快速 Dijkstra 最短路径优化算法的实现

司连法[1]，王文静[2]

（1. 中国地图出版社，北京 100054；2. 武汉大学，湖北 武汉 430079）

Realization of Optimal Algorithm for Fast Dijkstra Latest Path

SI Lian-fa，WANG Wen-jing

摘要：在分析已有 Dijkstra 算法的基础上，提出快速 Dijkstra 最短路径优化算法。该算法是将提高时间效率放在第一位，以十字链表结构记录顶点（Vertex）和边（Edge）为基础，采用顶点分区和记录绝对地址来优化 Dijkstra 算法的方法。

关键词：Dijkstra 算法；GIS；最短路径；优化

一、最短路径与 Dijkstra 算法

随着计算机和地理信息科学的发展，GIS（地理信息系统）的应用领域越来越广。最短路径问题作为地理学中一个重要问题，其在 GIS 中的应用无疑占有重要的地位，无论是交通运输，还是城市规划、物流管理、网络通讯等方面，它都发挥了重要的作用。因此，对它的研究不但具有重要的理论价值，而且具有重要的应用价值。

研究最短路径问题通常将它们抽象为图论意义下的网络问题，问题的核心就变成了网络图中的最短路径问题。当然，此时的最短路径不单指“纯距离”意义上的最短路径，它可以是“经济距离”意义上的最短路径，“时间”意义上的最短路径，“网络”意义上的最短路径。

关于最短路径问题，目前所公认的最好的求解方法，是 1959 年由 E.W.Dijkstra 提出的标号法，也就是人们常说的以他的名字命名的 Dijkstra 算法。在此算法思想基础上，人们演绎出了几十种不同的优化算法，效果较好的算法有 TQQ（graph growth with two queues），DKA（the Dijkstra's algorithm implemented with approximate buckets），DKD（the Dijkstra's algorithm implemented with double buckets）[1]，排序优化算法[2]等，前面 3 种算法将空间存储问题放在了一个重要的位置，以牺牲适当的时间效率来换取空间节省，排序优化算法将时间放在了第一位，在提高时间效率方面有较好的应用性。

本文提出的快速 Dijkstra 最短路径优化算法是将提高时间效率放在第一位，以十字链表结构记录顶点（Vertex）和边（Edge）为基础，采用顶点分区和记录绝对地址来优化 Dijkstra 算法的方法。

设 $G=(V,E)$ 是一个赋权有向图，即对于图中的每一条边 $e=(v_i,v_j)$ 都赋予了一个权值 w_{ij}，$D(v_j)$ 表示点 v_1 和 v_j 之间的最短距离。在图 G 中指定一个顶点，确定为起点，不妨设 v_1 为起点。

Dijkstra 算法的基本思想是：

1. 开始，先给 v_1 标上红色标号，且 $D(v_1)=0$，其余各点 $D(v_j)=+\infty\ (j\neq 1)$，红点是不可改变 D 值的点，它所具有的 D 值即为该点至起始点的最短距离；

2. 如果刚刚得到红色标号的点是 v_i，将从该点进行扩展蓝点集，那么，对于所有这样的点 $\{v_j\mid(v_i,v_j)\in E$，而且 v_j 的标号不是红色标号$\}$，将其 $D(v_j)$ 修改为：$\min[D(v_j),D(v_i)+w_{ij}]$，并将 v_j 标上蓝色标号，蓝点是具有至起始点临时最短距离的点；

3. 从所有标有蓝色标号的顶点中求得 $D(v_i)$ 最小的顶点 v_i，并将其标上红色标号，转向第 2 步，如果此步没有发现标有蓝点标号的顶点，则停止。

二、现有优化算法分析

TQQ，DKA，DKD 算法是以节约存储空间为基础的算法，从当前计算机硬件发展水平来看，它们已经失去了优化的意义。本文只针对节约时间的优化算法作分析。

排序优化算法是一种较好的算法。为了较好地分析它，在此对它作一简述。该算法是针对 Dijkstra 算法中的步骤 3 做的优化。按照 Dijkstra 算法来计算，从未标记为红点的顶点中选择一个 $D(v_i)$ 最小

收稿日期：2004-09-09

作者简介：司连法（1973-），男，河南商丘人，工程师，现从事电子地图、GIS 程序开发以及地理学的研究。

的点,如果不采用任何技巧,未标记点将以无序的形式存放在一个链表或数组中。那么要选择一个$D(v_i)$最小的顶点就必须把所有的点都扫描一遍,在大数据量的情况下,这无疑是一个制约计算速度的瓶颈。为了解决这一问题,排序优化算法将这些要扫描的点按其 D 值进行顺序排列,这样每循环一次即可取得符合条件的点,可提高算法的执行效率。

针对这一算法,我们可以看出它在 Dijkstra 算法中步骤 2 中每标记一个新的蓝点或改变一个已有蓝点的 $D(v_i)$时,将对所有顶点进行排序,或者说对所有未标记点进行排序。通过较快的排序方法对未标记点进行排序后,步骤 3 中就不用再循环查找$D(v_i)$最小的顶点 v_i,而是直接使用已排序后的$D(v_i)$最小的顶点。这在一定程度上节约了时间,也取得了较理想的效果。

但是我们可以从另一方面来分析该算法,它存在一定的缺点。假设在步骤 2 中刚刚得到红色标号的顶点 v_i 有 4 个相连接的顶点,而且 4 个顶点都是需要新标上蓝点的顶点,这时,在一个 2 ~ 3 循环步骤中有 4 次排序操作,而原 Dijkstra 算法中 2 ~ 3 循环步骤中只有 1 次查询过程,这样一比较,优化算法的优势就不明显了。

为了解决实际问题,还有采用带状区域求解最短路径的算法[3]。该算法是针对求两点间最短路径问题而设计的。它根据始末点划定一个带状区域,只有区域内顶点参与计算,大大提高了计算的速度,但是它不具有通用性,而且带状区域大小的设定也是一个问题。

三、Dijkstra 算法中数据存储结构的比较

Dijkstra 算法的基础是图,在计算机中如何对图进行存储表示,也直接影响着算法的计算效率。无向图可以用邻接矩阵和邻接多重表表示,而有向图则可以用邻接表和十字链表表示,其优缺点的比较见表 1。

表 1　图的几种存储结构的比较

名称	存储方法	优点	缺点	时间复杂度
邻接矩阵	2 维数组	易判断两点间关系	占用空间大	$O(n^2+m\times n)$
邻接多重表	链表	易判断两点间关系	结构较复杂	$O(n+m)$或 $O(n\times m)$
邻接表	链表	节省空间,易得到顶点的出度	不易判断两点间的关系,不易得到顶点入度	$O(n+m)$或 $O(n\times m)$
十字链表	链表	节省空间,易求得顶点的出度和入度	结构较复杂	$O(n+m)$或 $O(n\times m)$

四、快速 Dijkstra 最短路径优化算法的实现

本算法的思想是通过减少查找具有最小 $D(v_i)$的顶点的时间,从而达到提高算法时间效率的目的。

1. 拓扑关系的建立

根据对图的几种存储结构的比较,本算法采用了十字链表的存储结构。该结构不但能够清晰地表示顶点与其相应边的关系,而且在实现 Dijkstra 算法过程中可以节约许多结构处理上的时间。

首先,定义顶点为 CVertex 类,边为 CEdge 类,定义顶点列表 CVertexList 和 CEdgeList,在顶点中存储了与其相关的边的列表,边中存储了与其相关的首末顶点。在结构中存储的顶点或边均为指针。这种十字链表结构就构成了顶点与边的拓扑关系。

为了计算和记录最短路径,还必须在边中定义边的长度,在顶点中定义前一顶点及该点至起点的最短距离。

2. 算法实现

本算法主要采用了顶点分区只计算“单层洋葱皮上顶点”方法和记录绝对地址来减少查询时间的方法。

本算法将所有顶点分为 3 个区:红点集区、蓝点集区、白点集区。在顶点列表中,蓝点集区位于最前端,白点集区位于中央,红点集区位于末端。在计算开始时,除了起始点为红点外,其他顶点均为白点。计算过程中,当蓝点变为红点时,将该顶点移至顶点列表末端,当白点变成蓝点时,将该顶点移至顶点列表最前端。这样,在白点集顶点不断变成蓝点的过程中,蓝点集顶点也在不断地变成红点,因此,蓝点集顶点数的增长不会太快,其总点数也是有限的。如果将所有点看作为一颗洋葱,红点集就是洋葱的心,白点集是洋葱的外皮,而蓝点集是洋葱中间的一层。图 1 形象地表示了其特点。

由此看来,查找具有最小 $D(v_i)$的顶点时可以一次性排除占绝大多数的红点集和白点集顶点,而

只需在“单层洋葱皮”上蓝点集中进行，这样可以大大节省时间，提高了算法的时间效率。

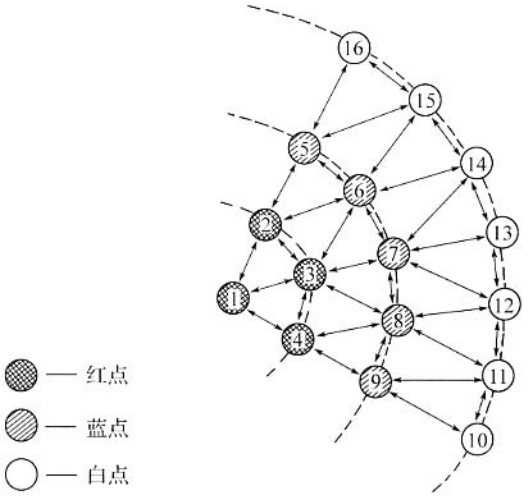

图 1　红点集、蓝点集、白点集标号变化示意图

由于此算法中有一个移动列表中顶点的过程，此过程在实现时必须在列表中找到需要移动的顶点，这一查找过程可以自己编写函数来完成，也可以通过 VC++ 中系统函数来完成，但是不管通过哪种方法，在上万个顶点的列表中查找也是较费时的工作。为了使这一工作不影响时间效率，本算法在顶点的属性中增加了一个位置变量，用来记录该顶点在列表中的位置。这样就不用在顶点列表中查找顶点位置，而是通过直接调用绝对地址来完成顶点的定位。

记录顶点绝对地址的方法，在 2 万多个顶点的运算中使运算效率提高了上百倍，而且，点数越多，效果越明显。

本算法的实现函数如下：

```
void Dijkstra(CVertexList m_VertexList,CEdgeList m_EdgeList,Cvertex * pVertex)
{
    CVertex * pVertexMin = pVertex;
    //将起始红点移至列表末端
    m_VertexList.RemoveAt(pVertexMin->POS);//使用绝对地址
    m_VertexList.AddTail(pVertexMin);
    pVertexMin->POS = m_VertexList.GetTailPosition();//调整绝对地址
    for(int I = 0;I < n - 1;I ++)
    {//处理与 pVertexMin 相关联边的另一顶点
        POSITION pos1 = pVertexMin->m_LinkEdgeList.GetHeadPosition();
        while(pos1 != NULL)
        {
            CEdge * pEdge = (CEdge *)pVertexMin->m_LinkEdgeList.GetNext(pos1);
            //如果 pEdge 的另一顶点是红点，返回；如果是白点，改为蓝点，移至列表最前端，并令 D(v_j) = D(v_i) + w_ij；如果是蓝点，令 D(v_j) = min[D(v_j),D(v_i) + w_ij]。
            ...
        }
        //在当前蓝点集中选估计距离最小的顶点 pVertexMin
        pos1 = m_VertexList.GetHeadPosition();
        POSITION POSRESULT = pos1;
        pVertexMin = (CVertex *)m_VertexList.GetNext(pos1);
        if(pVertexMin->m_bUsed == true || pVertexMin->m_dShortestLength == -1)
            return;//如果列表最前端顶点为红点或白点，循环结束
        //在蓝点集中查找最小点
        POSITION postemp = pos1;
        while(pos1 != NULL)
        {
            ...
            if(pVertex->m_bUsed == true || pVertex->m_dShortestLength == -1)
                break;
            ...
        }
        //将此点移至最后并调整绝对地址
        m_VertexList.RemoveAt(POSRESULT);//使用绝对地址
        m_VertexList.AddTail(pVertexMin);
        pVertexMin->POS = m_VertexList.GetTailPosition();
        pVertexMin->m_bUsed = true;//加入红点集
        if(pVertexMin->m_bDestination)
            break;//找到终点，该句用于求两点间最短距离
    }
}
```

通过上述算法可以看出，程序循环次数为 $n-1$，

每次计算边的次数为 c，每次查找最小点的循环次数为从 1 增大，平均值约为$\sqrt{n}$，因此算法的时间复杂度为 $O((n-1)\sqrt{n})$。

根据本算法，笔者用 Visual C^{++}6 对其运算效率进行了测试，以 22 500 个顶点、25 600 条边为例，在 PIII800、内存为 128 MB 的机器上，计算将所有点包含在内的路径，仅用 0.13 s。另外，如果在超大规模点集中运用此算法，还可以采用结合对蓝点集排序的优化算法。

参考文献：

[1] ZHAN F B. Three Fastest Shortest Path Algorithms on Real Road Networds[J]. Journal of Geographic Information and Decision Analysis, 1997, 1(1): 69-82.

[2] YUE Yang. An Efficient Implementation of Shortest Path Algorithm Based on Dijkstra Algorithm[J]. Journal of Wuhan Technical University of Surveying and Mapping, 1999, 24(3): 209-212.

[3] BAO Pei-ming. A Optimization Algorithm Based on Dijkstra's Algorithm in Search of Shortcut[J]. Journal of Computer Research and Development, 2001, 38(3): 307-311.