海量道路数据下的最短路径规划效率①

刘晓晖 陆 挺 (中国电子科技集团公司 第二十八研究所 江苏 南京 210007)

摘 要: 通过内存映射完成大拓扑文件的快速加载，估价函数快速过滤无效点和路段，折线简化算法加速长路径的显示等手段，解决了海量道路数据下的最短路径规划效率问题。经比较分析与实践检验证明，规划算法的综合性能指标小于 5 秒，具有较好的工程应用价值。

关键词: dijkstra; 最短路径; 导航; 内存映射; 估价函数; 曲线简化

Efficiency of the Shortest Path Problem in Huge Amount of Road Data

LIU Xiao-Hui, LU Ting (CETC 28, Nanjing 210007, China)

Abstract: In huge amounts of road data, calculating the short path is time-consuming. A completed solution for improving the efficiency of Shortest Path Problem is proposed. In the solution, file-mapping, evaluation function and simplification of polylines are used for increasing huge topo file loading, filling out ineffectual data to improve Dijkstra algorithm and accelerating results. With practical experience, the system works out the problem in no more than 5 seconds.

Keywords: dijkstra; short path; navigation; file-mapping; evaluation function

1 引言

经过几十年的经济建设，我国已经形成了贯通全国、四通八达的路网，加之幅员辽阔，全国的道路数据可以用海量形容。以四维公司 2008 年出品的数据为例，全国 5 级以上公路里程为 147 万公里，经拓扑计算后的道路路段有 262 万条、路段交点有 194 万个。按此数据进行最短路径规划，效率问题显得特别突出。分析原因，计算时间主要耗费在路径规划程序中数据准备、路径计算和结果显示这三个步骤:

1) 海量拓扑数据加载时间长;

2) 经典的或现有优化的 Dijkstra 算法效率尚不能令人满意;

3) 超过 500 公里以上的路径计算结果显示速度慢。

针对以上问题，分别使用内存映射，基于拓扑点坐标和拓扑路等级，曲线简化等三个方法逐一解决。

2 海量拓扑数据加载优化

计算最短路径的方法以 Dijkstra 算法最为经典，并由此派生了很多优化算法。但不论算法内部采用什么形式的数据结构，算法的输入参数类型却几乎完全相同，总可以表达为由若干相关联的 Node 和 Edge 组成的 Network，见图 1。所不同的是，数据结构采用数组或是链表。

```
struct Node {                   // 路段交点
  int       nodeId;             // 结点ID
  int       edgeLst[8];         // 邻接边ID数组
  float     x,  y;              // 结点坐标
};
struct Edge {                   // 路段
  int       edgeId;             // 边ID
  int       startNodeId;        // 边起点ID
  int       endNodeId;          // 边终点ID
  int       lddj;               // 0,高速；1，一级公路；
                // 2，二级公路；…；5，5级公路
};
struct Network {                // 网络拓扑
  Node    nodeLst[1944026];             // 结点集合
  Edge    edgeLst[2625149];             // 边集合
};
```

图 1 拓扑数据代码

① 收稿时间:2009-12-21;收到修改稿时间:2010-01-31

由于 Node 和 Edge 的数据量为 194 万和 262 万，因此，等到用户输入条件启动运算时再从空间数据库或地图文件中加载数据，耗时令人难以忍受，以奔四 2.4G，Oracle9i，内存 2G 的机器为例，平均读取时间超过 300 秒，如再使用链式存储结构动态构造链表则时间更长，且内存消耗约 130M。考虑到拓扑数据的不变性（数据更新频率超过 1 次/年），可以将拓扑数据预先写入到本地文件，能减少网络和数据库的时间消耗。但仍有大量的文件格式解析工作、动态数组或链表的构造工作，加载时间也只能下降到 10 秒左右。

分析上述问题的原因，首先是时间因素，空间数据读取、网络传输和数组构造这三个步骤都耗时过长；其次是空间因素，拓扑数据占用了 130M 的内存，对于 PDA 等内存受限系统显得过大。为此，可采用文件的内存映射机制解决时空效率问题。内存映射属于 Windows、Unix、Linux 系统的高级内存管理机制，其原理可概括为：每个用户进程都拥有独立的进程空间，寻址空间一般为 4G（32 位机器），不同进程的相同的内存地址以页为单位被映射到 RAM 的不同区域。也就是说，进程空间与内存空间是逻辑与物理的关系，在进程中拥有一个指向 130M 内存区域的指针，极有可能在实际的物理内存中才装载了 1K（设 1K/页），而其余的仍在辅存硬盘上，当出现页面中断时，操作系统自动将需要使用的数据读取到内存中，而这一切都发生在系统内核，对上层的用户进程是透明的。

应用内存映射机制，拓扑数据的处理步骤变为：

1) 系统发布前，开发人员一次性读取拓扑数据到内存，将 Network 对象整体性写入到指定的文件。将该文件打入安装包，作为系统的一部分部署到用户机器上；

2) 系统运行加载拓扑数据时，以只读模式打开该文件（注：此时不读取任何数据），调用 QFile::map(file)[1]命令返回一个 char*指针，将该指针强制转换为 Network*，即完成 130M 数据的加载。

步骤 2 加载 130M 的拓扑数据，平均耗时小于 1 毫秒，且内存分配小于 100K。

3 Dijkstra算法优化

围绕 Dijkstra 算法进行优化，是近年来研究的热点，众多学者提出了配对堆方法[2]、双向 Dijkstra 算法[3]、基于链表的 Dijkstra 优化算法[4]等。尤其是配对堆方法在数据量为 100 万 Node，1000 万 Edge 时，计算时间已能控制在 7 秒以内，但其中未给出具体的测试条件，尚无法判断计算是否覆盖了大部分的点。另一方面，配对堆方法使用了 1:10 的 Node 与 Edge 比例关系，这与实际道路情况不符，应用在道路优选上无法最大限度地发挥堆排序带来的性能提升。双向 Dijkstra 算法[3]和基于链表的 Dijkstra 优化算法[4]未能给出大数据量的验证数据，但通过研究其优化方法，估计与配对堆方法的性能相差不会太大。

总的来说，上述优化算法针对经典 Dijkstra 算法没采取任何措施将未标记节点无序地存放在链表或数组中，使得针对 Node 的循环比较计算耗时太多，提出了各种算法和数据结构，解决了临时标记节点的排序、存储和更新问题。但它们仍是以传统的图论、算法和数据结构为基础，未能针对实际道路的最短路径规划问题进行优化，因此，以时间复杂度来度量经典 Dijkstra 算法和各优化算法，性能虽由 $O(N^2)$ 提升为 $O(N\log_2 N)$，但当 Node 和 Edge 数据量过百万时，计算速度仍不能令人满意。

由于经典的 Dijkstra 算法是一个从起点开始的广度遍历算法，即以起点为圆心，沿着 Edge，不断向外扩大搜索半径，直到遇到终点。也就是说，搜索时间与拓扑数据的总体规模无关，而与起点终点的距离成 2 次指数相关。由此，最有效的优化方法就是大幅度降低“N”，即 Node 的循环次数，具体的方法是为 Node 增加地理坐标信息、为 Edge 提供道路等级信息，以此完成快速过滤。

3.1 通过建立外接搜索矩形快速过滤 Node

如果以人工方式按图查找南京到乌鲁木齐的最短路径，估计常人是不会去考察广州市的道路，但经典的 Dijkstra 算法却以南京为中心进行广度遍历。因此，最有效的优化方法是将这些无意义的 Node 排除在计算外。为此，建立一个针对 Node 地理坐标（见图 1 中结构 Node）的估价函数，该函数评估 Node 是否为可能最短路径上的候选点。估价函数必须简单快速，这里用以起点和终点为顶点的外接矩形，并将该矩形向外扩大 2 度。之所以扩大 2 度，主要考虑的是起点到终点的最短路径一般不是一条直线，经常会出现绕路的情况，特别是在我国的西北部地区，经常出现目

的地在南面却不得不从出发地先往北走几十公里的情况，因此在这里引入了一个 2 度（约 200 公里）的经验值。在 Node 的循环中，增加一个对待评估 Node 的坐标是否在该矩形中的判断，如在矩形中，则进行随后的计算，否则跳过。利用这种方法，再次计算南京到乌鲁木齐的最短路径，南京以南超过 200 公里的拓扑点可被快速排除。

3.2 通过设置道路搜索等级快速过滤 Edge

沿用南京到乌鲁木齐的例子，由于它们的 X，Y 坐标相差很大，外接矩形仍覆盖了国境内 50%的拓扑数据，在使用一种简单快速的 Dijkstra 优化算法[5]时，约需 30 秒左右。究其原因，时间主要浪费在低等级道路的计算上，例如，算法在衡量开封、西安、兰州等途经城市道路时，将市内道路也进行了计算。由于城市道路密度高，权值又低于高速、国道、省道，因此这样的计算不仅耗时长，而且计算结果也不可能成为最短路径的一部分。

因此，综合外接矩形和道路等级快速过滤 Node 和 Edge 的方法，为 Dijkstra 算法引入一个快速估计函数，整个估价函数的代码如下。

```
bool val ( Node node, Edge edge, int ldd) {
// 判断node是否在外接矩形中,bindingRect已扩大2度
if （! bindingRect . PtInRect ( node )）
    return false;
// 当Node与起点、终点的坐标差超过2度时，通过判断道路等级，快速过滤低等级道路。lddj是允许的最高道路等级值
float dis2From = abs(node.X – from.X) + abs(node.Y – from.Y);
float dis2To = abs(node.X – to.X) + abs(node.Y – to.Y);
if (edge.ldjd > ddj && dis2From > 2 && dis2To > 2)
    return false;
  return true;
}
```

图 2 估价函数代码

在该估价函数中，有一个重要的判断语句：dis2From > 2 && dis2To > 2 && edge.ldjd <= lddj。该语句表达了算法将以三个阶段完成路径搜索：

第一阶段（出城阶段），以起点为圆心进行广度搜索，搜索半径为待搜索点与起点的经纬度坐标差的绝对值的和不超过 2 度的点。虽然评估的路径为所有等级的道路，但由于搜索半径不大，算法可以很快完成；

第二阶段（快速行驶阶段），当搜索半径开始大于 2 度时，仅搜索指定等级以上的道路。例如，当 lddj 设置为 1，即仅搜索高速和一级公路（国道），如果查询失败，则进行第二次叠代，lddj 需要加 1。通过这样的方法，大量低等级的中间道路都被直接过滤掉，避免了无效计算。表 1 列出了测试数据的道路等级数量分布情况。

表 1 道路等级数量分布情况

公路等级	数量	占比
高速公路	87009	3.31%
一级公路	190964	7.28%
二级公路	393221	14.98%
三级公路	373102	14.21%
四、五级公路	1580853	60.22%
总计	2625149	100.00%

第三阶段（入城阶段），当待评估的 Node 接近终点时(两者经纬度差的绝对值的和小于 2 度)，再次面向所有路段进行搜索，直到遇到终点。

综合以上方法，随机选择了 500 组起点终点进行测试，测试环境为奔四 2.4G，内存 2G，开发环境为 MSVC2005，使用 QTime:: elapsed ()[1]计算各测试用例 100 次循环前后的时间差值，再求平均。测试结果表明，所有用例耗时均小于 1000 毫秒，平均耗时 477 毫秒。表 2 列出了其中不同里程、覆盖东西部路网的较有代表性的测试用例和测试结果。

表 2 优化后的 Dijkstra 算法的测试结果

序号	起点	终点	总里程 km	平均耗时 ms	备注
1	[118.6，28.1]	[118.2，28.6]	111	163	
2	[118.6，28.1]	[119.2，29.5]	268	210	
3	[118.6，28.1]	[115.2，27.0]	560	516	
4	[119.2，35.42]	[115.4，28.44]	1067	978	东部路网

序号	起点	终点	总里程 km	平均耗时 ms	备注
5	[105.1, 38.33]	[106.8, 33.5]	1293	192	中部路网
6	[118.5, 41.62]	[119.3, 28.44]	1949	893	
7	[108.5, 32.47]	[126.29, 54.69]	3961	491	
8	[108.5, 27.47]	[125.29, 54.69]	4832	446	
9	[110.4, 22.1]	[87.6, 50.1]	5802	453	广东到新疆
10	[88.5, 27.47]	[125.29, 54.69]	6967	513	东北到西藏

3.3 路径结果显示优化

Dijkstra 算法返回了最短路径上的所有路段，各路段的空间数据以点集合的形式表示。当路径长度超过 400 公里，集合中的点超过 20000 个时，使用 MapX 控件、C#语言开发的程序绘制这些路径点需要 5 秒以上，用户体验度明显下降。由于绘制是由 MapX 控件独立完成，不可能深入控件内部提高效率。因此，有效的方法就是降低路径点集合的数量，且保证道路线型在容许的范围内不失真。

计算几何中 Douglas-Peucker(DP)算法，可将端点过于密集的折线进行简化处理，且计算速度非常快。算法的基本思想是顶点到某条边的距离过于接近的话，将被舍弃，保留距离大于阈值的顶点。初始化时，首先连接第一个和最后一个顶点构建第一条边，找到剩余顶点中距离这条线段最远的顶点，然后分别连接第一和最后一个顶点到此最远的顶点，构成两条线段，继续寻找剩余顶点中分别到这两条线段距离最远的顶点，依此类推，直到顶点到边的距离小于设置的阈值停止处理，找到的这些顶点就构成了简化折线。

具体方法是，系统设置了超过 400 公里和 20000 个点为启动折线简化的条件，随后调用 DP 算法，输入的阈值为 0.01 度（精度为 1 公里），计算性能可达到：输入 10 万个点进行简化处理，处理可在 250 毫秒以内完成，点数据压缩率平均可达到 93%。简化计算完成后，再生成一个图层 A，根据简化后的点数据绘制道路。

经过以上处理，整个路径的显示时间可在 2 秒内完成。但由于经过 DP 算法简化的路径已经失真，当 MapX 窗口视野放大到 50 公里或更小时，可明显看出路径与实际道路不重合，这一现象在城市中的立交桥处特别明显，如图 3 所示。解决的方法也比较简单：当视野放大到 50 公里或以下时，将 A 图层设置为不可见，另生成一个 B 图层，并使用未简化的点数据在 B 图层上绘制路径。由于视野范围小、参与绘制的数据量不大，无需优化就能快速完成绘制，用户的体验能得到很好的保证。

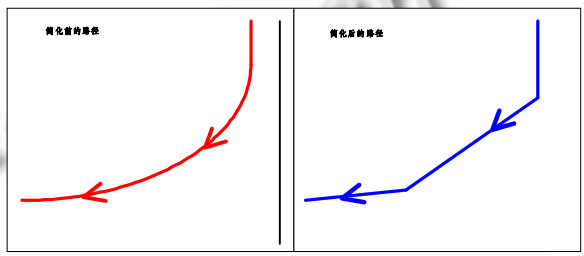

图 3　路径简化前后的对比图

4　结束语

通过内存映射完成大拓扑文件的快速加载，估价函数快速过滤无效点和路段，折线简化算法加速长路径的显示等手段，整个路径规划的时间一般不超过 5 秒。以南京到乌鲁木齐为例，Google 公司出品的网上地图耗时 6 秒左右，灵图公司出品的单机版 MapChina 需要 7 秒左右，相比均有小幅提高。当然，从本文的介绍中也可以看出，方法并没使用复杂的算法或数据结构，仅就具体的道路导航需求进行了有针对性的优化，以满足工程中技术指标要求为目标，如能在算法上进一步求精，相信性能仍有提高的空间。

参考文献

1 Blanchette J, Summerfied M. C++ GUI Programming with Qt 4.2nd ed. 北京:电子工业出版社, 2008.20－300.

2 王志和,凌云. Dijkstra 最短路径算法的优化及其实现. 微计算机信息, 2007,11(3):275－278.

3 宁建红.最短路径算法效率研究.上海电机学院学报, 2006,(3):38－41.

4 张红科.基于链表的 Dijkstra 算法优化研究. 电脑知识与技术, 2008,3(8):1702－1734.

5 乐阳,龚健雅. Dijkstra 最短路径算法的一种高效率实现.武汉测绘科技大学学报, 2005,22(2):208－210.