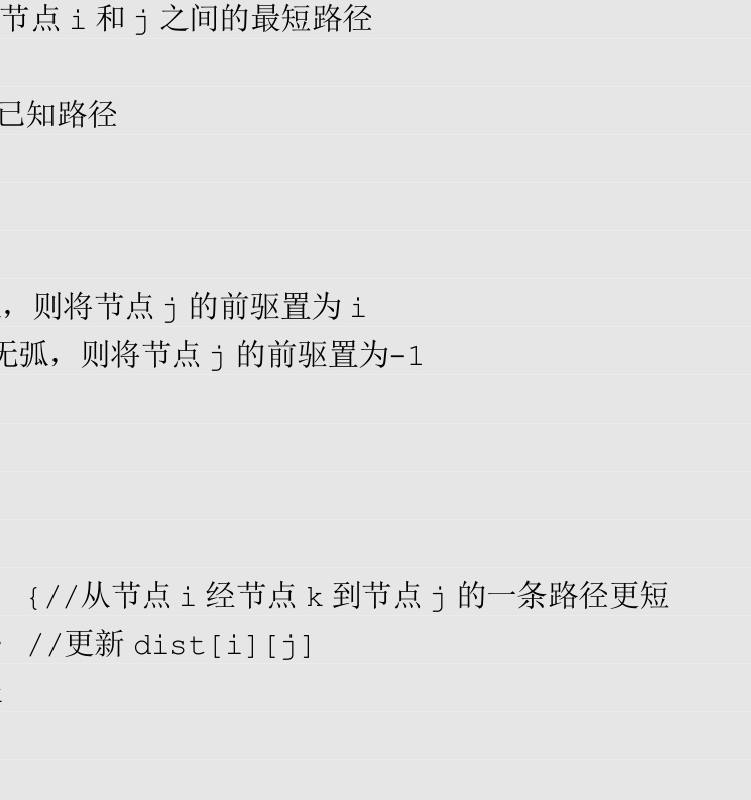


更新后的最短距离数组和前驱数组如下图所示。



（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**



**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(n3)，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(n3)。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**

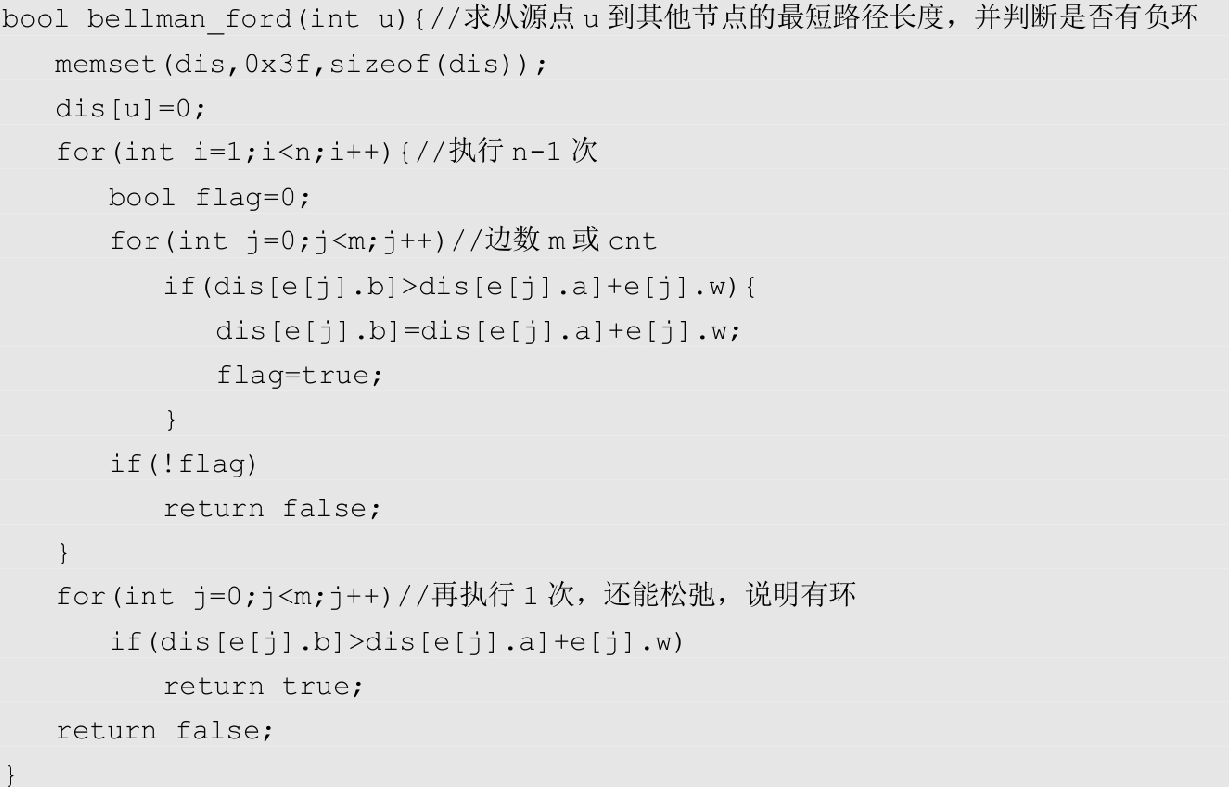
（1）数据结构。因为需要利用边进行松弛，因此采用边集数组存储。每条边都有三个域：两个端点a、b和边权w。

（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**



**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**。

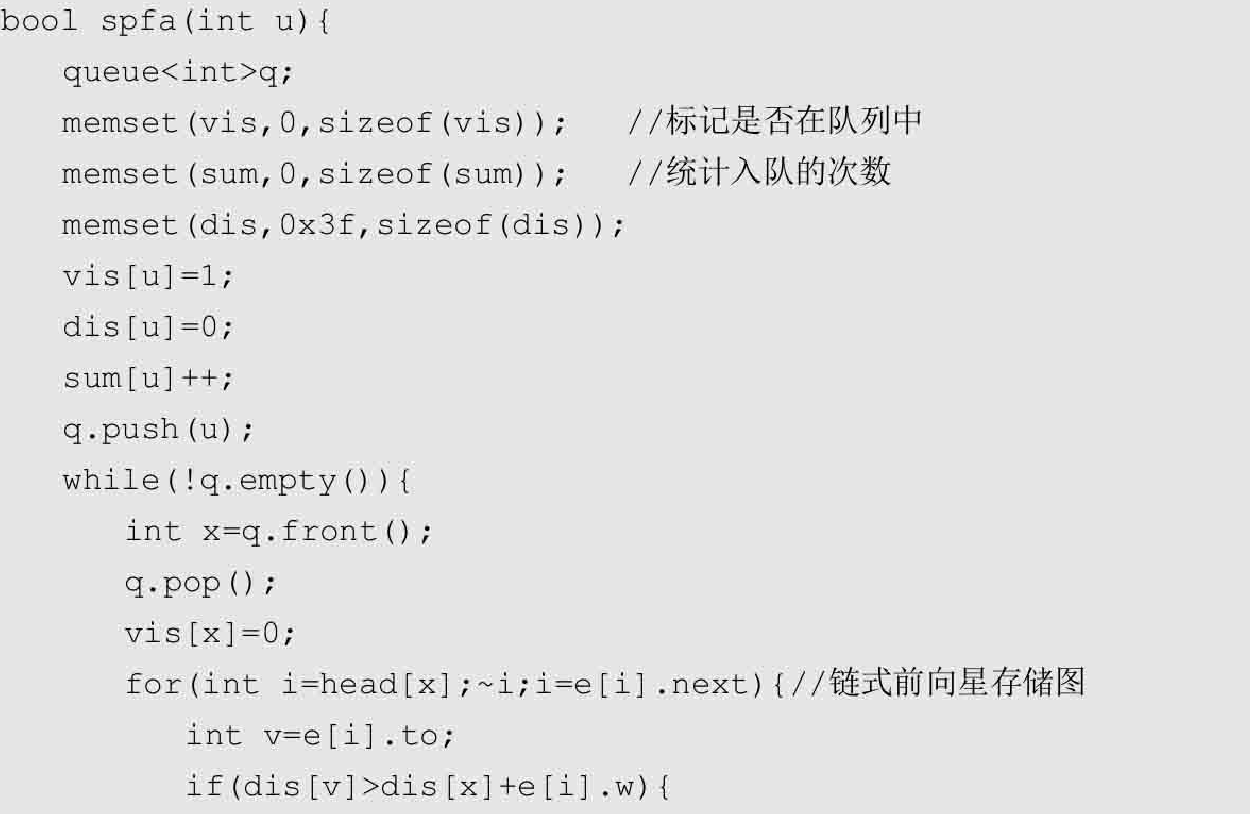
**1. 算法步骤**

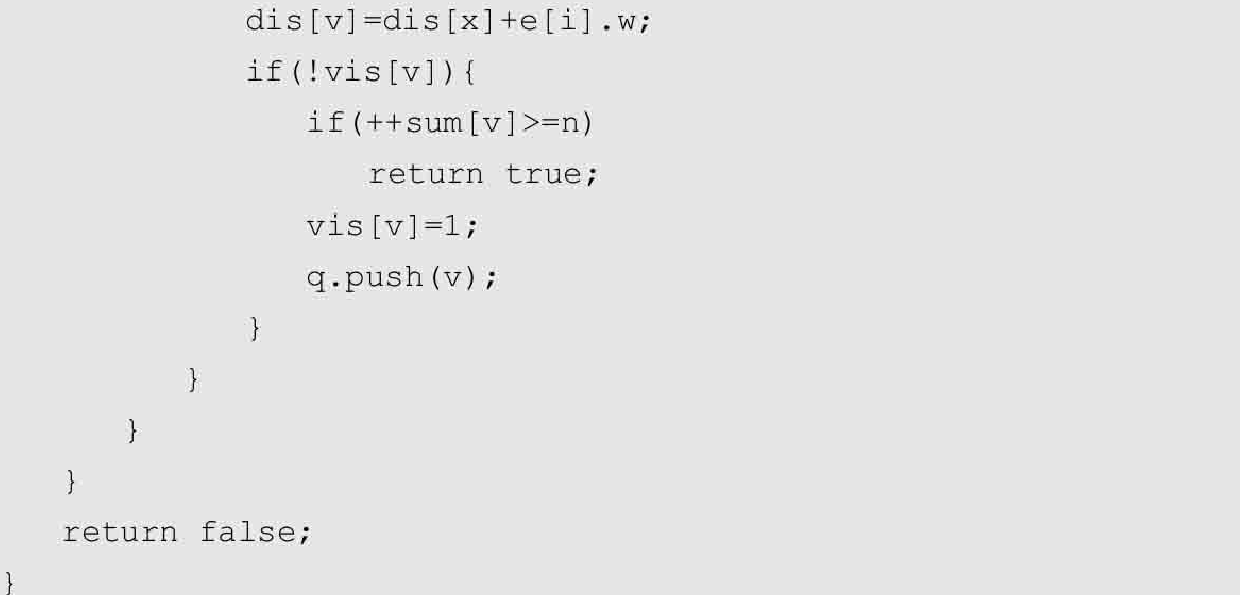
（1）创建一个队列，首先源点u入队，标记u在队列中，u的入队次数加1。

（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**





**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)**。