# 参考阅读一：常用的导航/路径规划软件都用到哪些算法？

昨晚睡前在看吴军先生的数学之美，在其12章中提到导航软件采用的算法是动态规划。  
感觉DP解路径规划问题划分一直是个大麻烦，相对于经典的路径规划算法，也没体现出巨大优势。  
早上在cnki下了一些相关论文（只讨论主干方法，因为学术界喜欢在算法上做优化以及几种算法的桥接，比如基于dijkstra与遗传免疫算法的XX研究当做为dijkstra）  
1.对于静态路径规划，大多采用dijkstra，A，floyd，dp等  
2.对于NP难问题，如TSP及变形（物流配送等），则采用遗传算法，蚁群算法，模拟退火，神经网络等  
  
问题：  
1.在实际的商业应用中，采用的是哪些算法或者理论呢？  
2.路径规划是提前规划好以后存储（像网页的索引），还是收到请求后实时规划的结果？

法dijkstra是无法满足互联网地图检索实时响应这种性能要求，所以各家公司都有各自的预处理方法：分层或者预计算。具体采用何种方式，这取决于采取的加速算法相关。在2008年前后，以KIT（[http://algo2.iti.kit.edu/routeplanning.php](https://link.zhihu.com/?target=http%3A//algo2.iti.kit.edu/)）为主的研究院产出了多个路径规划加速算法，其中以contraction hierarchies 和 highway hierarchies较出名，加之微软研究院提出的Customizable Route Planning，与传统的A-star，基本上支撑了目前工业界地图产品的路径规划服务。

A-star：[https://en.wikipedia.org/wiki/A](https://link.zhihu.com/?target=https%3A//en.wikipedia.org/wiki/A" \t "_blank)\*\_search\_algorthm

CH：[http://algo2.iti.kit.edu/schultes/hwy/contract.pdf](https://link.zhihu.com/?target=http%3A//algo2.iti.kit.edu/schultes/hwy/contract.pdf" \t "_blank)

HH：[http://algo2.iti.kit.edu/documents/routeplanning/esa06HwyHierarchies.pdf](https://link.zhihu.com/?target=http%3A//algo2.iti.kit.edu/documents/routeplanning/esa06HwyHierarchies.pdf" \t "_blank)

CRP：[http://research.microsoft.com/pubs/145688/crp-sea.pdf](https://link.zhihu.com/?target=http%3A//research.microsoft.com/pubs/145688/crp-sea.pdf" \t "_blank)

Dijkstra是一种非常经典的方法，这种方法已经很成熟。通过对权值的计算，可以一次性计算出所有出发点到目的地的所有路径信息，目前在一个城市里，基本使用这种方法已经足够。但是，实现路径规划只是基本的，想要将导航做的精确，需要对行驶在路段上的车辆进行统计，故障路段进行判断，并实时调整路径。

# 参考阅读二：导航路径规划示例



置顶 [罗家山的蚊子](https://blog.csdn.net/autonavi2012) 2018-07-05 10:53:13 28883 收藏 116

版权

**导航路径规划之一 路网模型**

https://blog.csdn.net/autonavi2012/article/details/80901337

考虑到路网作为网络的特殊性以及路线优化问题的要求，符合导航需要的路网表达方法和存储结构应满足如下要求：

1.        存储量小

2.        便于路线优化算法对其进行操作

3.        充分考虑路网作为网络的特殊性-大型稀稀疏网络

4.        能充分表达路网的要素和拓扑结构

5.        能表达单项交通、交叉口转向限制等交通管制措施

6.        能表达路网的各种特殊结构

7.        要考虑节点权重如何存储

描述路网的基本要素有节点、路段与形状点，如图1-1所示。从路网的直观结构考虑，很自然地会想到用图来表示路网。交叉口对应节点，两交叉口之间的路段对应边或弧，路段的某种量化属性作为权，这样用一个赋权图可以初步描述路网。又由于同一个路段的不同方向其属性一般而言不一定相同，故可以采用有向图描述路网。

节点、路段和形状点构成了路网的基本要素。三者可以用图1-1来表示。



                         图1-1

**1.1 节点与属性**

节点并不是实际的点，它是一个抽象的概念。在路网中我们通常理解的节点就是交叉口、高速路出入口、起点和终点等。为了导航中路线优化的需要，节点包括下列类型：道路特性发生变化了的点，以及可能进行转向操作的点。很明显，在节点处道路的权值会发生变化，这会影响到路径规划的结果。

在路径规划中，节点应具备如下属性：

1.        节点标识：全国路网中，每个节点都有唯一的一个标识。

2.        节点坐标（经纬度）：节点所处位置的经纬度。

3.        节点类型：是主节点还是子节点，具体信息参考下面的复合路口。

4.        红绿灯信息：该节点所处的交叉口是否有红路灯。

5.        路牌信息。

1.2 路段与属性

路段是两结点之间连接的弧,是道路包含的最小单元，是指两个节点间的道路属性完全相同的一段道路。在一条道路中，可以有很多个路段，也就是说，路段必须是具有同一属性、在路网中独立的一段道路。一般来说，路段的特征属性信息应包括如下内容：

1.        路段标识：在全国路网中，每个路段都应有唯一的标识。

2.        起始点号码：起始点和终止点的节点号码。

3.        路段长度：该路段的实际长度，单位为米。

4.        路段的幅宽：该路段的实际宽度，单位为米。

5.        道路等级：高速公里、国道、省道、县道、乡道、城市快速路等。

6.        道路属性：上下行、交叉点内部路、环岛、内部路等。

7.        形状点数据：描述该道路形状的坐标点数据。

8.        路段名称：该道路的名称。

9.        收费设置：该路段是否收费。

10.    通行关系：双向通行、逆方向、顺方向通行。

1.3   路网的拓扑关系

    拓扑结构是指如何描述节点、弧段等要素之间案的邻接、关联、包含等关系。对于路网来说，主要是连通性和方向性。连通性是指如何表达路段以及节点之间的链接关系。方向性主要考虑是否用有向图来表示路网的问题，有向图增加了路网的存储量，但由于同一路段两个方向的属性一般而言不尽相同，所以一般采用有向图来表示路网，另外，有向图便于处理单行线，交叉口转向限制等附加属性。路网连通性有两个层次的含义：实体意义上的相邻性和实际交通行为中的可达性。

1.      实体意义上的相邻性。

直观上，一个路段与哪些路段是相连的，一个节点和哪些路段邻接，这是较低层次的连通性，利用一般的网络表示方法就可以直观表达。

2.      交通行为中的可达性。

在实际路网中交通管理部门会经常采用一些交通管制措施，比如单项交通、转向限制、分时段限行等，由此造成直观上连通的路线，在实际交通行为中是不可达的，因而是不连通的

1.3.1 简单路口

       节点和路段是组成路口的两个基本要素。节点在形态上可以分简单路口和复杂路口。由一个节点、一个或多个路段组成的路口是简单路口，由两个或两个以上节点、两个或多个路段组成的路口构成复杂路口。简单路口的示意图如图1-2所示。



图1-2 单一路口

1.3.2 复杂路口

    由简单要素组成的复杂路口采用复合路口表示，引入复合路口的目的是将形状复杂的路口或立交桥，用一个简单路口替代，通过引入“复合路口”和“复合路段”，简化道路网络，加快路径规划的计算速度。复杂路的形状示意图如图1-3所示，其中红色的代表主节点和主路段。

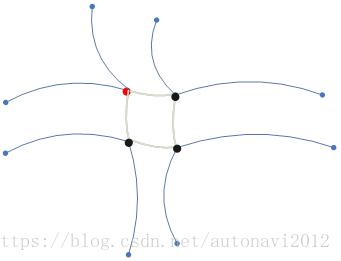


 图1-3 复杂路口

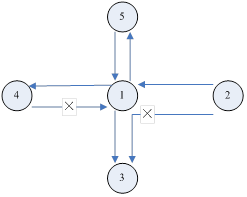
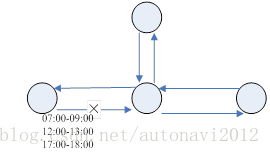
1.3.3 禁行关系  
在路网中，有的路段是禁止通行的（人行道），有的路段是允许单向通行，还有的路段是分时段限制通行的，在考虑路网的存储结构时，必须考虑此类禁行关系。在路径规划中，由于采用的是有向图表示路网，弧可以表示两个节点的连通关系，因此如果两个节点之间没有弧，则表示此两个节点不可达，而禁止左转则涉及到三个节点（或一个节点、两条弧），则需要特别考虑，而分时段限制通行被视为禁止通行。  
   
图1-5 单向行驶与禁止左转  
  
  
 

图1-6 分时段限制通行

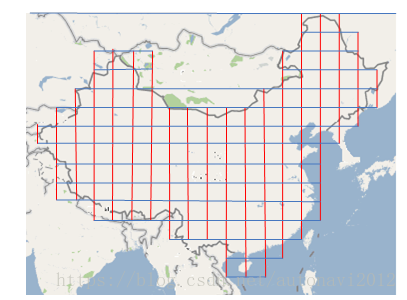
1.4 路网区域划分   
路网区域划分（Mesh）进行存储在路径规划中也称为分治法。其基本思想是把全国的道路网分割成相同大小的区域（块），路径规划时可以分批次加载路网数据，避免一次性加载所有道路网络。这样划分的每一块称为一个图符.   
将大范围数据分Mesh存储，把全国范围的数据按照一定的经纬度范围做Mesh划分,在此所划分的是源数据，无论源数据是什么形式，数据库也好，文件系统也好，首先按Mesh划分后存储。这样在计算道路进行路径规划时不必将全部道路网映射到内存中，而是先导入一部分图符的数据进内存，计算到Mesh边界点时候，将该邻接的Mesh映射到内存中进行局部道路继续运算，把长时间没有参加运算的Mesh从内存中清除掉，通过相邻两Mesh的边界点之间的拓扑关系，可以将各个Mesh连接起来组成一个大的拓扑网络。   
  

图1-7 路网mesh

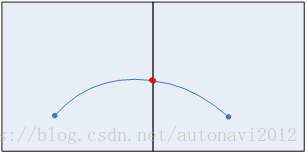
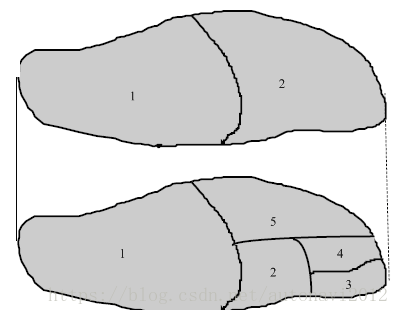
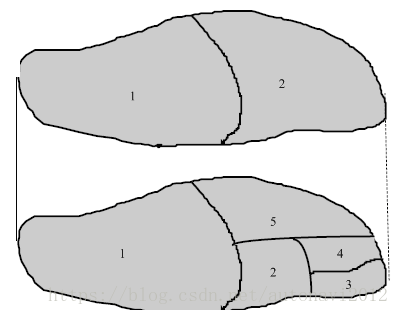
如果一个路段跨越了两个Mesh单元，则需要在每个Mesh单元的边界添加两个虚拟的节点，称为图框点，如下图1-8所示。图框点必须有相关信息，指示下一个节点在那个Mesh单元以及下一个节点的节点号。   
  

图1-8 框点

1.5 路网的分层   
在远距离的路径规划中，例如规划一条从深圳某地到北京某地的一条路径，计算的时候需要加载大量的图幅的数据文件，由于图幅数比较多，因此计算的速度会很慢。   
其实在远距离的路径计算中，一般都会选择走高速路。因此如果把全国的路网分成几个层次，然后根据不同的层次划分不同的区域。在最底层划分的区域最小，但区域内包含全部的路网数据，最上面的层次划分的的区域最大，但只包含等级比较高的路网，比如高速公路和城市主干道。划分区域的时候，上层的区域必须全部包含对应的下层区域，例如图1-9的上层区域2必须包含下层区域的2、3、4、5。   
    
图1-9 路径规划数据的区域层次

    路网分层后，必须建立层与层之间的节点关系。因此在设计节点的数据结构的时候，要考虑到节点所在的图幅号和对应的上层或下层的节点号，如图1-10所示。

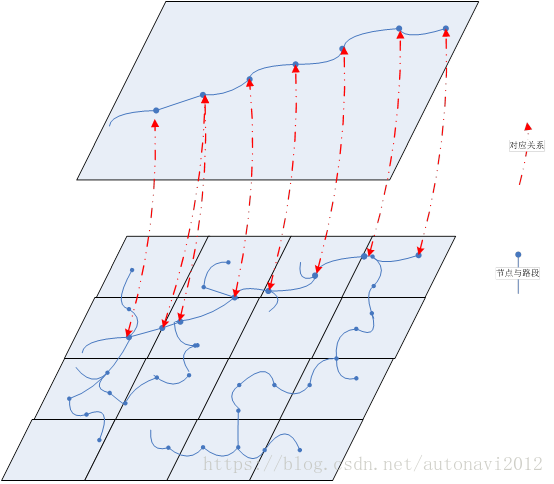


图1-10

建立高层路网的节点的原则是：   
1) 划分两个层次：高层路网和低层路网。   
2) 一个高层路网的区域包含16（暂定）个低层区域。   
3) 高层路网的节点数据包含低层路网的高速公里、城市主干道的节点数据。   
4) 与高速公路、城市主干道相邻接的导航路段也必须包含在高层路网中。   
5) 建立高层路网中的节点与低层路网中的节点

**导航路径规划之二 路网的存储结构**

路网的存储结构

图的两种基本存储结构：邻接矩阵和邻接表，下面分别介绍。

1.1 邻接矩阵

设有一个具有n个节点的有向图G=(V,E),V=(v1,v2…vn).

最简单的图的存储结构是一个n×n的0-1阶矩阵A=(aij)n\*n来定义，其中：



   使用邻接矩阵存储路网的最大优点在于它容易确定某一给定节点射入和发出的弧的集合。如对于节点vi来说，邻接矩阵第i行中的每一个”1”就相当于从节点vi发出一条弧，而第i列中的每个1就相当于有一条弧射入节点vi.

由于邻接矩阵有n的平方个元素，故采用邻接矩阵存储路网的空间代价是O(n3)，而与弧的实际数量无关。由于实际路网是一个大型的稀疏图，数据的冗余很大，不适宜采用。

1.2 邻接表   
邻接表是另一种常用的图的存储结构。常用于稀疏图。它为稀疏图G的每一个节点分配一个链表，共计m个链表。节点vi∈V对应的链表包含的元素是邻接于节点vi的节点集。

根据定义，稀疏图有m=O(n)条边，因此，用一个邻接表描述稀疏图只需要O(n)个存储空间，显然优于邻接矩阵。我们使用改造的邻接表来表达路网。

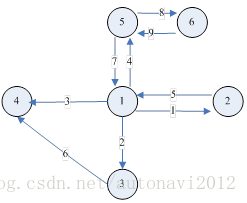
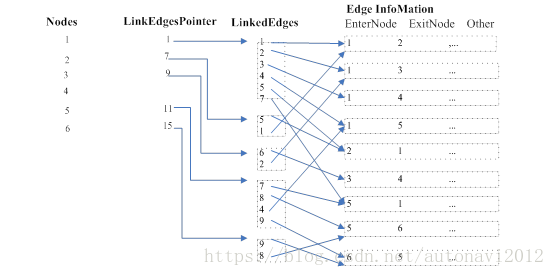
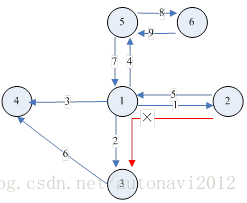
1.3 适合路网的存储结构   
路网作为大型稀疏网络，具有交叉口转向限制，因此普通的邻接表存储结构不能满足，需要针对路网的特点以及导航中路线优化的要求，寻求合适的存储结构存储路网，达到存储量尽可能小，同时便于路线优化算法对其进行操作，还能够正确表达交叉口转向限制，合理存储节点权重等要求。   
当前最常用的路网存储结构是一种称为“前向关联边”的链表结构。其核心是在于将同一个节点发出的所有弧存放在一起。首先分配一个长度为n的数组，该数组每一个元素对应一个节点，每个节点中有一个指针LinkEdgesPointer，记录由此节点发出的第一条弧在整个弧集中的起始位置，此外，还用一个长度为m的数组LinkEdges存储LinkEdgePointer指针所指向的弧   
这种存储结构的一个明显好处是节省存储空间。仅仅使用一个长度为n的数组和一个长度为m的数组，而通常的存储结构需要两个长度为m的数组(一般n<<m).此外，该结构还可方便地对某一给定的节点发出的所有弧进行操作。   
更重要的是，该存储结构拥有一个其他存储结构所不具备的特性，即在表示路网的同时，还暗含了交叉口行为，换句话说，车辆在临近给定交叉口时允许采取的操作，如直行，左转、右转等也通过该存储结构隐含的表达出来，具体说，当车辆由弧（i,j）即将到达节点j时，他下一步可采取的操作（下一个可能到达的节点）可以在LinkedEdges数组中找到，其在LinkedEdges数组中的起始位置由该节点的LinkEdgesPointer给出。例如图1-9所示的交叉路口，可以用如图1-10所示的结构图来表示。   
    
图1-9 交叉路口   
    
图1-10 没有禁则关系的交叉路口的表达方法   
考虑转向限制，有的交叉路口是禁止左转的，如下图1-11。这种情况下，上面的前向关联边就无法表达这种关系，因此还需要对上面的结构加以改造。   
  


图1-11 有禁行关系的交叉路口

可以在每个节点中添加一个标志信息，该标示信息提示在该节点处有没有转向限制，同时将转向禁止的入边和出边（也可以是进入的节点和出去的节点，如图1-11的进入的节点2和出去的节点2）存储起来，这样在路径规划的时候，当扩展到这样的像节点1的时候，如果下一步扩展的节点是3，则需要检查进入的节点是否是节点2，如果是的话，则表示扩展失败，因为此交叉路口禁止左转。

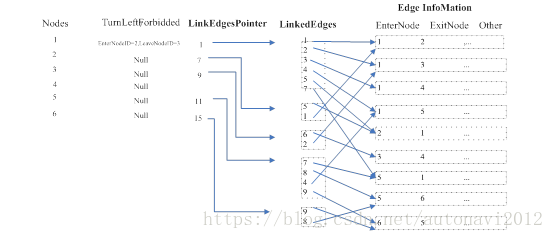


 图1-12 有禁则关系的交叉路口   
根据上面的设计思路，可以设计路网的节点与路段的数据结构如下：

节点信息，Nodes（结构体）

|  |  |  |  |
| --- | --- | --- | --- |
| 描述 | 占用空间 | 指派空间 | 名称 |
| X | 4字节 | 4字节 | m\_nX |
| Y | 4字节 | 4字节 | m\_nY |
| 邻接图符节点号（只在图框点中有，其他0） | 2字节（编译时需判断同一图符内nodeid号是否会大于32767，是则溢出，需修改） | 4字节（高17位存邻接图符号，低15位存节点号） | m\_nAdjNode |
| 邻接图符号（只在图框点中有，其他0） | 17位 |
| 主节点或子节点号码（当前节点如果是子节点则是主点号码，如果是主点则是记录内部路的子点号码，简单路口空，非路口节点空） | 2字节 | 2字节 | m\_snCrossNode |
| 种别代码（收费站，图框点等6种） | 3位 | 1字节（从高位到低位3，2，1，1，1） | m\_ucAttribute |
| 路口标识（非路口，主点，子点，简单路口） | 2位 |
| 红绿灯（有无） | 1位 |
| 交通禁行（有无） | 1位 |
| 路牌（有无）暂填0 | 1位 |
| 接续link | 1字节 | 1字节 | m\_ucLinkNum |
| 接续link地址 | 4字节 | 4字节 | m\_psnLinkId |

与节点关联路段：接续link（数组）

|  |  |
| --- | --- |
| 描述 | 空间 |
| 接续link号 | 2字节 |

Edge是弧，该信息分为2部分，第一部分是路径规划需要的信息，第二部分是其他模块需要的信息，这样在路径规划的时候，只需要导入第一部分的信息，从而加快规划的速度，第二部分的信息在其他模块需要的时候再导入到内存中。

Edge（结构体）

|  |  |  |  |
| --- | --- | --- | --- |
| 描述 | 空间 | 指派空间 | 名称 |
| Part1 | | | |
| 起点号码 | 2字节 | 2字节 | m\_snStartNode |
| 终点号码 | 2字节 | 2字节 | m\_snEndNode |
| 道路长度 | 2字节 | 2字节 | m\_snLength |
| 道路等级 | 4位 | 1字节 | m\_ucLevel |
| 通行方向 | 2位 | 1字节（2，1） | m\_ucDrection |
| 是否收费 | 1位 |
| Part2 | | | |
| 道路名称编号 | 2字节 | 2字节 | m\_snNameId |
| 道路属性 | 5位 | 1字节（5，2） | m\_ucAttribute |
| 车道数（大于4就取4）  =通往起点的车道数+通往终点的车道术-所有附加车道数 | 2位 |
| 起点入角度 | 9位 | 1字节（低8位） | m\_ucStartAngle |
| 终点入角度 | 9位 | 1字节（低8位） | m\_ucEndAngle |
|  |  | 1字节（最低位是起点入角度的最高位，次低位是终点入角度的最高位） | m\_ucAngleEx |

typedef struct MAP\_POINT\_TAG   
{   
 int   m\_nx;   
 int   m\_ny;   
}MAP\_POINT;   
typedef struct MAP\_SHORT\_POINT\_TAG   
{   
 short m\_snX;   
 short m\_snY;   
}MAP\_SHORT\_POINT;   
typedef struct EDGE\_EX\_TAG   
{   
  int m\_nEdgeAttr;    //路段属性   
  short m\_snEdgeAngle;            //路段角度   
 short m\_snEdgeAngleRever;        //路段对角   
 int m\_nSharpOffset;              //路段形状点数据的偏移地址   
 int m\_nEdgeNameOffset;          //路段名称数据的偏移地址    
 unsigned char m\_ucNum\_Slane;    //车道数   
    unsigned char m\_ucWidth;        //路段宽度   
}EDGE\_EX;   
    
typedef struct EDGE\_TAG   
{   
 unsigned short m\_nEdgeLen;  //路段长度   
 unsigned char  m\_ucEdgeType;      //路段等级              
 unsigned char  m\_ucDirection;      //通行方向(通行方向+收费设置)   
 unsigned short m\_nStartNodeId;  //开始节点ID   
 unsigned short m\_nEndNodeId;  //结束节点ID   
 EDGE\_EX \*      m\_pstEdgeEx;        
}EDGE;   
typedef struct NODE\_TAG   
{   
 unsigned int    m\_nAdjMapId:17;     //联接图幅号  如果当前节点是图框点，则存储联接图幅号     
 unsigned int    m\_nAdjNodeId:15;    //联接节点号   
 unsigned short  m\_nNodeId;          //节点ID  如果当前节点是子节点，则存储该节点的主节点     
                               //主节点或子节点号码（当前节点如果是子节点则是主点号码，如果是             
//主点则是记录内部路的子点号码，简单路口空，非路口节点空）   
 unsigned char   m\_usnType:2;  //节点类型 分主节点 子节点 单一路口 和不是路口的节点   
 unsigned char   m\_ucTurnLeft:1;     //是否有禁止左转   
 unsigned char   m\_ucLed:1;          //是否有红绿灯   
unsigned char   m\_ucritical:1;      //是否是图框点   
 unsigned char   m\_ucGuidePost:1;    //是否有路牌   
unsigned char   m\_ucTollGate:1;     //是否是收费站   
unsigned char   m\_ucTube:1;         //是否是隧道   
 unsigned char   m\_ucRelNodedCount:6;//关联节点数   
 unsigned char   m\_ucCrossJCT:1;     //平面交叉点JCT   
 unsigned char   m\_ucCrossInOut:1;   //平面交叉点出入口   
 unsigned int    m\_nLongtitude;      //经度   
 unsigned int    m\_nLatitude;        //纬度   
union {   
struct {   
unsigned int m\_nMapID:13;   
unsigned int m\_nNodeID:19;   
}LowerLevel;   
Struct {   
Unsigned int m\_nMapID:17;   
Unsigned int m\_nNodeID:15;   
}UpperLevel;   
}   
unsigned short  \*m\_pRelNodeID;         //关联节点数组   
}NODE;   
typedef struct FORBID\_TURN\_TAG    //禁止转向数据结构   
{   
   int   m\_nInEdgeId;             //进路段号   
   int   m\_nNodeId;               //节点      按节点的大小顺序存储  可用二分查找   
   int   m\_nOutEdgeId;            //出路段号   
}FORBID\_TURN;   
typedef struct MAP\_DATA\_TAG    //图幅数据   
{   
unsigned short m\_unNodeNum; //节点个数   
unsigned short m\_unLinkedEdgeNum;  //与节点连接的边的个数   
struct Edge \* Edges; //边的信息   
} MAP\_DATA;

**导航路径规划之三 路网数据的组织**

    路网数据的组织如下图1-1所示。将一个图符的节点、关联路段信息存入在一起，将全国的图符编译进一个文件，这样的好处是在路径规划时可以减少磁盘I/O的次数，从而提高规划的速度。

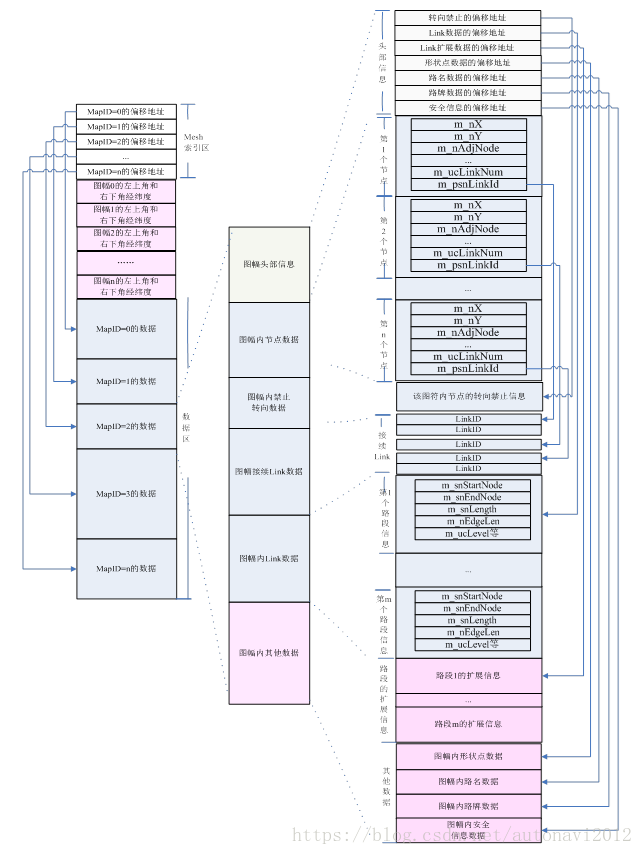


图1-1 数据的组织形式

# 导航路径规划之四 路径规划概述

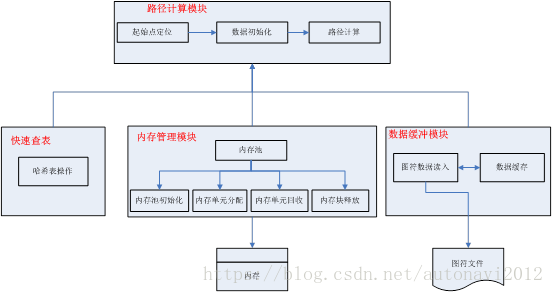
   路径规划的主体是路径计算，即规划一条从起始地到目的地的一条合理路径，路径计算采用的算法是A\*算法，实现A\*算法还需要一些辅助步骤，以提高程序运行的效率。因此在内部把路径规划模块细分为路径计算模块、内存管理模块、数据缓冲模块和快速查表模块，其主体是路径计算模块，其他模块处于从属地位，也就是说，其他模块是为计算模块提供辅助支持，如图1-1所示。

    路径计算模块需要实现起始点\目的地的定位、数据初始化、路径计算等功能。

    在路径计算过程，需要维护一个Open表和Close表。当扩展一个节点的时候，需要把与该节点相邻接的节点放入Open表，而把该节点放入Close表，因此会频繁地从内存区域申请小块内存以存放节点信息到Open表中，如果使用默认的系统的内存管理，会造成系统性能下降。因此要实现自己的内存管理，内存管理模块主要实现一个内存池，该内存池对内存分配、释放以及回收进行管理，以提高程序运行的效率。

    全国的道路网分割成相同大小的区域（块），路径规划时可以分批次加载路网数据，以避免一次性加载所有道路的网络数据，缓冲模块主要管理路径计算时对文件的I/O操作。

    快速查表模块主要实现一个哈希表，以快速在Open表和Close表中进行定位。



**导航路径规划之五 A\*算法**

    A\*算法是启发式搜索，是一种尽可能基于现有信息的搜索策略，也就是说搜索过程中尽量利用目前已知的诸如迭代步数，以及从初始状态和当前状态到目标状态估计所需的费用等信息。

   A\*算法可以选择下一个被检查的节点时引入了已知的全局信息，对当前结点距离终点的距离作出估计，作为评价该节点处于最优路线上的可能性的量度，这样可以首先搜索可能性大的节点，从而提高了搜索过程的效率。

   A\*算法的基本思想如下：引入当前节点j的估计函数f\*,当前节点j的估计函数定义为：

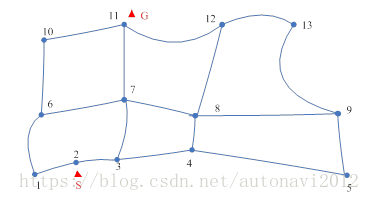
                            f\*(j)= g(j)+h\*(j)

其中g(j)是从起点到当前节点j的实际费用的量度，h\*(j)是从节点j到终点的最小费用的估计，可以依据实际情况，选择h\*(j)的具体形式，h\*(j)要满足一个要求：不能高于节点j到终点的实际最小费用。从起始节点点向目的节点搜索时，每次都搜索f\*(j)最小的节点，直到发现目的节点。

A\*算法的核心是设计估价函数，设计估价函数h(j)有很多方法，下面介绍其中的两种。

**估价函数1：欧几里德距离**

可以证明，在和起点距离相等的中间节点集合里，与终点直线距离（欧几里德距离）越小的节点，方向夹角越小。假设在图2-3求S到G点的最短路径，与S最近的点是节点2，与G最近的是节点11，因此可以确定上路点是节点2，下路点是节点11，求S到G点的最短路径就是求节点2到节点11的路径。定义L(i,j)表示从节点i到节点j的有向线段，Angle[ L(I,j),L(a,b) ]表示有向线段ij和有向线段ab的夹角。由于Angle[ L(2,11),L(2,3) ]< Angle[L(2,11),L(2,1) ] ,因此选择节点3，而与节点3连接的节点有节点4和节点7，Angle[ L(3,11),L(3,7) ]< Angle[ L(3,11),L(3,4) ]，选择节点7，与节点7相连的节点有节点8，节点11，和节点6，由于Angle[L(7,11),L(7,11) ]< < Angle[ L(7,11),L(7,6) ]<Angle[ L(7,11),L(7,8) ],选择节点11，至此，已经找到从S到G点的一条路径。



假设起点S的坐标（Sx,Sy），终点G的坐标(Gx,Gy),中间点N的坐标(Nx,Ny),

估价函数取欧几里德距离，表示为：



这个估价函数的计算量很大，不利于海量数据的路径规划计算。

**估价函数2：曼哈顿距离**

利用欧几里德距离计算估价函数的计算量很大，因此考虑将两点之间的曼哈顿距离作为估价函数。其中A点的经纬度为(Ax,Ay),B点的经纬度是(Bx,By),则A、B之间的Manhattan距离可以表示为:

                            M(A,B)=P\*(|Ax-Bx|+|Ay-By|)

P = 2\*∏\*R / 360

由于P是常数，可以简化为

                            f(j)=g(j)+(|Ax-Bx|+|Ay-By|)

此估价函数计算量小，虽然不是严格的方向优先，但基本能保证最短路径的搜索方向向目标点的方向进行。

       A\*算法在搜索中设置两个表:Open表和Close表。Open表保存了所有已生成而未被考察的节点，Close表中记录已被考察过的节点。算法中有一步是根据估计函数重排Open表中的节点，这样，循环的每一步只考虑Open表中状态最好（代价最小）的节点，如果被发现在Open表中存在同一个节点，就应比较两个节点代价的大小，如扩展得到的新节点代价大于已有的节点代价，可以放弃扩展得到的新节点，否则就以新节点代替原来的节点。如果在Close表中存在同一个节点号，则新新生成的路径的代价肯定大于原来路径的代价，可以淘汰新生成的节点。

算法步骤如下：

**步1**：生成空的Open表、Close表，将起点S放入Open表

**步2**：如果Open表为空，则失败退出。

**步3**：从Open表中，找出头节点U,作为当前节点,将它从OPEN表中移除。

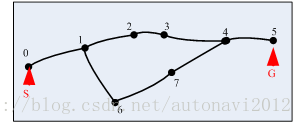
**步4**：判断U是否为终点，如果是，则通过节点U的父指针，一直遍历到起点，找到到最优路径，算法结束。否则，转步5.

**步5**：扩展当前节点U,找到其扩展的后继节点集合V,遍历集合V中的节点，如果节点即不在Open表也不在Close表，计算该节点的估计值，将节点加入到OPEN表，设置父节点为U.如果节点在Open表、Close表，比较当前节点的代价g(v)和Open表、Close表中代价，如果当前节点的代价g(v)小，更新表中的代价和父节点指针。

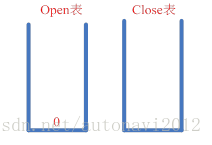
**步6**：按估价值递增的顺序，对Open表中所有节点进行排序。

**步7**：转步2.

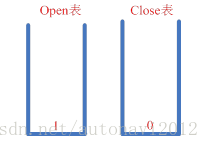
举一个例子来说。有路网如下图，假设需要从S节移动到G节点，则A\*算法的计算过程如下：



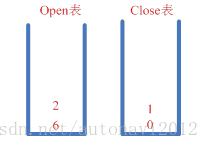
1）初始状态：设置节点0的f(x)=0, 节点0进Open表，寻路成功标志bFind为false;设置Close表为空。



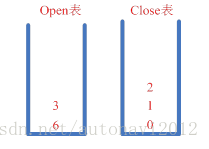
2） 估算节点0，考察与节点0相连的所有节点（只有节点1）是否是目标节点，如果不是则估算这些节点的f(x)值，将其放进Open表,同时节点0被放入Close表；最后需要将open表中的节点按f(x)的值从小到大的顺序排列。



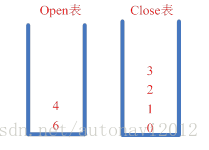
3）从Open表取出f(x)值最小的节点（节点1），考察与节点1相连的所有节点（节点2，6）是否是目标节点，如果不是则估算这些节点的f(x)值，将其放进Open表，同时节点1被放入Close表。最后需要将open表中的节点按f(x)的值从小到大的顺序排列（由于节点2比节点6接近目标节点，因此节点2的f(x)值小于节点6的f(x)值，节点2排在节点6的前面）。



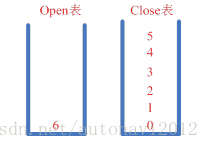
4） 从Open表取出f(x)值最小的节点（节点2），考察与节点2相连的所有节点（节点3）是否是目标节点，如果不是则估算这些节点的f(x)值，将其放进Open表，同时节点2被放入Close表。将open表中的节点按f(x)的值从小到大的顺序排列。



5）从Open表取出f(x)值最小的节点（节点3），考察与节点3相连的所有节点（节点4）是否是目标节点，如果不是则估算这些节点的f(x)值，将其放进Open表，同时节点3被放入Close表。将open表中的节点按f(x)的值从小到大的顺序排列。



6）从Open表取出f(x)值最小的节点（节点4），考察与节点4相连的所有节点（节点5）是否是目标节点，如果是则将节点4被放入Close表，然后将考察的节点（节点5）放入Close表，此时发现一条丛开始地到目的地的路径。



7） 从节点5开始往前回溯,从Close表中提取路径。

根据以上思路，A\*算法的伪代码如下：

1. A\*()
2. {
3. Open = [起始节点];
4. Closed = [];
5. while (Open表非空)
6. {
7. 从Open中取得一个节点X，并从OPEN表中删除。
8. if (X是目标节点)
9. {
10. 求得路径PATH；
11. 返回路径PATH；
12. }
13. for (每一个X的子节点Y)
14. {
15. if (Y不在OPEN表和CLOSE表中)
16. {
17. 求Y的估价值；
18. 并将Y插入OPEN表中；
19. }
20. //还没有排序
21. else if (Y在OPEN表中)
22. {
23. if (Y的估价值小于OPEN表的估价值)
24. 更新OPEN表中的估价值；
25. }
26. else //Y在CLOSE表中
27. {
28. if (Y的估价值小于CLOSE表的估价值)
29. {
30. 更新CLOSE表中的估价值；
31. 从CLOSE表中移出节点，并放入OPEN表中；
32. }
33. }
34. 将X节点插入CLOSE表中；
35. 按照估价值将OPEN表中的节点排序；
36. }//end for
37. } //end while
38. } //end function