

带限制条件的最短路径算法与实现

林小玲, 何建农, 周勇

(福州大学数学与计算机科学学院, 福建 福州 350002)

摘要: 给出了在 GIS 环境下带限制条件的单源最短路径算法, 该算法是基于二叉堆优先级队列及邻接表的 Dijkstra 算法. 根据用户给出的起始节点和目标节点以及避开节点列和必经节点列, 在建立的搜索图中用 Java 语言实现分段查找最短路径.

关键词: 单源最短路径; Dijkstra 算法; 二叉堆; 限制条件

中图分类号: TP311

文献标识码: A

The approach to the shortest path algorithms with restrictive conditions

LIN Xiao - ling, HE Jian - nong, ZHOU Yong

(College of Mathematics and Computer Science, Fuzhou University, Fuzhou, Fujian 350002, China)

Abstract: In this article, a Dijkstra algorithm with restritive conditions is proposed in GIS environment on the basis of binary - heap priority queue and adjacent list. It can find a shortest path on search graph according to start node, goal node, avoidable and passing node array given by user. Finally, this algorithm is implemented by Java language.

Keywords: single - source shortest path; Dijkstra algorithm; binary - heap; restrictive condition

最短路径问题(SP)作为交通网络分析的一个最基本的问题, 是许多领域中选择最优问题的基础, 特别是在现代化的智能交通系统中占有重要地位. 在智能交通系统中, 给定限制条件的最短路径有着更广泛的应用. 例如, 从家到学校若必须经过某个邮局, 如何求得最短路径, 或者某处出现交通堵塞, 如何求避开该处的最短路径等等.

从有向图的某一顶点出发, 到达图中其余各顶点的最短路径称为单源最短路径问题. 这一问题已由荷兰数学家 Dijkstra 于 1959 年解决. Dijkstra 算法是目前理论上最完善的算法, 但不同的实现方式或者运行结构决定了不同的时间复杂度. 若某个交通网络图的顶点个数为 n , 则传统 Dijkstra 算法时间的复杂度 $O(n^2)$, 而基于二叉堆优先级队列的单源最短路径算法(改进的 Dijkstra 算法)时间复杂度为 $O(n \log n)$. 显然用优先级队列实现最短路径算法拥有更小的时间复杂度. 用邻接表来存储网络图, 空间复杂度仅为 $O(m + n)$, m 为网络边的个数, 而用邻接矩阵来存储, 其空间代价为 $O(n^2)$, 所以邻接表数据结构已被公认是网络表达中最有效率的数据结构.

本文首先讨论了基于二叉堆优先级队列的单源最短路径算法的基本思想及时间复杂度, 在此基础上补充了带限制条件的约束. 然后对文献[1]提出的最短路径算法进行一些修改, 解决了该算法对有些始点不能正确输出的问题. 最后采用面向对象编程语言 Java^[2]来实现.

1 单源最短路径算法分析

图可用 $G = (V, E)$ 来表示, 其中, V 是顶点集合, E 是边集合. 如果图的边限定为从一个顶点指

收稿日期: 2004-03-07

作者简介: 林小玲(1981-), 女, 硕士研究生.

基金项目: 福建省教委科技资助项目(K20019)

向另一个顶点则为有向图,若边上附有权称为有向带权图. 交通网络可以用有向带权图表示,因为交通网络存在有向性,图中顶点可表示路口,边表示两个路口之间的道路,边上的权值表示两路口间的距离、交通费用或途中所需的时间等等.

最短路径算法的运行结构多种多样,是最短路径研究的热点. 基于各种运行结构的 Dijkstra 算法仍是交通网络最短路径算法的首选,如 Arc/Info 中的 Network 就采用二叉堆优先级队列来实现 Dijkstra 算法. 文献[3]中还提出了四叉堆优先级队列及逆邻接表的 Dijkstra 算法. Dijkstra 算法是解决关于带权图(权为非负数)的单源最短路径问题的一种贪心算法,它要一个一个地按路径长度递增序找出从某个源点出发到所有其他顶点的最短路径. 单源最短路径问题就是已知有向带权图 $G = (V, E)$,找出从某个源点 $s \in V$ 到 V 中其余各顶点的最短路径. Dijkstra 算法是按长度递增的次序生成从源点 s 到其它顶点的最短路径,则当前正在生成的最短路径上除终点以外,其余顶点的最短路径均已生成(将源点的最短路径看作是已生成的源点到其自身的长度为 0 的路径).

算法的基本思想可由以下几个步骤来体现. 设 S 为最短距离已确定的顶点集(看作红点集), $V - S$ 是最短距离尚未确定的顶点集(看作蓝点集).

1) 初始化. 初始化时,只有源点 s 的最短距离是已知的($D(s) = 0$),其余各点的估计最短距离 D 值均设为无穷大. 用数组 $D[v]$ 记录从源点 s 到达 v 且除 v 外中间不经过任何蓝点(若有中间点,则必为红点)的“最短”路径长度(简称估计距离). 经过一次循环后,红点集 $S = \{s\}$,其余各点的估计最短距离 D 值更新为该点到源点而中间不经过任何点的边的权值. 若路径不存在则仍为无穷大.

2) 重复以下工作,按路径长度递增次序产生各顶点最短路径. 在当前蓝点集中选择一个最短距离最小的蓝点来扩充红点集,并置已访问标志,以保证算法按路径长度递增的次序产生各顶点的最短路径. 当蓝点集中仅剩下最短距离为无穷大的蓝点,或者目标节点已扩充到红点集时, s 到目标节点的最短路径就求出来了. 该算法同时还把源点到红点集中任何一个顶点的最短路径都求出来.

3) 在蓝点集中选择一个最短距离最小的蓝点 k 来扩充红点集. 若 k 是蓝点集中估计距离最小的顶点,则 k 的估计距离就是最短距离.

4) 将 k 扩充到红点后,剩余蓝点集的估计距离可能由于增加了新红点 k 而减小,此时必须调整相应蓝点的估计距离. 对于任意的蓝点 j ,若 k 由蓝变红后使 $D[j]$ 变小,则必定是由于存在一条从 s 到 j 且包含新红点 k 的更短路径: $P = \langle s, \dots, k, j \rangle$. 且 $D[j]$ 减小的新路径 P 只可能是由于路径 $\langle s, \dots, k \rangle$ 和边 $\langle k, j \rangle$ 组成. 所以,当 $D[k] + w < k, j >$ 小于 $D[j]$ 时,应该用该值来更新 $D[j]$ 的值.

在蓝点集中选择一个最短距离最小的蓝点 k 来扩充红点集是 Dijkstra 算法的关键. 若能快速地访问到具有最小 D 值的蓝点,则可大大减少算法的时间复杂度. 若用传统的方法通过扫描整个网络图的顶点来搜索最小值,总时间代价为 $O(n^2)$. 在 Dijkstra 算法中,由累计权值决定的最短路径具有优先程度差异特征,所以若将未被处理的顶点的最短路径 D 值构造优先级队列,则可大大提高算法的效率. 用堆来实现优先级队列是很自然的,堆是一种抽象数据结构,由一系列元素集合构成,每个元素有个实数类型的关键字^[4]. 而二叉堆是一种最普及的数据结构,它可被视为一棵完全二叉树. 堆有最大堆和最小堆两种. 最小堆结构必须满足以下性质:除了根节点,每个节点的值不小于其父节点的值. 这样,堆中的最小元素就存在根节点中. 因为具有 n 个元素的二叉堆是一棵完全二叉树,其高度为 $\log n$. 所以对于二叉堆的操作例如 insert(插入)和 removemin(从堆中删除并返回最有最小关键字的元素),其作用时间至多与树的高度成正比,为 $O(\log n)$.

所以,将未被处理的顶点以 D 值大小为顺序保存在一个最小堆中,可以使用 $O(\log n)$ 次搜索找出下一个最近顶点. 每次改变 $D(x)$ 值时,都可以通过先删除再重新插入的方法改变顶点 x 在堆中的位置. 为了实现优先更新需要将每个顶点连同它的数组下标存储在堆中,其时间复杂度在算法实现部分分析.

2 带限制条件的最短路径算法分析

所谓带限制条件的最短路径是指满足用户约束条件的最短路径. 用户的约束条件是: 给定起始节点(start)和目标节点(end)以及该路径上的避开节点列 $B = \{B_1, B_2, \dots, B_n\}$ 和必经节点列 $J = \{J_1, J_2, \dots, J_n\}$.

若要避开某些节点, 则可在改进的 Dijkstra 算法中加入判断语句, 即当从堆中抽出具有最小 D 值的点并标记已访问时, 判断该点是否是避开节点列中的点, 若是则不入红点集.

若需经过某些节点, 则利用分段求最短路径的方法求出总的最短路径. 可将 J 存成一个数组, 用改进的 Dijkstra 算法先算起始节点 start 与 $J[0]$ 的最短路径, 然后对数组下标 i 进行循环, 判断下一个节点是否是已经过节点, 若是继续数组下一个节点, 否则调用 Dijkstra 算法计算 $J[i]$ 与该节点的最短路径, 然后再以该节点为起始节点重复上述判断直到数组最后一个元素, 最后计算它与终点 end 的最短路径, 再将所求得最短路径值相加即得总最短路径值. 要值得注意的是每次在调用 Dijkstra 算法之前, 要重新将所有的点设置为未访问过(UNVISTED), 因为算法结束时很多点都已访问过(VISITED). 而且要把各段所计算出的最短路径 D 值以及经过的点保存起来.

3 算法实现

首先是网络图的实现, 可以用一个 Graph 类来达到这个目的. Graph 类具有返回顶点数和边数的函数(分别为函数 n 和 e). 类中 first 函数返回与给定顶点关联的第一条边. next 函数返回此顶点的下一条与之关联的边. isEdge 函数用来判断给定边在图中是否存在. 这可以用来判断边表是否已处理完毕, 因为 next 函数在到达最后一条边时将返回 null 值. 任意给定一条边, 函数 $v1$ 和 $v2$ 分别返回它的起点和终点. Weight(w) 返回边 w 上的权值. 而 Mark 数组用来判断一个给定顶点是否已被访问过, 函数 getMark 和 setMark 访问和改变 Mark 数组的值.

在算法初始时创建数组 E , E 是由类 DijkElem 的对象构成的数组, 对象存储的是一个顶点以及它与给定始点之间的(当前最短)距离. 然后将始点入堆, 用最小堆类 MinHeap 建立堆. 对网络图的每一个顶点, 首先从堆中抽出最小 D 值的顶点 v , 并标记为已访问顶点. 这里需要注意的是执行这个语句的前提是首先要保证堆中有元素, 若没有则需返回. 而文献[1]中没有这个前提, 所以对某些始点不能正确输出. 对当前顶点 v 所有连接边的下一个顶点 $G.v2(w)$, 判断 $D[v] + G.weight(w)$ 是否小于 $D[G.v2(w)]$, 若是则将其插入最小堆. 该算法是在文献[1]基础上的改进, 关于图类 Graph、边类 Edge、最小堆类 MinHeap 以及 DijkElem 类等可参考该文献.

若要求出最短路径上经过的点, 可以设立一个由顶点组成的数组 P , 用 $P[v]$ 记录从起点到顶点 v 的最短路径上 v 的前一个顶点. 初始时, 对所有 v , 置 $P[v] = -1$. 在 Dijkstra 算法中更新最短路径长度时, 只要 $D[G.v2(w)] > D[v] + G.weight(w)$, 就置 $P[G.v2(w)] = v$, 否则不修改 $P[v]$ 的值. 当 Dijkstra 算法终止时, 就可以根据数组 P 找到从起点到 v 的最短路径上每个顶点的前一个顶点, 从而找到从起点到 v 的最短路径. 其核心代码如下:

```
Dijkstra(Graph G, int s, int e, double[] D, int[] B) { // 其中 G 是网络图, s 是始点,
    e 是终点. 数组 D 返回的是终点到始点的最短距离, 数组 B 是避开节点序列
    int v = 0;
    DijkElem[] E = new DijkElem[G.e()]; // 创建具有足够大空间的数组 E
    E[0] = new DijkElem(s, 0);
    MinHeap H = new MinHeap(E, 1, G.e()); // 建堆
    for (int i = 0; i < G.n(); i++)
        D[i] = Double.MAX-VALUE;
```

```

D[s] = 0;
for (int i = 0; i < G.n(); i++) {
    do { if(H.heapsize() <= 0) return; // 若堆中无元素则返回
        else v = ((DijkElem)H.removemin()).vertex(); } // 从堆中抽出具有最小 D 值的顶点
    while (G.getMark(v) == VISITED);
    G.setMark(v, VISITED); // 将抽出的点标记为已访问
    if (D[v] == Double.MAX_VALUE || v == e) return; // 当前节点是不可到达或是终点时返回
    int label = 1; // 判断是否为避开节点的标记
    for(int j = 0; j < B.length; j++) { if(v == B[j]) label = 0; }
    if(label == 1) { // 当 v 不是避开节点时
        for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
            if (D[G.v2(w)] > (D[v] + G.weight(w))) {
                D[G.v2(w)] = D[v] + G.weight(w); // 更新 D 值
                H.insert(new DijkElem(G.v2(w), D[G.v2(w)])); // 将满足条件的该点插入堆
                P[G.v2(w)] = v; // 记录最短路径上每一点的前一个顶点
            }
        }
    }
}

```

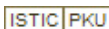
从算法的实现过程可知,第一个循环的运行时间为 $O(n)$. 在第二个主循环中, `removemin()` 操作在最坏情况下的运行时间为 $O(\log n)$, 在交通网络中由于避开节点个数远小于 n , 所以第三个子循环为常数时间 $O(1)$. 在第四个子循环中, 因为每条边在整个算法中仅仅处理一次, 所以此子循环(包括主循环)运行时间最多为 $O(m \log n)$. 也就是说在主循环中花费时间为 $O(n \log n) + O(n) + O(m \log n)$, 而对交通网络这样的稀疏图 ($m = O(n)$), 所以整个算法花费时间为 $O(n \log n)$. 与传统 Dijkstra 算法时间复杂度 $O(n^2)$ 相比, 明显有较高的运行效率.

4 结语

从算法时间复杂度可看出优先级队列运行结构的高效率及二叉堆在实现优先级队列中的优越性. 该算法不仅大大改善运行效率, 而且约束条件设置方便简单. 例如当某一路段发生交通事故时, 可把该道路节点设置为避开节点, 或者从家到商店, 若要取钱, 可将银行设置为必经节点.

参考文献:

- [1] Shaffer C A(美). 数据结构与算法分析(Java版)[M]. 张 铭, 刘晓丹译. 北京: 电子工业出版社, 2001.
- [2] 布莱恩·奥弗兰, 迈克尔·莫里森(美). Java2 精要(语言详解与编程指南)[M]. 刘伟等译. 北京: 清华大学出版社, 2002.
- [3] 江 斌, 黄 波, 陆 峰. GIS 环境下的空间分析和地学可视化[M]. 北京: 高等教育出版社, 1999.
- [4] 傅清祥, 王晓东. 算法与数据结构[M]. 北京: 电子工业出版社, 1999.

作者: [林小玲](#), [何建农](#), [周勇](#)
作者单位: [福州大学数学与计算机科学学院, 福建, 福州, 350002](#)
刊名: [福州大学学报 \(自然科学版\)](#) 
英文刊名: [JOURNAL OF FUZHOU UNIVERSITY \(NATURAL SCIENCE EDITION\)](#)
年, 卷(期): 2004, 32 (z1)

参考文献(4条)

1. Shaffer C A. [张铭](#), [刘晓丹](#) [数据结构与算法分析 \(Java版\)](#) 2001
2. [布莱恩·奥弗兰](#), [迈克尔·莫里森美](#) [Java2精要 \(语言详解与编程指南\)](#) 2002
3. [江斌](#), [黄波](#), [陆峰](#) [GIS环境下的空间分析和地学可视化](#) 1999
4. [傅清祥](#), [王晓东](#) [算法与数据结构](#) 1999

本文读者也读过(3条)

1. [周鹏](#), [张骏](#), [史忠科](#), [ZHOU Peng](#), [ZHANG Jun](#), [SHI Zhong-ke](#) [分段路径寻优算法研究及实现](#)[期刊论文]-[计算机应用研究](#)2005, 22(12)
2. [冯德民](#), [谢娟英](#), [FENG De-min](#), [XIE Juan-ying](#) [带单一限制条件的单源多权最短路径算法及其实现](#)[期刊论文]-[西南师范大学学报 \(自然科学版\)](#) 2000, 25(3)
3. [王文峰](#) [扩展RBM下的动态最短路径搜索算法的研究与实现](#)[学位论文]2009

本文链接: http://d.wanfangdata.com.cn/Periodical_fzdxxb2004z1012.aspx