

最短路径问题的解答图算法

余金山

[计算机科学(电脑)系]

摘 要

本文给出最短路径问题的一种算法——解答图算法。它是一种十分有效的算法。对于边的数目为 $n \times (n-1)$ 的图 (n 为图的顶点数), 本算法具有与 Dijkstra 算法同样的性能。而对于边稀疏的图, 本算法在时间和空间两方面都优于 Dijkstra 算法, 与 Dijkstra 算法相比较, 图的边数越少, 本算法所需的存贮空间也越少, 而其执行速度却越高。文中分析了时间和空间的复杂性, 并给出几个实际结果。

一、 引 言

图论在计算机科学中有着广泛的应用, 但是, 反过来如何利用计算机来解决图论问题, 特别是应用图论问题, 却又是图论研究中的一个十分重要的部分。正是计算机应用的普及, 才使图论迅速渗透到许多科学领域中去。可以说图论算法即是沟通理论与实际的桥梁, 又是把计算机应用于理论分析的关键环节。在应用图论中, 许许多多实际问题的解决就体现在找到一个有效的算法上。在实际应用中, 由于大量的最优化问题都等价于一个图的最短路径问题, 因此最短路径问题在应用图论中占有特别重要的位置。一般说来, 最短路径问题可以分为下面五个类型^[1]:

1. 某两个固定结点之间的最短路径;
2. 某个结点到其它所有结点的最短路径;
3. 各结点对间的最短路径;
4. 中间必须经过某些结点间最短路径;
5. 次最短路径。

这些问题, 由于它的重要性, 多年来人们已对其进行了深入的研究, 并给出了许多算法其中 Dijkstra 算法和 washall—Floyd 算法是公认的最有效的算法^[1]。但值得注意的是在 Dijkstra 算法和 washall—Floyd 算法中, 由于至少要存贮一个 $n \times n$ 的距离矩阵 (其中 n 为图的结点数), 它要占据很大的存贮空间。然而就应用而言, 绝大多数的实际问题所对应的图都是稀疏图, 即其边的数目远远小于 $n \times (n-1)$, 因而这种存贮方式显然是不经济的。另一方面, 也可以说是更重要的一方面是在寻找最短路径时, 这两个算法都必须首先探索两结点间

是否存在通路,然后再判别此通路是否最短,因此在处理规模较大且边又稍疏的图形时,其部分的时间将花在无用的探索上,从而影响了算法的执行速度。对于边稀疏的一类图形,由于它是实际中最常遇到的,因此又有许多人对其进行了专门的研究,以寻求更有效的算法。在这些研究中比较出名的是 *Hu-Torres* 的分解算法^[1]。用 *Hu-Torres* 算法求解第三类最短路问题,其计算速度的可比 *Warshall-Floyd* 算法提高 4 倍。本文介绍一种称为解答图的算法(以下简称本算法),对于边数为 $n \times (n-1)$ 的图形,本算法具有与 *Dijkstra* 算法同等的性能,而对于边稀疏的图形,本算法在时间和空间两方面都优于 *Dijkstra* 算法。稍后我们将指出,且从几个实例中也可看到,与 *Dijkstra* 算法相比较,本算法所需的存贮空间将随图的边数的减少而减少,而计算速度却可大大提高。

二、算法的实现

(一)基本定义

为了叙述上的方便,下面先给出几个基本定义。

定义1 一个图 G ,在存贮时如果对于其中的每一个结点 k 都用某个具有 m 个分量的指针或指向它的 m 个后件,则称这个图是按标准形式存贮的^[2]。

定义2 一个图 G ,如果至少存在一个结点 k_0 ,从 k_0 出发可以到达 G 中的每一个结点,则称图 G 是有根图,并称 k_0 是图 G 的根。

定义3 设给定了位置的有限集合 K 和规则 R ,对于每个 $k \in K$, R 定义了 k 的有限个后继位置的集合,即 $R(k) = \{k_1, k_2, \dots, k_t\}$ 。显然, R 可以看成是 K 上的一个二元关系。在 K 中,指定一个 k_0 为起始位置,指定若干个位置为终止位置。用 K^1 表示能够从 k_0 出发而按规则 R 到达的位置的集合。对于一个给定问题,若它的解答能够归结为从 K^1 中找出符合某些指定条件的位置集合 p ,则称这个问题能够用解答图来解答,并把二元组 (K^1, R) 称为解答图。显然 (K^1, R) 还是一有根图。

(二)解答图算法的实现

对于一个非简单图 G 来说,若去掉它的所有自环,并且在每一并行边的集合中选出其中最短的一条边来代替这些并行边,从而得到一个图 G^1 ,则显然 G 的最短路问题与 G^1 的最短路问题等价,且 G^1 是简单的。因此不失一般性,我们可假设所讨论的图都是简单的。此外我们在此还假定任意两点间的路径长度均非负。

显然,对于图 $G = \{V, E\}$ 中的某一结点 k_0 ,以及由 k_0 出发可以到达的结点的全体构成一个以 k_0 为根的有根图。设图 G 的结点数和边的数目分别为 n 和 $n \times m$ 。用 $d(k_i, k_j)$ 表示 k_i 到 k_j 的某一通路的长度,(其中, k_j 为 k_i 的直接后件),用 K 表示由 k_0 出发可以到达的结点的全体,则求 k_0 到其它各结点之间的最短路问题可以形式地描述为:对于任一结点 $k \in V - \{k_0\}$,求出一结点序列 $(k_i, k_{i_1}, \dots, k_{i_l})$ 使得:

(1) $k_{ij} \in K (1 \leq j \leq l)$;

(2) $k_{i(p+1)}$ 是 k_{ip} 的直接后件 $(1 \leq p \leq l-1)$ 且 $k_{i_1} = k_0, k_{i_l} = k$

对任一满足条件(1)和(2)的结点序列 $(k_{s_1}, k_{s_2}, \dots, k_{s_t})$,

$$\sum_{j=1}^{i-1} d(ksj, ks(j+1)) \geq \sum_{j=1}^{i-1} d(kij, ki(j+1)) = D$$

由此不难看出, 最短路径问题可用解答图的方法来解答. 算法的具体实现过程如下:

1. 用标准形式存贮图 G

2. 用约束查找法寻找最短路径, 即对任一 $k \in V - \{k_0\}$, 以前面所描述的条件(3)为约束条件, 求出满足条件(1)、(2)的结点序列及其最短路长 D . 不失一般性, 若图 $G = (V, E)$ 的边数为 $m \times n$, 则对任一 $ki \in V (0 \leq i \leq n)$, 可假定它恰好有 m 个后件, 记其为 Si .

命 $\left[\frac{n}{m} \right] = c$ ($\left[\frac{n}{m} \right]$ 表示不大于 $\frac{n}{m}$ 的最大整数). 则查找过程可描述如下:

(1) 建立 $\left[\frac{xn}{c} \right] + 1$ 棵结点数最多为 $\left[\frac{c}{x} \right]$ 的二叉平衡分类树 (初始时这些树均为空), 并记这些树为 T .

(2) 对每一个 $k \in S_0$, 以 $d(k_0, k)$ 为键并把其改记为 $t(k_0, k)$ 并加入到 T 中.

(3) 选取 $Di = t(k_0, ki) = \min \{t(k_0, ki) | t(k_0, ki) \in T\}$, 登记 Di 和 ki 并把以 $t(k_0, ki)$ 为键的结点从 T 中删除.

(4) 对任一 $k \in Si$, 若 k 不是 k_0 的直接后件, 则命 $t(k_0, k) = Di + d(ki, k)$, 并把 $t(k_0, k)$ 加入到 T 中. 否则, 若 $Di + d(ki, k) < t(k_0, k)$, 则命 $t(k_0, k) = Di + d(ki, k)$, 并调整此结点在树中的位置使该树仍为平衡分类的.

重复执行步骤(3), (4) $n-1$ 次即可求得 k_0 到其它所有结点之间的最短路径.

三、 讨 论

算法的正确性是显然的. 现着重讨论一下算法的执行速度和存贮空间问题. 无需复杂推导即可从算法的描述中看出, 执行步骤(3)最多仅需要 $\left[\frac{xn}{c} \right] + 1$ 步. 由于在 T 中插入一个结点 (或改变结点在 T 中的位置) 最多需要 $\log_2 \left[\frac{c}{x} \right]$ 步, 所以执行步骤(4) 最多需要 $m \log_2 \left[\frac{c}{x} \right]$ 步. 因此用本算法求出某一结点 k_0 到其它 $n-1$ 个结点之间的最短路径的时间复杂性为 $O(X)$, 其中 $X = \text{Max} \left(nm \log_2 \left[\frac{c}{x} \right], n \left(\left[\frac{xn}{c} \right] + 1 \right) \right)$

在算法中 x 是一个比较重要的参数, 它的选择依赖于 n, m 的比值 c . 在实际应用中, 可赋予一适当的固定值, 也可以编写一简单的程序来实现最佳选择.

下面我们就几个不同的 c 值把本算法的执行速度与 *Dijkstra* 算法的执行速度作一下比较.

(1) $c=1$. 若选取 $x = \frac{1}{2}$, 则 $n \left(\left[\frac{xn}{c} \right] + 1 \right) = n \left(\left[\frac{n}{2} \right] + 1 \right)$, $nm \log_2 \left[\frac{c}{x} \right] = n^2$.

(2) $c=8$. 若选取 $x=2$, 则有 $n \left(\left[\frac{xn}{c} \right] + 1 \right) = n \left(\left[\frac{n}{4} \right] + 1 \right)$, $nm \log_2 \left[\frac{c}{x} \right] = \frac{n^2}{4}$.

(3) $c=24$. 若选取 $x=3$, 则有 $n\left(\left[\frac{xn}{c}\right]+1\right)=n\left(\left[\frac{n}{8}\right]+1\right)$, $nm\log_2\left[\frac{c}{x}\right]=\frac{n^2}{8}$.

上面的数字表明, 当 $c=1$ 时, 本算法的执行速度与 *Dijkstra* 算法的执行速度一样. 当 $c=8$ 时, 本算法的执行速度约为 *Dijkstra* 算法的 4 倍. 当 $c=24$ 时, 本算法的执行速度约为 *Dijkstra* 算法的 8 倍. 由此可见, 对于边稠密的图形, 本算法与 *Dijkstra* 算法的执行速度是一样的, 而对于边稀疏的图, 本算法的执行速度却远高于 *Dijkstra* 算法且与 C 值的大小有关.

实际上若 $\frac{c}{x}$ 的值较小, 则算法中的树没有必要要求是平衡的, 甚至可以干脆采用链表组织, 这样可以使算法更简单些而且对执行速度的影响甚微. 在“四, 实例分析”中所给出的结果就是采用链表组织得到的.

关于存贮空间问题, 虽然说对许多图论算法来说, 它不如时间那么重要, 但在特定的情况下, 它也可能成为主要矛盾, 从而迫使人们不得不牺牲时间来换取空间. 特别是在处理规模较大的图时, 空间问题还是十分值得注意的. 容易看出, 本算法的空间复杂性为 $O(n^2)$. 在 n 较大且边稀疏的情况下, 本算法的空间复杂性显然优于 *Dijkstra* 算法. 例如当 $n=100$, $c=8$ 时, 若算法用一般的高级语言实现, 则 *Dijkstra* 算法至少要 40k 字节的内存. 但若采用本算法, 则仅需 10k 字节. 因此可以说, 即使应用较低档的微型机, 采用本算法来求解最短路问题一般是不会存在空间不足的问题的, 但若采用 *Dijkstra* 算法则不然. 在下面的实例分析中将看到这一点.

四、实例分析

上面我们从理论上对本算法进行了初步的讨论. 现给出几个实际结果. 并与 *Dijkstra* 算法做一比较.

一些实际结果的比较

实 例	Dijkstra 算法		解 答 图 算 法		x 值
	响应时间	空间要求	响应时间	空间要求	
实 例 1 ($n=24, m \approx 3$)	25	4.6k	10	2.8k	2
实 例 2 ($n=37, m \approx 3$)	51	8.4k	13	3.9k	2
实 例 3 ($n=51, m \approx 3$)	100	14.6k	16	5 k	2.5
实 例 4 ($n=74, m \approx 3$)	—	—	25	5.2k	2.5
实 例 5 ($n=127, m \approx 4$)	—	—	45	8.6k	3

表中的数据是在微型机 TRS-80 上运行得到的, 该机容量为 ROM 12K, RAM 16K.

算法程序用 BASIC 语言编写,表中响应时间的数据单位为秒,空间要求的数据单位为字节,响应时间包括程序的解释执行。

从表中可以看到,实例 1, $n=24$, $m \approx 3$, $c=8$, x 取值 2,理论上本算法的执行速度应为 *Dijkstra* 算法的 4 倍,实际为 2.5 倍;实例 2, $n=37$, $m \approx 3$, $c=12$, x 取值 2,理论上本算法的执行速度应为 *Dijkstra* 算法的 5 倍,实际为 4 倍;实例 3, $n=54$, $m \approx 3$, $c=18$, x 取值 2.5,理论上本算法的执行速度应为 *Dijkstra* 算法的 6.7 倍,实际为 6.3 倍;实例 4、实例 5 用 *Dijkstra* 算法,由于空间不足,已无法实现。可见本算法确实优于 *Dijkstra* 算法。当 n 较大时,本算法执行速度实际所提高的倍数基本上与理论值相符,当 n 较小时,则稍低于理论值,这是由于本算法要执行一些指针操作造成的。但这并不影响本算法的使用价值,因为与 *Dijkstra* 算法相比,本算法的执行速度还是大大提高了,而且当 n 很小时,问题是很简单的并不一定要用计算机来解答。在空间要求方面,对于内存空间较小的小型机或微型机来说,本算法更是明显地体现出了它的优越性,从表中可以看到,两个算法的空间要求相差是较大的。当 $n=70$ 时, *Dijkstra* 算法就无法在具有 16KRAM 的 TRS-80 机上实现,除非使用辅存或采取一些其它措施。但这样一来,在时间方面又要做出很大的牺牲,这显然是有局限性的。而若使用本算法,则情况就好得多。例如,当 $n=127$, $m \approx 4$ 时,在象 TRS-80 这样的低档微型机上执行本算法,内存空间尚剩 6.9K 字节左右。

五、 结 语

从上面的分析可以看出,本算法确是解决最短路径问题的一种很有效的算法。本质上能用 *Dijkstra* 算法解决的问题,同样能用本算法来解决。在效率上,本算法是优于 *Dijkstra* 算法的。正如引言中所指出,本算法的执行速度可随着图的边数的减少而提高,而空间要求却随之降低。特别是当图的规模较大,而计算机的执行速度较慢或内存空间有限时,更是体现出本算法的优越性。

电脑系部分教师对本文提出许多宝贵意见,微型机房的一些同志也给笔者很大的支持,谨在此表示感谢。

参 考 文 献

- [1] Narsingh Rao, Graph Theory with Application to Engineering and Computer Science, prentice-Hall, (1974).
- [2] H. Maurer, Datenstrukturen und Programmierverfahren, B. G. Teubner, (1976).