第七节：最短路径问题

7.1 概述

最短路径问题的抽象：

在网络中，求两个不同顶点之间的所有路径中，边的权值之和最小的那一条路径。这条路径就是两点之间的最短路径。第一个顶点称为**源点**，最后一个顶点为**终点**。

问题分类：

（1）单源最短路径问题：从某固定源点触发，求其到所有其他顶点的最短路径。又分为（有无向）有权图和（有无向）无权图两种。

（2）多源最短路径问题：求任意两顶点间的最短路径。

7.2 无权图的单源最短路

按照递增（非递减）的顺序找出到各个顶点的最短路。例如图 1：

图 1

我们以 v3 作为源点，与 v3 距离为 0 的点为 v3；与 v3 距离为 1 的点为 v1 和 v6；与 v3 距离为 2 的点（也就是距离 v1 为 1，距离 v6 为 1）的点为 v2 和 v4；距离 v3 距离为 3 的点（也就是距离 v2 为 1，距离 v4 为 1）的点为 v5 和 v7。至此，距离各点距离 v3 都已经清楚。

这个算法类似于 BFS 算法。但是与 BFS 不同的是，我们不需要定义一个已经访问的变量，需要重新定义一个数组为

（1）dist[W]=S 到 W 的最短距离。初始情况下，默认距离可以为-1、负无穷或者正无穷。

（2）dist[S]=0

（3）path[W]=S 到 W 的路上经过的某顶点。

代码如下：

```
void Unweighted(Vertex S)
{
    Enqueue(S,Q);
    while(!IsEmpty(Q)){
        V = Dequeue(Q);
        for(V 的每个邻接点 W)
            if(dist[W]==-1){
                dist[W]=dist[V]+1;
                path[W]=V;
                Enqueue(W,Q);
            }
    }
}
```

path 这个数组的作用就是记录从 S 到 W 路径中所经过的点，这个算法就是利用刚刚讲的那个每次基于前面那个点加 1 来得到下面一个点的这个方法。

时间复杂度为 **T=O(|V|+|E|)**。看这个代码中 while 的部分，由于它都 Dequeue 和 Enqueue 了一次，所以总的次数为 2V；又由于 for 循环遍历了 V 的每个邻接点 W，个数为 E，所以时间复杂度是 O(|V|+|E|)。

7.3 无权图的单源最短路示例

这里，我们以图 1 中的图来做示例，仍将 v3 看做是源点，因此，dist 和 path 做以下初

始化:

下标 W	1	2	3	4	5	6	7
dist	-1	-1	0	-1	-1	-1	-1
path	-1	-1	-1	-1	-1	-1	-1

调用 Unweighted(3)，然后将 v3 入队列。然后 while 循环，判断队列不为空，让 v3 出队列给 V。然后开始 for 循环，对于 V（即 v3）的每个邻接点 W，如果这个点没有访问过（即该点的 dist[W]==-1），则将该点的 dist 赋值为前一点的 dist+1（刚开始的时候前一点的 dist 为 0），然后把前一点（刚开始是 v3）赋值给 path 的第 w 个值，然后把下一个点压入队列。然后把 v1 压出，开始 for 循环，对于 V（即 v1）的每个邻接点，再进行上面的操作，如此循环直到最后。到最后的时候，dist 和 path 数值为:

下标 W	1	2	3	4	5	6	7
dist	1	2	0	2	3	1	3
path	3	1	-1	1	2	3	4

具体代码:

```
/* 邻接表存储 - 无权图的单源最短路算法 */

/* dist[]和 path[]全部初始化为-1 */
void Unweighted ( LGraph Graph, int dist[], int path[], Vertex S )
{
        Queue Q;
        Vertex V;
        PtrToAdjVNode W;

        Q = CreateQueue( Graph->Nv ); /* 创建空队列, MaxSize 为外部定义的常数 */
        dist[S] = 0; /* 初始化源点 */
        AddQ (Q, S);

        while( !IsEmpty(Q) ){
                V = DeleteQ(Q);
                for ( W=Graph->G[V].FirstEdge; W; W=W->Next ) /* 对 V 的每个邻接点 W->AdjV
*/
                        if ( dist[W->AdjV]==-1 ) { /* 若 W->AdjV 未被访问过 */
                                dist[W->AdjV] = dist[V]+1; /* W->AdjV 到 S 的距离更新 */
                                path[W->AdjV] = V; /* 将 V 记录在 S 到 W->AdjV 的路径上 */
                                AddQ(Q, W->AdjV);
                        }
        } /* while 结束*/
}
```

7.4 有权图的单源最短路

仍然是按照递增的顺序找出到各个顶点的最短路。解决这个问题的经典算法就是: **Dijkstra 算法**。

（1）Dijkstra 算法

①令 S={源点 s+已经确定了最短路径的顶点 vi}

②对任一未收录的顶点 v，定义 dist[v]为 s 到 v 的最短路径长度，但该路径**仅经过 S 中**

的顶点，即路径{s->(vi∈S)->v}的最小长度。

若路径是按照递增（非递减）的顺序生成的，则：

真正的最短路径必须只经过 s 中的顶点；

每次从未收录的顶点中选一个 dist 最小的收录（贪心算法）

增加一个 v 进入 S，可能影响另外一个 w 的 dist 值（**若使 w 的 dist 值缩小，则 v 必定在 s 到 w 的路径上，并且 v 到 w 有一条边**），dist[w]=min{dist[w],dist[v]+<v,w>的权重}。代码如下：

```
void Dijkstra(Vertex s)
{
    while(1){
        V=未收录顶点中 dist 最小者;
        if(这样的 V 不存在)
            break;
        collected[V] = true;
        for(V 的每个邻接点 W)
        if(collected[W] == false)
            if(dist[V]+E<k,w> < dist[W])
            {
                dist[W] = dist[V] +E<v,w>;
                path[W] =V;
            }
    }
}
```

注意：该算法不能解决有负边的情况。

若只看这段伪码，我们无法判断其复杂度，这主要是因为**未收录顶点中 dist 最小者**的这个算法的复杂度我们无法判断。这个算法有以下几种：

①直接扫描所有未收录顶点-O(|V|)，则 **T=$O(|V|^2+|E|)$**。**对稠密图效果好**。

②将 dist 存在最小堆中-O(log|V|)。更新 dist[w]的值-O(log|V|)。总体复杂度为 **T**=O(|V|log|V|+|E|log|V|)=**O(|E|log|V|)**。**对稀疏图效果好**。

7.5 有权图的单源最短路示例

图如图 2 所示。

图 2

这里我们以 v1 位源点。dist 和 path 初始化结果如下所示：

下标 W	1	2	3	4	5	6	7
dist	**0**	**2**	无穷	**1**	无穷	无穷	无穷
path	-1	**1**	-1	**1**	-1	-1	-1

当首先访问 v4 的时候，结果如下

下标 W	1	2	3	4	5	6	7
dist	**0**	**2**	3	**1**	3	9	5
path	-1	**1**	4	**1**	4	4	4

然后看 v2，由于 v4 已经访问过了，因此接下来看 v5，由于到 v5 的距离为 12，大于前面的距离，因此跳出循环。由于 v2 的邻接点只有 v4 和和 v5，因此跳出 v2。

接下来看 v3 和 v5 距离一样，那么就先看 v3。v3 的邻接点有 v1 和 v6，v1 已经被访问

过，所以无视，直接看 v6，此时可以看出 v3 到 v6 距离为 5，所以总距离为 8，小于原来的 9。因此更新 dis[6]=9，path[6]=3。然后看 v5，v5 只有一个邻接点，就是 v7，这样经过 v5 到 v7 的距离为 9，大于原来的距离，不更新。

在 v6 和 v7 中，v7 的距离更短一点，所以先看 v7。v7 的邻接点只有 v6。若经过 v7 访问 v6 的距离为 6，小于原来的 8，更新 dis[6]=6，path[6]=7。

由于 v6 没有任何邻接点，所以 v6 直接不做判断。最后更新的结果如下：

下标 W	1	2	3	4	5	6	7
dist	**0**	**2**	3	**1**	3	**6**	5
path	-1	**1**	4	**1**	4	**7**	4

7.6 多源最短路算法

方法 1：直接将单源最短路算法调用|V|遍，$T=O(|V|^3+|E|\times|V|)$，适合稀疏图。

方法 2：**Floyd 算法**，复杂度 $T=O(|V|^3)$，更适合稠密图。

（1）Floyd 算法

①D^k[i][j]=路径{i->{l<=k}->j}的最小长度。

②$D^0,D^1,\cdots,D^{|V|-1}$[i][j]即给出了 i 到 j 的真正最短距离。

③最初的 D^{-1} 是**带权的邻接矩阵，对角元素是 0**。

④若 i 和 j 之间没有直接的边，D[i][j]定义为**正无穷**。

⑤当 D^{k-1} 已经完成，递推到 D^k 时：

或者 k∉最短路径{i->{l<=k}->j}，则 $D^k=D^{k-1}$；

或者 k∈{i->{l<=k}->j}，则该路径必定**由两段最短距离组成**：$D^k[i][j]= D^{k-1}[i][k]+ D^{k-1}[k][j]$。

源代码为：

```
void Floyd()
{
	for(i=0;i<N;i++)
		for(j=0lj<N;j++){
			D[i][j]=G[i][j];
			path[i][j]=-1;
		}
		for(k=0;k<N;k++)
			for(i=0;i<N;i++)
				if(D[i][k]+D[k][j]<D[i][j])
				{
					D[i][j]=D[i][k]+D[k][j];
					path[i][j]=k;
				}
}
```