对 Dijkstra 算法的优化策略研究

陈益富，卢 潇，丁豪杰
（空军工程大学 电讯工程学院，陕西 西安 710077）

摘 要：Dijkstra 算法是许多工程解决最短路径问题的理论基础，但实际工程中涉及到的许多限制条件要求人们必须对该算法进行改进和优化。文中在对经典的 Dijkstra 算法思想进行分析的基础上，论述了 Dijkstra 算法的一种改进算法——A * 算法，并对它们之间的联系进行了剖析。在总结了一个实际工程项目开发的基础上，提出了一种基于 Dijkstra 算法上的针对铁路中两站点最优路径算法。文中提出的算法通过提取出铁路中的关键站点组成一个新图，之后将起点和终点插入到新图中，经过最多四次的排列组合后选出一个最短路径；该优化方法能将 Dijkstra 算法的时间复杂度 $o(n^2)$ 中的 n 降到一个很小的值。实践证明该方法在实际工程中完全可行且已取得了令人满意的效果。

关键词：最短路径算法；A * 算法；Dijkstra 算法；铁路两站点最优路径
中图分类号：TP301.6 **文献标识码**：A **文章编号**：1673－629X(2006)09－0073－03

Optimized Strategy Research over Algorithm of Dijkstra

CHEN Yi-fu, LU Xiao, DING Hao-jie
(Telecommunication Engineering Institute, Air Force Engineering University, Xi'an 710077, China)

Abstract: The algorithm of Dijkstra is academic foundation that many engineerings were solved in the shortest path issue, but people must improve and optimize to this algorithm what involves many limitative condition in the real engineering. So makes a summary on the foundation analysed to the algorithm of Dijkstra thought of classics, and discussed one kind of algorithm improving the algorithm of Dijkstra called A * Algorithm, to dissect the contact between them. On sum up the foundation of a real project, put forward based on being directed against railway two stations the shortest path algorithm on the algorithm of Dijkstra. The algorithm that was made in this article is put forward a new figure by way drawing out the hinge station in the railway. Afterwards, inserting starting station and terminal station to the new figure, you can select out the shortest path of railway after the combination of unexcelled four times the range. The optimized method can fall n value with the algorithm of Dijkstra of the time complexity $o(n^2)$ to a very small value. The practice proves this method is feasible completely in the real engineering and has taken satisfied effect.

Key words: the shortest path algorithm; A * algorithm; algorithm of Dijkstra; the best path between two railway stations

0 引 言

随着计算机的普及以及网络技术的发展，网络分析已受到人们的高度重视，它在电子导航、交通旅游、城市规划以及电力、通讯等各种管网、管线的布局设计中发挥着重要的作用，而网络分析中最基本最关键的问题就是最短路径问题[1]。当然最短路径不仅仅指一般地理意义上的距离最短，还可以引申到其他的度量，如时间、费用、线路容量等，相应地，最短路径问题就成为最快路径问题、最低费用问题等[2]，但无论是距离最短、时间最快还是费用最低，它们的核心算法都是最优(短)路径算法。

最优或最短路径问题是算法分析与设计中一个最基本的问题[2]，也是一个应用性比较广的问题，人们对此问题作了大量的研究，提出了多种算法。但当面对具体的项目工程时，它采用的算法细节是不一样的，很少有可以直接拿来使用的，一般都要对它们进行适当的改进，以满足时间复杂度、空间复杂度或其它方面的特殊要求。

经典的最短路径算法之一——Dijkstra 算法是目前许多工程解决最短路径问题采用的理论基础[2]，只是不同工程对 Dijkstra 算法采用了不同的改进方法。

1 经典 Dijkstra 算法的主要思想

Dijkstra 算法的基本思路是：假设每个点都有一对标号 (d_j, p_j)，其中 d_j 是从起始点 s 到点 j 的最短路径的长度（从顶点到其本身的最短路径的长度是零）；p_j 则是从 s 到 j 的最短路径中 j 点的前一点。求解从起始点 s 到点 j 的最短路径算法的基本过程如下[1]：

1）初始化。起始点设置为：

① $d_s = 0$，p_s 为空；

② 所有其他点：$d_i = \infty$，$p_i = ?$；

收稿日期：2005－11－30
作者简介：陈益富(1972－)，男，江苏盐城人，硕士研究生，研究方向为计算机网络与数据库技术；卢 潇，副教授，研究领域为算法分析与数据库技术。

③ 标记起始点 s,记 $k = s$,其他所有点设为未标记的。

2) 检验从所有已标记的点 k 到其直接连接的未标记的点 j 的距离,并设置:$d_j = \min[d_j, d_k + l_{kj}]$,式中 l_{kj} 是从点 k 到 j 的直接连接距离。

3) 选取下一个点。从所有未标记的结点中,选取 d_j 中最小的一个 i:$d_i = \min[d_j$, 所有未标记的点 $j]$。点 i 就被选为最短路径中的一点,并设为已标记的。

4) 找到点 i 的前一点。从已标记的点中找到直接连接到点 i 的点 $j*$,作为前一点,设置:$i = j*$。

5) 标记点 i。如果所有点已标记,则算法完成退出搜索,否则,记 $k = i$,再到第二步继续搜索,直到所有点都已标记。

可以看出,在按标记法实现 Dijkstra 算法的过程中,核心步骤就是从未标记的点中选择一个权值最小的弧段[2],即上面所述算法的 2) ~ 5) 步。这是一个循环比较的过程,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。那么要选择一个权值最小的弧段就必须把所有的点都扫描一遍,在有若干个顶点的情况下,这无疑是一个制约计算速度的瓶颈。显然由于 Dijkstra 在运行时要执行两重嵌套的 FOR 语句,因此其总的时间复杂度是 $o(n^2)$[3]。

2 一种改进的 Dijkstra 算法——A * 算法

在介绍 A * 算法前,先研究一下广度优先搜索。广度优先搜索就是每次将当前状态可能发展的策略逐层展开[3],比如一个地图中,对象允许向 4 个方向移动,那么,就将地点处对象向上下左右各移动一步,将 4 个状态都保存在内存中,然后再从这 4 个出发点向各自的 4 个方向再移动一步……(当然这里可以剔除不合理的移动方法,比如不准向回移动)。实际上,整个搜索好似一个圆形向外展开,直到到达目的地,很明显这样求解一定能找到最优解,但节点展开的数量是和距离成级数增加的[3]。

而 A * 算法实际是一种启发式搜索,所谓启发式搜索,就是利用一个估价函数评估每次的决策的价值,决定先尝试哪一种方案[4]。这样可以极大地优化普通的广度优先搜索。一般来说,从起始点 (A) 到终点 (B) 的最短距离是固定的,可以写一个函数 judge() 估计 A 到 B 的最短距离,如果程序已经尝试着从起始点 (A) 沿着某条路线移动到了 C 点,那么认为这个方案的 AB 间的估计距离为 A 到 C 实际已经行走了的距离 H 加上用 judge() 估计出的 C 到 B 的距离。如此,无论程序搜索展开到哪一步,都会算出一个评估值,每一次决策后,将评估值和等待处理的方案一起排序,然后挑出待处理的各个方案中最有可能是最短路线的一部分的方案展开到下一步,一直循环到对象移动到目的地为止。

假如,在此所讨论的 A * 算法给图添加一个附加条件——两点之间直通距离最短,就是说如果两点 A 和 B 之间有直通的路径(不经过其他顶点),那么这条路径就是它们之间的最短路径。其结果是找到的最短路径将是经过顶点最少的一条线路。如果求的是顶点 A 能够直接到达的顶点 B 的最短路径,时间复杂度就会由原来的是 $o(n^2)$ 降低为 $o(n\log n)$。

这个变化的产生,是因为添加了"两点之间直通距离最短"这样的附加条件,实际上就是引入一个判断准则,把原来 Dijkstra 算法内循环的 $o(n)$ 搜索降到了 $o(\log n)$。这个思想就是 A * 算法的思想。

从 Dijkstra 到 A * 是判断准则的引入(例如两点之间直通距离最短),如果这个判断条件不成立,同样地,只能采用不完整的 Dijkstra 算法(求到目的顶点结束算法)。

由此可见,怎么写这个算法中的估价函数非常的重要,如何保证一定能找到最短路径呢?充要条件是,你的估价函数算出的两点间的距离必须小于等于实际距离。如果你的估价函数不满足这点,就只能叫做 A 算法,并不能保证最后的结果是最优的,但它可能速度较快。

3 一个实际工程中对 Dijkstra 算法的优化策略

在笔者参与的一项工程中涉及到这样一个有关最优路径的问题:求取全国任意两个铁路站点之间的最优路径。由于全国铁路站点的数量上万个,若直接运用 Dijkstra 算法其时间复杂度是相当大的,而等待时间也是客户所无法忍受的,因此运用经典的求到目的顶点结束的 Dijkstra 算法是不能达到客户的需求的,那么能不能运用 A * 算法呢?首先是估价函数 judge() 的寻找,因为铁路线错综复杂,相邻站点之间的距离大小无规律可言,两个站点最短路径之间最多和最少要经过多少个站点与其估计距离之间没有明显联系[5],因此无法创建一个有效的估价函数;其次是两个站点之间的估计距离如按直通距离计算,即假设两站点之间有直通铁路,那么就要先求出两站点之间的直线距离。笔者试验过根据站点的地理坐标来进行计算,但许多小站点的地理坐标资料无法得到,即便通过地图标绘能得到一个近似值(其工作量也是相当大的),引入到算法的初始化阶段求出了两站点之间的直线距离,但关键问题是最终发现有不少站点之间的最短路径并不是经过站点最少的路径,也就是说,运用 A * 算法最后求出的路径是经过站点最少的路径,但并不一定是距离最短的路径,这同样也不符合客户的要求。

在实际项目的开发过程中也进行了许多其它的尝试,先后提出了分片计算、关键路径连接、咽喉要道点插入等多种方案,最后均因为达不到客户某些方面的要求而被迫放弃了。经过一段时间的研究尝试,最后确立了选择临近交叉点方案,其中心思想是就是通过先选出全国铁路中所有交叉点来形成一个只有较少站点数的新图,之后根据起点站、终点站与交叉点的临接关系来确定一至四张插入图,此时再调用 Dijkstra 算法,其时间复杂度将会变得很小,且只要求客户输入起点站和终点站即可,这样在智能

性上也达到了用户的要求。

选择临近交叉点方案的理论基础和具体算法如下:因为不在同一条铁路段上两站点的连结相通肯定会经过铁路的交叉点,而且每一个站点要通向其它铁路段的站点必定会经过其两端交叉点中的一个,同理其它铁路段的站点要来到这个站点也必定会经过其两端交叉点中的一个;因此将全国所有的交叉点提前进行初始化(即去除所有不是交叉点的站点),得到以它们为顶点的一个图(称该图为 ForkGraphics);每一条边的权值为各交叉点与它们直接连接的其它交叉点的最短(优)距离。对全国的任何一个站点要么其本身就是铁路的交叉点,要么沿其所在的铁路至少能找到一个与其相连接的交叉点,一般的情况是要查询的起始点和终点均有两个与其相临的交叉点,这里称之为它们的前交叉点和后交叉点,将起点和终点插入到 ForkGraphics 中,用穷举的方法将各种情况进行排列组合最多可形成 4 个图,起点与终点之间的最优路径一定包含在这 4 张图中,对这 4 个图分别调用 Dijkstra 算法,之后再选出最小值,则其必为起点到终点的最短(优)路径;当然,如果判定两个站点在同一铁路段上,此时问题就变得非常简单,因为其最短(优)路径即为它们之间的直通距离。这个算法利用了分治法[2]的一些思想,但又不是严格意义上的分治法。当然在这里省略了工程中涉及到的具体细节,例如速度、载重能力、能通过的最大宽度等,因为都可以将其转化为两个顶点之间的权值。

将起始站和终点站插入到 ForkGraphics 中可得到如图 1 所示的铁路交叉站点示意图。

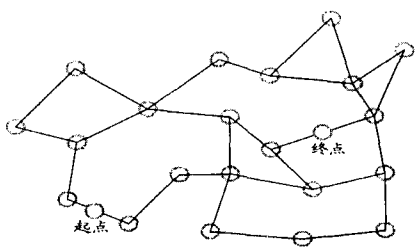

图 1　交叉站点示意图

经穷举组合后可得到图 2~图 5 的 4 个可选路径图。

如果起始点或终点本身就是一个交叉点,则情况就只会有两种,若起点和终点均是交叉点,显然就在 ForkGraphics 中寻找最优路径即可。也有这样的情况,那就是有一个站点甚至两个站点都在铁路的末端线路上,这时的情况也变得简单了,因为与之相连的交叉点只有一个,只

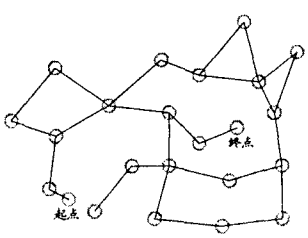

图 2　可选路径图一

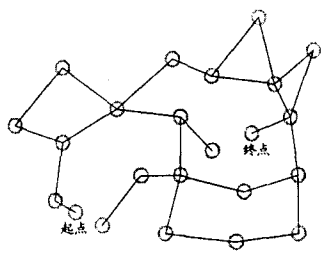

图 3　可选路径图二

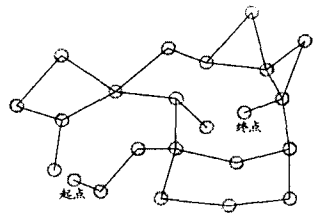

图 4　可选路径图三

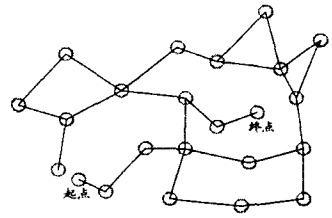

图 5　可选路径图四

要将该站点连接到与它相临的交叉点,再进行排列组合即可,此时的图也只会有一个或两个。

其算法的流程如图 6 所示。

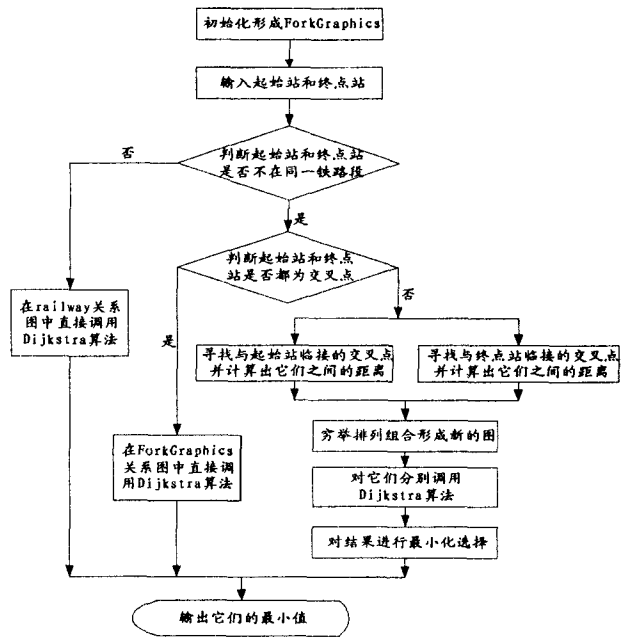

图 6　程序流程图

4　小　结

现在的工程项目中客户对系统的要求越来越高,尤其是要做到及时响应和智能化,而目前提出的求取最短路径的算法大约有 17 种之多。F. Benjamin Zhan 等人对其中的 15 种进行了测试,结果显示基于 Dijkstra 算法的改进算法,更适合于计算两点间的最短路径问题[6]。总体来说,这些算法采用的数据结构及其实现方法由于受到当时计算机硬件发展水平的限制,将空间存储问题放到了一个很重要的位置,以牺牲适当的时间效率来换取空间节省[1]。目前,空间存储问题已不再是要考虑的主要问题,因此有必要根据工程的实际情况和客户的特殊要求对已有的算法进行必要的改进,必要时应该用空间换时间来提高最短路径算法的效率,用对算法的合理改进来使软件产品更加智能化,而不应要求客户做出太多的判断或进行许多参量的输入。

参考文献:

[1]　乐　阳,龚健雅. Dijkstra 最短路径算法的一种高效率实现

(下转第 78 页)

组中,保证调度策略的公平性,内核在一个进程时间片耗尽时,通过 EXPIRED_STARVING()宏判断当前 CPU 的 runqueue 中是否有进程在 expired 队列中等待过长时间。如果有这样的进程,那么即使当前进程是交互进程也不再将其插入 active 数组,而是排空 active 数组,切换到 expired 数组运行。

在多处理器系统中,由于存在多个就绪进程队列 runqueue,可能存在多个队列之间负载的不平衡状况:其中一部分 runqueue 包含的进程远远大于其余 runqueue,这种情况将影响系统的整体性能。Linux 通过 load_balance()函数实现各就绪队列间的负载平衡。该函数在两种情况下被调用:schedule()在当前 runqueue 为空时调用或者在系统空闲时由定时器每 1ms 调用一次,而在系统忙时则由定时器每 200ms 调用一次。当 load_balance()由 schedule()调用时,由于当前 runqueue 是空的,它简单地寻找任意就绪进程将其加入当前队列;而当由定时器调用时,load_balance()处理的工作相当复杂:

a. 首先,load_balance()调用 find_busiest_queue()确定最繁忙的 runqueue,即包含最多就绪进程的队列。

b. 其次,load_balance()决定从最繁忙 runqueue 的哪个 prio_array 数组"拉"出进程。除非该 runqueue 的 expired 数组为空,否则 load_balance()总是选择 expired 数组。

c. Load_balance()然后从选定的 prio_array 数组中找到优先级最高的进程列表。接着分析列表中的每个进程以确定是否适合被"拉"出,如果适合,则调用 pull_task()将该进程从原 runqueue"拉"到当前 runqueue。重复此过程直到负载平衡。

4 实时进程

Linux 2.6 内核对实时进程[1]的支持相对于以前版本的内核有很大的加强。其引入的两项新特性(分别是 O(1)调度算法和内核抢占支持)有利地提高了系统的实时性能,这两点都保证实时进程能在可预计的时间内得到响应。

Linux 支持两种实时进程调度策略:SCHED_RR 和 SCHED_FIFO,而普通的非实时进程的调度策略是 SCHED_NORMAL[2,4,5]。SCHED_FIFO 实现一个简单的无时间片先进先出的调度算法。一旦一个 SCHED_FIFO 实时进程获得运行,它将一直运行下去,除非它由于某种原因被阻塞,或者它自己放弃 CPU。在这种情况下,只有高优先级的实时进程可以抢断它。具有相同优先级的 SCHED_FIFO 进程以轮转的方式运行,而所有低优先级的进程将永远不会得到运行机会直到高优先级的进程全部运行结束。

SCHED_RR 策略与 SCHED_FIFO 相似,差别在于每个 SCHED_RR 进程被分配一个预定的时间片,当一个进程的时间片耗尽后,它将被抢占,CPU 以轮转的方式运行具有相同优先级的 SCHED_RR 实时进程。与 SCHED_FIFO 进程相同,高优先级的 SCHED_RR 进程总是立刻抢占低优先级的进程,而一个低优先级的进程永远不会抢占高优先级进程,即使高优先级进程的时间片已经耗尽。

实时进程的优先级范围为 0~MAX_RT_PRIO-1,对于实时进程来说,内核并不计算它的动态优先级,实时进程的动态优先级由 setscheduler()函数设定,而且一旦设定就不再改变。进程运行的顺序只取决于设定的动态优先级,而静态优先级与非实时进程一样决定进程的时间片长短。

5 结束语

Linux 操作系统经过十余年的发展,已经成为当今最成功的操作系统之一。其最新 2.6 版本的内核实现了一个高效的 O(1)级调度器,相对于 2.4 版内核具有更好的实时性能、重负载下更高的 CPU 使用率以及交互作业更快的响应等优良特性。但 2.6 版内核的实时性能仍然不能满足硬实时系统的要求,可抢占内核也只限于对 CPU 的抢占,还不支持对内存等其他资源的抢占。所有这些都将是今后 Linux 发展过程中值得深入研究的课题。

参考文献:

[1] Love R. Linux Kernel Development(2nd ed)[M]. 北京:机械工业出版社,2005.

[2] 倪继利. Linux 内核分析及编程[M]. 北京:电子工业出版社,2005.

[3] Bovet D P, Cesati M. Understanding the Linux Kernel(2nd ed)[M]. 北京:中国电力出版社,2004.

[4] 李善平. Linux 内核 2.4 版源代码分析大全[M]. 北京:机械工业出版社,2001.

[5] 毛德操,胡希民. Linux 内核源代码情景分析[M]. 杭州:浙江大学出版社,2001.

(上接第 75 页)

[J].武汉测绘科技大学学报,1999,24(3):209-211.

[2] 王晓东. 算法设计与分析[M]. 北京:清华大学出版社,2004.

[3] 严蔚敏,吴伟民. 数据结构[M]. 北京:清华大学出版社,1997.

[4] 金炳尧. 最优化问题中的若干新技术[J]. 科技通报,2002(2):119-124.

[5] 米涅卡 E. 网络和图的最优化算法[M]. 李家滢,赵关旗译. 北京:中国铁道出版社,1984.

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