三维空间最短路径算法设计与实现

陈东

二〇二一年五月

**Southwest Petroleum University**  
**Graduation Thesis**

|  |  |
| --- | --- |
| Grade: |  |
| Name: |  |
| Specialty: |  |
| Instructor: |  |

image image

SouthWest Petroleum University

**郑重声明**

本人呈交的学位论文, 是在导师的指导下, 独立进行研究工作所取得的成果, 所有数据、图片资料真实可靠. 尽我所知, 除文中已经注明引用的内容外, 本学位论文的研究成果不包含他人享有著作权的内容. 对本论文所涉及的研究工作做出贡献的其他个人和集体, 均已在文中以明确的方式标明. 本学位论文的知识产权归属于培养单位.  
本人签名: $\underline{\hspace{3.5cm}}$ 日期: $\underline{\hspace{3.5cm}}$

在现实生活中的许多场景中，关于给定的某一起点出发到一终点的最短路径问题是极其普遍的，可以将问题抽象为二维空间或三维空间的最短路径问题，其中二维空间的最短路径问题已经拥有了成熟的适用于不同情况的最短路径算法，如单源最短路径算法中：Dijiktra算法拥有优越的时间和空间复杂度但适用于带正权图；Bellman-Ford算法不仅适用于正权图，也适用于负权图；基于Bellman-Ford算法改进的SPFA算法不仅适用于正负权值图，也拥有优越的平均时间复杂度；多源最短路径：Floyd算法可以计算出顶点到顶点之间的最短路径；

三维空间最短路径算法，基于离散化近似计算原理，将障碍物离散到指定精度下的格点网络中，再利用A\*、SPFA算法计算出离散格点下的给定起点到终点的最短路径。由于计算的格点路径还可以进一步优化路径，因此再不断拟合格点路径为连续线段，最终算法将计算出起点到终点经过的线段路径和最短路径长度。基于给定精度计算最短路径长度，因此可以根据实际情况调整计算精度，在结果精度和计算时间和空间消耗之间取的最好的平衡。

通过分析可行算法和粗略解决思路，提出了将连续空间离散化成为格点的思路，再利用最短路径算法计算最短路径，最后拟合成为连续线段。在算法设计完成后进行了样例测试并针对测试结果进行进一步完善。最后，本算法成功解决了三维空间最短路径求解问题，至此，本算法的设计与实现工作顺利结束。

In many real-life scenarios, the shortest path problem from a given origin to an end point is extremely common and can be abstracted as a two-dimensional or three-dimensional shortest path problem, where the two-dimensional shortest path problem has mature shortest path algorithms for different situations, such as the single-source shortest path algorithm: Dijiktra algorithm has The Bellman-Ford algorithm is applicable not only to positive-weighted graphs but also to negative-weighted graphs; the improved SPFA algorithm based on the Bellman-Ford algorithm is not only applicable to positive- and negative-weighted graphs but also has superior average time complexity; the multi-source shortest path: Floyd’s algorithm can calculate the shortest path between The Floyd algorithm can calculate the shortest path from vertex to vertex;

The shortest path algorithm in 3D space is based on the principle of discretization approximation, which discretizes the obstacles into a network of lattice points with specified precision, and then uses A\* and SPFA algorithms to calculate the shortest path from a given starting point to the end point under the discrete lattice points. Since the calculated grid point path can be further optimized, the grid point path is continuously fitted as a continuous line segment, and the final algorithm will calculate the line segment path and the shortest path length from the starting point to the end point. The shortest path length is calculated based on the given accuracy, so the calculation accuracy can be adjusted according to the actual situation to get the best balance between the result accuracy and the calculation time and space consumption.

By analyzing the feasible algorithms and approximation solutions, we propose the idea of discretizing the continuous space into lattice points, and then use the shortest path algorithm to calculate the shortest path and finally fit it into continuous line segments. After the algorithm was designed, a sample test was conducted and further improved based on the test results. Finally, the algorithm successfully solves the shortest path problem in three-dimensional space, and thus the design and implementation of the algorithm are successfully completed.

# 绪论

## 论文研究背景与意义

### 选题的背景

对于许多实际生活中需要求得两个地方之间的路径和距离，并且考虑障碍物的影响，以提供后续决策等的帮助，例如房间当中两个窗户之间风的流向轨迹，在山体之间修路等问题的求解时需要计算出起点到终点之间的最短路径。对于小数据范围如房屋空间，不需要复杂的数学模型建模转换为二维模型以求解最短路径，可以在考虑一定误差范围内将障碍物转换为AABB包围盒，利用最短路径算法直接在三维空间中求解起点到终点的近似最短路径。

### 选题的技术现状

本算法通过将空间离散为若干个格子点，将障碍物近似为每一边都平行于一个坐标平面的简单六面体，通过常规最短路径算法：BFS广度优先搜索算法和A\*寻路算法求解给定起点到终点不经过障碍物的的可行路径，再拟合成若干线段，通过检查线段是否穿过障碍物实现路径拟合，最后通过动态规划计算最优路径长度。

### 选题的意义

完成本选题，是为了设计并实现算法以解决实际项目中遇到的路径长度求解问题，提供近似最短路径和欧式距离以提供路径选择参考。

## 国内外研究现状

国内外对于三维最短路径算法的研究一般基于数字高程模型（Digital Elevation Model，简称DEM），是通过用一组有序数值阵列形式表示地面高程的数据集。对于DEM给出的地面高程信息，在测绘、水文、气象等应用地面模型的领域都取得了突破性进展。本文不利用DEM模型，而是将三维空间离散为若干个格点，再应用最短路径算法在格点间，求得起点格点到终点格点的最短路径后再将格点之间的路线拟合为线段的实现方式。

## 论文组织结构

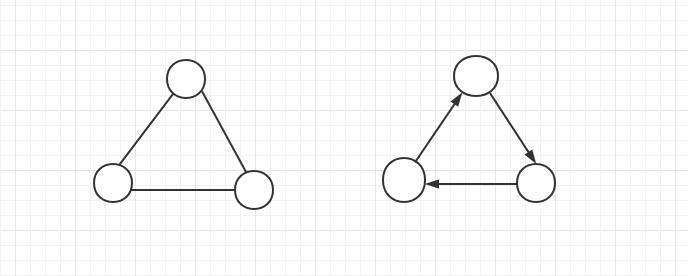
本文针对三维空间路径规划问题开展研究，首先将障碍物的aabb包围盒以某个尺度离散为具体的格点，对格点进行不可访问标记处理；随后基于所给的起点，采用最短路径算法求得到终点所经过的格点序列，提出一种基SPFA和A\*的最短路径算法实现，同时对比了基于不同最短路径算法之间的空间-时间开销。本文的组织结构如下：  
第一章：本章主要介绍研究课题的背景和意义，三维空间最短路径研究现状及发展的趋势，并讨论了国内外在相关方面的研究状况及其应用前景，给出本文的主要贡献、创新性和组织结构。  
第二章：本章主要介绍了最短路径模型、二维和三维空间最短路径的区别和联系、图的概念、存储方式并对比了不同存储方式的优缺点。最后，总结和阐述了三维空间最短路径的规划方法。  
第三章：本章主要对三维空间最短路径求解算法进行分类总结，提出将三维空间离散为格点后，在格点图上进行BFS、A\*、SPFA等算法求解最短路径的效率和开销。  
第四章：对全文进行了总结，对未来基于三维空间的最短路径算法深入研究作出了展望。

# 最短路径优化算法研究综述

## 图

### 图的基本概念

图论，是组合数学分支，和其他数学分支也有密切关系，如群论、拓扑学。图是图论的主要研究对象，图是给点若干顶点以及两顶点所构成的图形，这种图形可以用来描述事物之间的某种特定关系。 顶点可以用来描述某种事物，连接两顶点可以代表事物之间存在的某种关系，如顶点代表学习任务，连接顶点代表学习该任务之前需要学习的其他任务。图论起源于著名的柯尼斯堡七桥问题即著名的欧拉图一笔画问题。该问题于1736年被欧拉解决，因此普遍认为欧拉是图论的创始人。通常在图论中，以有序对，其中V是点集；是边集，由所有顶点序列构成，其中一条边中的x，y被称作边的端点； 图的分类分为有向图和无向图，其中有向图是指给图的每条边都规定一个方向，其边称为有向边，以；相反，边没有方向的图称为无向图，无向边以表示，则在边集会同时存在。并且V，E的元素个数通常都是有限的，图的阶数是其顶点个数，图的边数是。每条边都连接两个不同的顶点且没有两条不同的边连接一对相同顶点的图称为**简单图**，可能有**多重边**连接同一对顶点的图成为**多重图**。其中最短路径问题、最小生成树问题、关键路径问题都属于是基于图上的问题，并提出了相应的解决方案。如图[2.1](#fig:graph_ex)分别是无向图和有向图的一个实例。其中，对于实际问题中每条边代表了从顶点到所需要的权值的图叫做带权图，其中边上的权值也叫边的长度。

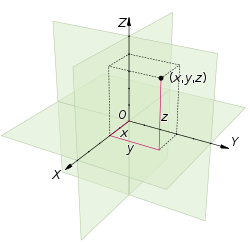


无向图和有向图实例

**三维空间**是类似我们生存的空间的数学模型，由长、宽、高三个维度，也即是三维欧几里得空间，定义欧几里得平面为装备了内积的二维实数的向量空间。满足

1. 在这个向量空间中的向量对应欧几里得平面中的点，
2. 在向量空间中的加法运算对应于平移，
3. 内积蕴含了角和距离的概念，它可被用来定义旋转。

如图[2.2](#fig:3axis)就是三维欧几里得空间的示意图，三个轴相互垂直，点的坐标为其到三个轴的投影。



三维欧几里得空间

### 图的存储表示

一般在计算机中存储图信息，有以下几种方式

1. 邻接矩阵，使用多维数组记录点与点之间的边权信息；
2. 邻接表，使用链表等数据结构维护该点到其他点的边信息；
3. 十字链表，将有向图的邻接表和逆邻接表的结合；
4. 邻接多重表，将无向图的邻接表和逆邻接表的结合；

邻接矩阵是应用多维数组保存的点和点之间的关系，因此可以很容易的判断两个点之间的关系，但缺点在于占用空间大，尤其对于多维空间时保存信息时需要占用大量的存储空间；邻接表使用链表保存点所连接的边信息，节省了空间，但不易得到两点之间的关系；十字链表和邻接多重表都能节省空间、快速判断点与点之间的关系，但缺点是实现起来比较复杂。

## 路径规划

首先，在图中的**路径**是指从顶点到顶点的一个序列，有时简写为，其中表示起点终点为及；称为路径的长度，即经过的边的数量；称为路径的起点；称为路径的终点。当没有必要区分多重边时，就用顶点序列表示通路，其中对于，这种记法仅仅指出通路所经过的顶点。其中论文求解的最短路径是指：若从顶点到顶点之间长度最短的路径，即是加权图中一条路径的长度是这条路径上各条边的总和最小。

## 最短路径规划算法

最短路径问题是图论研究的一个经典算法问题，目的在于求出两顶点之间的最短路径。算法具体的形式有

* 确定起点的最短路径问题，也即已知起始起点求最短路径的单源最短路径问题；
* 确定终点的最短路径问题，与确定起点的问题相反，该问题是已知终结顶点，求最短路径的问题；
* 确定起点终点的最短路问题，即已知起点和终点，求两顶点之间的最短路径；
* 全局最短路径问题，也叫多源最短路问题，求出图中所有的最短路径。

用于解决最短路径问题的算法叫做“最短路径算法”，常用的路径算法有：

* BFS算法；
* Dijkstra算法；
* A\*算法；
* Bellman-Ford算法；
* SPFA算法（Bellman-Ford算法的改进版本）；
* Floyd算法；

广度优先搜索(BFS)和深度优先搜索(DFS)会从起点开始求解出可行路径，但由于DFS需要求解所有路径才知道最短路径，而BFS算法的特点是逐层扩展的，决定了其不需要遍历所有的路径就能找到最短路径。Dijkstra、A\*、Bellman-Ford、SPFA算法都是用于解决单源最短路径，其中Dijskra用于正权路径最短路径求解，Bellman-Ford、SPFA、A\*可用于正负权路径最短路径求解，Floyd用于解决多源最短路径；其中不同算法在基于不同的图结构上拥有不同的时间、空间复杂度，其中Dijkstra和A\*在寻路算法中应用广泛，由于A\*使用了估值函数，会更倾向的选择路线，更加节省内存空间。

## 本章小结

本章主要介绍了图结构的定义以及常见的计算机存储图结构的数据结构，以及介绍了路径定义和最短路径的定义和求解算法。对于常见的图结构，使用邻接矩阵和邻接表在不同的应用领域都有很好的发挥，如邻接矩阵可以很快的得到顶点与顶点之间的关系，但需要记录每个顶点到其他所有顶点的关系，占用的内存空间较大；邻接表使用链表结构存储每个顶点所连接的边信息，可以节省大量的空间，但不容易得到顶点与顶点的关系。随后介绍了关于对图上最短路径求解的常用算法，如单源最短路径算法，有用于正权图最短路径求解的Dijskra算法，正负权都适用的Bellman-Ford、SPFA算法，基于估值函数优化的A\*算法在寻路方面有广泛的应用；以及用于多源最短路径求解的Floyd算法。

# 最短路径算法实现

## 最短路径算法分析

在二维空间中，通常是基于二维垂直的欧几里得空间，其中x轴和y轴垂直，顶点通常以类似经纬度形式给出，如 (x,y) 代表该顶点分别该点坐标分别在x轴、y轴上的长度分量，点 (a,b) 在 x 轴的分量为 a ，在 y 轴的分量为 b ；其中顶点之间会与若干条路径相连，其中路径不能经过相关障碍物，路径的权值可以是该条路径的长度或该条路径所需费用，记为 W(i,j) 表示第 i 个顶点到第 j 个顶点的边的权值。例如在运输网络中，顶点代表的是不同城市的集散地，顶点之间的路径就是城市之间的交通网，路径的权值代表该条交通路线所需要的时间，则所求的顶点到顶点的最短路径即是最短时间的路线。

### 基于Dijkstra算法的最短路径求解

Dijkstra该算法基于贪心的思想，解决了图 G=<V,E> 上带权的单源最短路径问题，通过设置一顶点集合 S ，在集合S中所有的顶点与源点s之间的最终最短路径权值均已被计算出来。算法反复选择最短路径估计最小的点并将 u 加入到 S 中，最终计算出源点到其他所有点的最短距离。举例来说，若图中的顶点代表城市，而边上的权值表示城市间开车行车的距离，该算法可以用来找到两个城市之间的最短路径。但是Dijkstra算法并不能有效处理带有负权边的图。

算法描述：Dijkstra算法通过保留目前为止所找到的每个顶点从 s 到 v 的最短路径来工作。初始时，原点s的路径权重被赋为0。同时把所有顶点的路径长度设为无穷大，即表示我们不知道任何通向这些顶点的路径。当算法结束后， d[v] 中存储的便是从s到v的最短路径，或者如果路径不存在的话是无穷大。

松弛操作是Dijkstra的基础操作：如果存在一条从u到v的边，那么从s到v的一条新路径是将边添加到从s到u的路径尾部来拓展一条从s到v的路径。这条路径的长度是。若这个值比目前已知的的值要小，那么可以用这个值来替代当前中的值。松弛边的操作一直运行到所有的都代表从s到v的最短路径的长度值。由于基于松弛操作，因此若存在负权值边中的负环时会重复松弛，导致无法正确求解出最短路径。

算法维护两个顶点集合openList和clostList，集合openList保留所有已知实际最短路径值的顶点，而集合clostList则保留其他所有顶点。集合S初始状态为空，而后每一步都有一个顶点从Q移动到S。这个被选择的顶点是Q中拥有最小的值的顶点。当一个顶点 u 从 Q 中转移到了 S 中，算法对u的每条外接边进行松弛。

算法的步骤如下

1. 输入边全为正权的图，G中带有顶点V=和若干边
2. 设置一个待检测列表clostList和一个不需检测列表clostList；
3. 将起始点startPoint加入openList中；
4. 初始化距离向量 d ，其中startPoint的距离为0，其他全为正无穷；
5. 检测openList，找出 d 值最小的一个点，将此节点作为当前节点(curNode)，并加入closeList中不再检测；
6. 若该点等于终点endPoint，此时终止算法，已找到最短路径，返回该点的距离向量；
7. 否则遍历curNode的相邻节点，若该节点是不可通过的、或在closeList中的、或超过边界的，则跳过；
8. 保存该节点的parent为curNode，计算其距离向量值 d ，再保存到openList中；
9. 若此时openList为空，算法结束，未找到起点到终点的最短路径；
10. 若openList不为空，回到步骤5；

算法中的步骤8，计算相邻节点(Node)的距离向量 d 时，即是算法的松弛操作，具体计算方法是：，通过不断的将所有点加入openList来松弛到其他顶点的距离向量，最后就能计算出从起点到终点的最短路径了。

例如使用该算法来寻找两个城市之间的最短路径，整个图是城市之间的交通网，此时顶点是城市，顶点之间的路径是城市之间的交通路线，路径的权值是行驶所需的时间，因此两个城市之间可能有多条权值不同的路径，此时使用Dijkstra算法求解得到的最短路径即是从起点出发到其他顶点的所需要的最短时间的路线。

### 基于A\*算法的最短路径求解

A\*搜索算法综合了最良优先算法(Best-first search)和Dijkstra算法的优点：在进行启发式搜索提高算法效率的同时，可以保证找到一条最优路径（基于评估函数）。

该算法中，以 g(n) 表示从起点到任意顶点n的实际距离， h(n) 表示任意顶点n到目标顶点的估算距离，那么A\*算法的估算函数为 f(n)=g(n)+h(n) ，这个公式遵循以下特性:

* 如果 g(n) 为0，即只计算任意顶点n到目标的评估函数 h(n) ，而不计算起点到顶点n的距离，则算法转化为使用贪心策略的最良优先搜索，速度最快，但可能得不出最优解；
* 如果 h(n) 不大于顶点n到目标顶点的实际距离，则一定可以求出最优解，而且 h(n) 越小，需要计算的节点越多，算法效率越低，常见的评估函数有-欧几里得距离、曼哈顿距离、切比雪夫距离；
* 如果 h(n) 为0，即只需求出起点到任意顶点n的最短路径 g(n) ，而不计算任何评估函数 h(n) ，则转化为单源最短路径问题，即Dijkstra算法，此时需要计算最多的顶点。

其步骤如下

1. 设置一个待检测列表openList和一个不需检测列表clostList；
2. 把起始点startPoint加入openList中；
3. 检测openList，找出其中 f 值最小的一个点，将此节点作为当前节点(curNode)，并加入closeList中不再检测；
4. 若该点等于终点endPoint，此时终止算法，已找到最短路径；
5. 否则遍历curNode周围的节点，若该节点是不可通过的、在closeList中的、超过边界的，则跳过；
6. 保存该节点的parent，计算 f 值，再保存到openList中；
7. 若此时openList为空，算法结束，未找到目的点；
8. openList不为空，回到步骤3；

A\*算法通过启发式搜索，即评估函数 h(n) ，进一步提高了寻找最短路径的效率，减小了无用点的遍历过程，可以大大的节省时间和空间的消耗。

## 三维最短路径

对于类似实际空间的三维空间，由于三维空间当中顶点的数量较多，且存在复杂的障碍物关系，使得在三维空间中求解最短路径算法对于空间复杂度和时间复杂度之间存在一定的受限，在三维空间当中求解最短路径更偏向于求解近似最短路径，以满足时间和空间的要求，因此算法主要解决在一定的精度误差内近似求解出给出起点到终点之间的最短路径长度及所经过的顶点路径。 假设空间离散时精度为 precision ，即代表实际空间中的1单位长度对应离散化空间的 precision 单位长度。 则在实际空间中的点坐标 (x,y,z) 可以离散化为格子点坐标，则我们需要对起点、终点和障碍物都完成离散化的过程，然后在离散化空间中求解起点到终点的最短路径。

### 数据处理

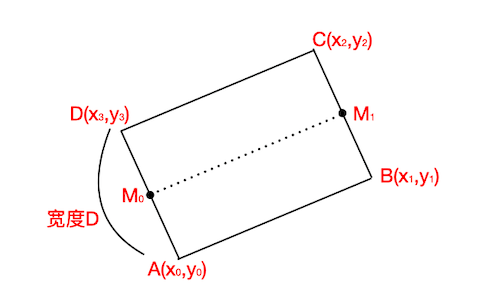
首先，根据实际空间的障碍物，模型成若干个如图的长方体，其中长方体满足底面为一个矩形，但可能长宽不平行于x、y轴，高平行于三维空间的z轴。数据中会以实数给出底面矩形平行的两边的中点，与之垂直的边的长度 D ，以及该立方体的z轴范围。通过给出的数据可以计算得出该长方体障碍物所对应的顶点。 根据实际空间模型，需要首先按照指定的精度将实际空间的模型划分到指定精度的空间，将空间划分为若干个离散化的格点，通过计算该精度下障碍物所经过的格子点，这些障碍物的离散格点都是在计算最短路径时不可以通过的。

根据障碍物底面为一个矩形，其满足长与宽垂直，则设两中点所在直线为矩形的长，则如图[3.1](#fig:obstacle_bottom)，设该矩形的四个顶点分别为、、、且满足相邻的关系，根据直线的表达式，和可以计算出与之垂直的直线的表达式为：

设，则原式为。且矩形的宽为 D ，则，故有。 将代入方程，可以得到与相邻的顶点A、D的x坐标分别为

将计算出来的坐标代入可得该点对应的y坐标。相似地，计算与相邻的两顶点时，将与的直线方程所经过的点修改为，即该直线方程为，类似的满足与相邻的顶点B、C满足的x坐标的关系为

将计算的结果代入相应的直线方程当中即可。



障碍物底面

特别注意特殊判断长宽平行与轴的情况，此时可以直接通过的坐标计算四个顶点的坐标，即四个顶点的坐标分别为。 具体的代码如图[[code:obstacle\_vertice\_process]](#code:obstacle_vertice_process)，通过if-else判断情况来具体计算，最后通过重载结构体的读入来实现使用c++的cin就可以读入障碍物数据时计算顶点并存储在结构体当中，方便后续使用。 其中结构体当中的conveyByPrec函数是将原始数据下的障碍物离散到指定precision精度下的处理后的障碍物。

struct node {  
 point2D vec[4];  
 double z0, z1;  
 /\*friend ostream & operator <<(ostream & os,const node & p){  
 return os << "[" << p.lb << ", " << p.ru << "]";  
 }\*/  
 process\_data\_node conveyByPrec(double precision) {  
 process\_data\_node node;  
 node.vec[0] = vec[0] / precision;  
 node.vec[1] = vec[1] / precision;  
 node.vec[2] = vec[2] / precision;  
 node.vec[3] = vec[3] / precision;  
 node.z0 = z0 / precision;  
 node.z1 = z1 / precision;  
 return node;  
 }  
 friend istream & operator >>(istream &in, node& p){  
 double x0, y0, x1, y1, d;  
 in >> x0 >> y0 >> x1 >> y1 >> d >> p.z0 >> p.z1;  
 //特判平行的状况  
 if (y0 == y1 || x0 == x1) {  
 p.vec[0] = {x0, y0 - d / 2};  
 p.vec[1] = {x1, y1 - d / 2};  
 p.vec[2] = {x1, y1 + d / 2};  
 p.vec[3] = {x0, y0 + d / 2};  
 }  
 else {  
 double k = (x1 - x0) / (y1 - y0);  
 double delta = (d / 2) / sqrt(1 + k \* k);  
   
 p.vec[0] = {x0 - delta, y0 + k \* delta};  
 p.vec[1] = {x1 - delta, y1 + k \* delta};  
 p.vec[2] = {x1 + delta, y1 - k \* delta};  
 p.vec[3] = {x0 + delta, y0 - k \* delta};  
 }  
 return in;  
 }  
};

通过处理数据，使得可以将最短路径算法应用在三维空间当中，通过控制离散化的精度来实现控制最短路径的精度，离散后格点的标记来实现不可访问情况的判断，达到将实际空间离散化到指定精度的理想空间，从而计算出格点组成的最短路径。

### 相交判断

通过计算出格点组成的最短路径后，由于存在三角形定则，即三角形两边之和大于第三边，所以在计算的格点路径后，是可以通过运用三角形定则拟合计算出更短的路径，此时就需要判断所拟合的新路径是否经过了障碍物。此时问题可以抽象为线段与长方体是否相交的判定问题，根据线段的向量表示法，有线段:，若存在与某个障碍物相交的情况，即存在k属于0～1之间，使得该点在障碍物之内。 考虑障碍物：底面，高的范围。由于障碍物的高与z轴平行，故只需要考虑该障碍物由无数个矩形在z轴上堆叠而成。首先，我们讨论如何判断一个二维点坐标是否位于底面矩形当中，设该点坐标为，若该点在矩形内，则满足，若该式子成立则说明该点在边AB和边CD之间；同理，还需要成立，即该点在边DA和BC之间。两个条件需要同时满足，则说明该点在四条边之内，即在该矩形范围内。

将问题放大到三维空间当中，该线段若与该障碍物有交点，则至少存在一个可行的k解满足E的坐标在障碍物内，则我们求出要与该障碍物相交时k值所取的范围，若存在有效范围，则表明该线段与该障碍物相交。 首先，对于z轴相交，设为向量的方向向量，则k的范围应在，对于障碍物的z轴所计算的k的取值为：，若小于0时交换的值。则存在z轴可行解，否则无解。代码如下[[code:check\_insect\_zaxis]](#code:check_insect_zaxis)，首先初始化k取值范围为[0,1]，通过分别计算对应的k取值来更新。

bool check\_insect(pointInt P, pointInt Q, process\_data\_node box) {  
 double t1 = 0.0, t2 = 1.0;  
 pointInt R = Q - P;  
 //check the k of z-axis  
 if (R.z == 0) {  
 if (!(box.z0 <= P.z && P.z <= box.z1))  
 return false;  
 }  
 else {  
 double tmp1 = 1.0 \* (box.z0 - P.z) / R.z;  
 double tmp2 = 1.0 \* (box.z1 - P.z) / R.z;  
 if (R.z < 0) swap(tmp1, tmp2);  
 t1 = max(t1, tmp1);  
 t2 = min(t2, tmp2);  
 }  
 if (t1 > t2)   
 return false;

对于x、y所决定的k值的范围，需要计算的是该点在矩形内部的情况，则可以用到之前我们所讨论到的公式

和

公式[[con:innerABandCD]](#con:innerABandCD)和[[con:innerDAandBC]](#con:innerDAandBC)两者需要同时成立。针对可能出现该点在xy平面内是一个点的情况，即一条平行与z轴的线段，此时利用公式[[con:innerABandCD]](#con:innerABandCD)和[[con:innerDAandBC]](#con:innerDAandBC)对该点做一个计算，若两公式都满足则说明该线段在xy平面的点在障碍物底面中之中，该部分代码如下[[code:check\_insect\_onepoint]](#code:check_insect_onepoint)。

//special judge one point situation  
if (R.x == 0 && R.y == 0) {  
 point2DInt E(P.x, P.y);  
 int tmp1 = ((box[1] - box[0]) \* (E - box[0])) \* ((box[3] - box[2]) \* (E - box[2]));  
 int tmp2 = ((box[0] - box[3]) \* (E - box[3])) \* ((box[2] - box[1]) \* (E - box[1]));  
 if (tmp1 >= 0 && tmp2 >= 0)  
 return true;  
 return false;  
}

对于线段能到达的点坐标E，有。将该点E坐标代入公式[[con:innerABandCD]](#con:innerABandCD)中有

我们设k的系数，后的常数为，则公式可化简为。

设k的系数为，后的常数为，则公式可化简为。 则等价于方程，即是二次函数求解的问题。 关于二次函数根求解的问题，我们需要讨论系数是否为0，以及特判二次函数出现非实数解的情况。若不存在这些情况，则对于二次函数，二次函数根的计算公式:

故我们将公式[[con:Quadratic\_functions]](#con:Quadratic_functions)应用在方程上，可以得到该方程的解为

我们将之前假设的代入该公式既可以求出对于公式[[con:innerABandCD]](#con:innerABandCD)限制下k的取值范围。使用该求解值更新基于z范围求出的k范围，若新的则存在可行解，否则无解。代码如下[[code:check\_insect\_firstcross]](#code:check_insect_firstcross)，代码首先判断了出现系数为0时二次函数退化为一次函数的情况，下文将进行分析，否则计算二次函数对应的k值用来更新并判断。

//judge ABxAE\*CDxCE>=0  
double a1 = 1.0 \* (Q.y - P.y) \* (box[1].x - box[0].x) - 1.0 \* (Q.x - P.x) \* (box[1].y - box[0].y);  
double b1 = 1.0 \* (P.y - box[0].y) \* (box[1].x - box[0].x) - 1.0 \* (P.x - box[0].x) \* (box[1].y - box[0].y);  
double a2 = 1.0 \* (Q.y - P.y) \* (box[3].x - box[2].x) - 1.0 \* (Q.x - P.x) \* (box[3].y - box[2].y);  
double b2 = 1.0 \* (P.y - box[2].y) \* (box[3].x - box[2].x) - 1.0 \* (P.x - box[2].x) \* (box[3].y - box[2].y);  
double delta = 1.0 \* (a1 \* b2 + a2 \* b1) \* (a1 \* b2 + a2 \* b1) - 4.0 \* a1 \* a2 \* b1 \* b2;  
if (delta < 0) return false;  
if (a1 == 0 && a2 == 0) {  
 if (b1 \* b2 < 0)  
 return false;  
}  
else if (a1 == 0) {  
 double tmp = -1.0 \* b2 / a2;  
 if (a2 > 0) t1 = max(t1, tmp);  
 else t2 = min(t2, tmp);  
}  
else if (a2 == 0) {  
 double tmp = -1.0 \* b1 / a1;  
 if (a1 > 0) t1 = max(t1, tmp);  
 else t2 = min(t2, tmp);  
}  
else {  
 double x1 = (-1.0 \* (a1 \* b2 + a2 \* b1) - sqrt(delta)) / (2.0 \* a1 \* a2);  
 double x2 = (-1.0 \* (a1 \* b2 + a2 \* b1) + sqrt(delta)) / (2.0 \* a1 \* a2);  
 if (x1 > x2) swap(x1, x2);  
 t1 = max(t1, x1);  
 t2 = min(t2, x2);  
}  
if (t1 > t2)   
 return false;

对于公式[[con:innerDAandBC]](#con:innerDAandBC)，相似的有，我们将点E坐标代入有

同样的，设k的系数，后的常数为，则公式可化简为。

设k的系数为，后的常数为，则公式可化简为。 故我们将新的的值代入公式[[con:kBaseOnQuadFun]](#con:kBaseOnQuadFun)中计算出对于公式[[con:innerDAandBC]](#con:innerDAandBC)限制下的k的取值范围。使用新求解的k值更新由公式[[con:innerABandCD]](#con:innerABandCD)更新的k值，若最后求解的k值仍满足，则说明线段存在与该障碍物相交的情况。 具体代码如下[[code:check\_insect\_secondcross]](#code:check_insect_secondcross)，代码同样先判断了退化的情况，再计算的二次函数对应k取值来更新并判断。若最后更新后的k取值仍满足，则说明存在可行k解使得存在该线段与障碍物相交，此时代码true表示产生碰撞。

//judge DAxDE\*BCxBE>=0  
a1 = 1.0 \* (Q.y - P.y) \* (box[0].x - box[3].x) - 1.0 \* (Q.x - P.x) \* (box[0].y - box[3].y);  
b1 = 1.0 \* (P.y - box[3].y) \* (box[0].x - box[3].x) - 1.0 \* (P.x - box[3].x) \* (box[0].y - box[3].y);  
a2 = 1.0 \* (Q.y - P.y) \* (box[2].x - box[1].x) - 1.0 \* (Q.x - P.x) \* (box[2].y - box[1].y);  
b2 = 1.0 \* (P.y - box[1].y) \* (box[2].x - box[1].x) - 1.0 \* (P.x - box[1].x) \* (box[2].y - box[1].y);  
delta = 1.0 \* (a1 \* b2 + a2 \* b1) \* (a1 \* b2 + a2 \* b1) - 4.0 \* a1 \* a2 \* b1 \* b2;  
if (delta < 0 || a1 \* a2 == 0) return false;  
if (a1 == 0 && a2 == 0) {  
 if (b1 \* b2 < 0)  
 return false;  
}  
else if (a1 == 0) {  
 double tmp = -1.0 \* b2 / a2;  
 if (a2 > 0) t1 = max(t1, tmp);  
 else t2 = min(t2, tmp);  
}  
else if (a2 == 0) {  
 double tmp = -1.0 \* b1 / a1;  
 if (a1 > 0) t1 = max(t1, tmp);  
 else t2 = min(t2, tmp);  
}  
else {  
 double x1 = (-1.0 \* (a1 \* b2 + a2 \* b1) - sqrt(delta)) / (2.0 \* a1 \* a2);  
 double x2 = (-1.0 \* (a1 \* b2 + a2 \* b1) + sqrt(delta)) / (2.0 \* a1 \* a2);  
 if (x1 > x2) swap(x1, x2);  
 t1 = max(t1, x1);  
 t2 = min(t2, x2);  
}  
if (t1 <= t2) return true;  
return false;

针对可能出现的二次函数系数为0的情况，即公式退化为一次函数的正负问题，如，此时特判系数为0的各种情况，计算一次函数下k的取值即可，代码如图[[code:check\_insect\_firstcross]](#code:check_insect_firstcross)的8行至20行。对于出现非实数解的情况，即时，此时表明不存在有效k的取值的点使之在障碍物内，故该线段与障碍物不相交。

至此，已经完成了三维空间内线段与障碍物相交的讨论，利用通过离散化格点求解的格子点路径，不断检查格子点之间是否存在不与障碍物相交也能相连且距离更短的情况，来进一步优化路径。

### 最短路径算法

三维空间最短路径算法的实现，在指定精度后，将实际空间离散到指定精度的空间，因此算法需要求解出离散空间中从起点到终点的格点最短路径，再利用上述讨论的相交算法来拟合路径，最后给出该精度下近似的最短路径和所通过的格点线段。

#### 基于BFS的三维最短路径算法

广度优先搜索(BFS)算法，又叫做宽度优先搜索，是一种在图上的搜索算法。算法将从根节点开始，沿着树的宽度遍历树的节点。在图上时，可以视为根节点为起点的一棵树。BFS是一种暴力算法，目的是系统的展开并检查图中的所有节点，以找寻结果。算法的实现一般采用open-closed表，所有因为展开节点而得到都会被加进一个先进先出的队列中，其邻居节点尚未检验的节点会被存放在一个称为open的容器，而被检验过的节点则被存放在被称为closed的容器中。其实现方法为：

1. 首先将根节点放在队列中。
2. 从队列中取出现在的第一个节点，并检查。
   * 如果找到目标，则结束搜索并返回结果。
   * 否则将它所有还没有检查过的相邻子节点加入队列中。
3. 若队列为空，表示检查了所有点都没有找到目标。结束搜索并返回未找到目标。
4. 重复步骤2.

在离散后的三维空间中，将把每一个整数格点视为顶点，即，则对于顶点相邻的顶点有分别在x轴、y轴、z轴相邻的6个顶点：。 则基于BFS的三维空间最短路径算法实现，是基于从起点格点逐渐向外发散并记录所经过的路径，直到找到终点。算法需要遍历大量的点，若起点与终点之间三维坐标差的最大值为，则算法至少需要遍历个点，则算法的时间复杂度为，空间复杂度为。 搜索算法初始化部分如代码[[code:bfs\_init]](#code:bfs_init)所示，首先定义了距离向量dist，用来存储起点距离该格点的最短路径的长度；方向向量dir，用来找到现在顶点相邻的顶点；以及使用数组q和指针front和rear实现队列的功能，并能将顶点保存起来用来回溯路径；pre数组存储该顶点nowNode的前驱顶点preNode，即是BFS时nowNode是通过preNode搜索到的且经过格子点数量最少的情况时的前驱顶点。

namespace BFS {  
 //define  
 int dist[maxn][maxn][maxn];  
 int dir[6][3] = {{-1, 0, 0}, {1, 0, 0}, {0, -1, 0}, {0, 1, 0}, {0, 0, -1}, {0, 0, 1}};  
 bool set\_dist(point p, point pre) {  
 if (dist[p.x][p.y][p.z] > dist[pre.x][pre.y][pre.z] + 1) {  
 dist[p.x][p.y][p.z] = dist[pre.x][pre.y][pre.z] + 1;  
 return true;  
 }  
 return false;  
 }  
 point q[maxm]; int front, rear;  
 int pre[maxm], min\_dist; bool min\_dist\_flg;  
 vector<int> ed\_idxs;  
 //initial  
 void init() {  
 memset(dist, 0x3f, sizeof(dist));  
 dist[st.x][st.y][st.z] = 0;  
 vis[st.x][st.y][st.z] = 1;  
 front = 0; rear = -1;  
 q[++rear] = st; pre[0] = -1;  
 min\_dist\_flg = false;  
 ed\_idxs.clear();  
 }

初始化完成后，算法使用数组实现的队列q，front指针表示当前节点在队列中的位置，rear指针表示队列的最后一个节点的位置，当时表示队列不为空。则算法通过不断取出队列的当前节点，并检查是否为终点，或者跳过路径长度以及大于了已经找到的最短路径长度的节点，对搜索做剪枝。随后遍历该节点的相邻节点，若相邻节点没有超出边界且可以访问，则更新其距离向量，并将其加入到队列中。

bool bfs() {  
 cerr << "正在搜索最短路径。。。" << endl;  
 while (front <= rear) {  
 point now = q[front++];  
 if (min\_dist\_flg && dist[now.x][now.y][now.z] > min\_dist) continue;  
 if (now == ed) {  
 if (!min\_dist\_flg || dist[now.x][now.y][now.z] <= min\_dist) {  
 min\_dist\_flg = true;  
 min\_dist = dist[now.x][now.y][now.z];  
 ed\_idxs.push\_back(front - 1);  
 }  
 continue;  
 }  
   
 for (int i = 0; i < 6; ++i) {  
 point nx(now.x + dir[i][0], now.y + dir[i][1], now.z + dir[i][2]);  
 if (check\_outofbound(nx)) continue;  
 if (check\_insideofobstacles(nx)) continue;  
 if (!set\_dist(nx, now)) continue;  
 q[++rear] = nx;  
 vis[nx.x][nx.y][nx.z] = 1;  
 pre[rear] = front - 1;  
 }  
 }  
 return ed\_idxs.size() != 0;  
 }

若算法返回true则代表至少存在一条从起点格点到终点格点的格点最短路径，此时需要我们对路径做一个拟合，进一步优化路径长度。根据上述障碍物相交判断中所实现的函数check\_insect做该条路径上顶点到顶点的拟合操作，即若直接在这两个顶点之间连成线段不经过障碍物且距离更短则直接连接这两个顶点为一条线段而不经过中途格点。此时拟合算法所需要的时间与所找到的最短路径长度有关，对于BFS搜索算法，平均最短路径上的格子节点数量在kn个，即n的常数级别的个数。 由于路径上的每个顶点都至多与路径上的所有顶点做一次障碍物相交判断算法，且使用一个长度为路径上的顶点个数的数组作为距离向量，则算法的时间复杂度在，空间复杂度为。 具体的代码如代码[[code:bfs\_checkinsect]](#code:bfs_checkinsect)，check\_segement\_cross\_all\_obstacles(P,Q)是对所有的障碍物做相交判断，若出现与一个障碍物与该线段相交则返回非-1，算法通过使用定义的pre数组逐渐找到整条路径并记录到tmp链表中，对每找出的一条路径做一次拟合，dp\_dist记录该条路径上顶点的距离向量，若此条路径长度更短，则记录在path链表中。

vector<point> path;  
double optimal\_dist; bool optimal\_dist\_flg;  
  
vector<double> dp\_dist;  
vector<int> tmp, tmp\_pre;  
void getpath() {  
 if (!bfs()) {  
 cerr << "未找到路径！" << endl;  
 return;  
 }  
 cerr << "共找到" << ed\_idxs.size() << "条路径！开始拟合..." << endl;  
 optimal\_dist\_flg = false;  
 for (int k = 0; k < ed\_idxs.size(); ++k) {  
 int ed\_idx = ed\_idxs[k];  
 cerr << "正在拟合第" << k + 1 << "条路径..." << endl;  
 tmp.clear();  
 for (int idx = ed\_idx; idx != -1; idx = pre[idx]) {  
 tmp.push\_back(idx);  
 }  
 reverse(tmp.begin(), tmp.end());  
 dp\_dist.clear();  
 dp\_dist.resize(tmp.size());  
 tmp\_pre.clear();  
 tmp\_pre.resize(tmp.size());  
 for (int i = 0; i < tmp.size(); ++i) {  
 if (!i) dp\_dist[i] = 0;  
 else dp\_dist[i] = dp\_dist[i - 1] + get\_dist(q[tmp[i]], q[tmp[i - 1]]);  
 tmp\_pre[i] = i - 1;  
   
 for (int j = 0; j < i; ++j) {  
 if (check\_segement\_cross\_all\_obstacles(q[tmp[i]], q[tmp[j]]) == -1) {  
 if (dp\_dist[j] + get\_dist(q[tmp[i]], q[tmp[j]]) < dp\_dist[i]) {  
 dp\_dist[i] = dp\_dist[j] + get\_dist(q[tmp[i]], q[tmp[j]]);  
 tmp\_pre[i] = j;  
 }  
 }  
 }  
 }  
 cerr << "拟合完毕，该条路径长度为：" << dp\_dist[tmp.size() - 1] << endl;  
 if (!optimal\_dist\_flg || dp\_dist[tmp.size() - 1] < optimal\_dist) {  
 optimal\_dist\_flg = true;  
 optimal\_dist = dp\_dist[tmp.size() - 1];  
 path.clear();  
 for (int i = tmp.size() - 1; i != -1; i = tmp\_pre[i]) {  
 path.push\_back(q[tmp[i]]);  
 }  
 reverse(path.begin(), path.end());  
 }  
 }  
 cerr << "找到经过" << path.size() << "个点的路径";  
 for (int i = 0; i < path.size(); ++i) {  
 if (!i) cerr << path[i];  
 else cerr << "->" << path[i];  
 }  
 cerr << endl;  
 cerr << "该条路径的长度为：" << optimal\_dist << "，起点到终点的欧式距离为：" << get\_dist(st, ed) << endl;  
}

至此已经完成了使用BFS算法对三维空间带有障碍物寻找最短路径的设计和实现，如分析中拟合算法的时间复杂度平均为会小于寻路算法的时间复杂度，拟合算法的空间复杂度会小于寻路算法的空间复杂度，因此整个算法的时间复杂度和空间复杂度都为。当时间空间较大或离散精度较大时，整个算法运行时间和占用空间不太理想。

#### 基于A\*的三维最短路径算法

正如章节[3.1.2](#section:A*algorithm_exp)中对A\*算法的分析，由于A\*算法采用了估价函数h(n)进行启发式搜索，提高了找到路径的时间和空间的效率，并且可以保证找到基于估价函数的最优路径。对于估价函数h(n)，我们采用欧几里得距离，即$nowNode(x\_0,y\_0,z\_0),\\endNode(x\_1,y\_1,z\_1)$之间的欧几里得距离为$d(nowNode,endNode)=\\\sqrt{(x\_0-x\_1)^2+(y\_0-y\_1)^2+(z\_0-z\_1)^2}$。则g(n)表示的是起点到任意顶点n的最短路径长度，估算函数即为f(n)=g(n)+h(n)。 代码如代码[[code:astar\_init]](#code:astar_init)所示，结构体heap\_node用来存储函数和当前节点在队列中的索引，并重载了小于和大于运算符配合优先队列priority\_queue以实现大顶堆；链表arr记录的是队列遍历的点，dist向量存储的是该点的最短路径长度，pre向量存储的是该点在最短路径中前驱节点的编号，paths链表存储的是最后计算出的最短路径所经过的格子节点，record哈希表存储的是三维空间内的格子点对应在队列元素arr中的索引，\_\_init\_\_函数实现算法的初始化操作。

namespace Astar {  
 //define  
 struct heap\_node {  
 double h, g;  
 int id;  
 heap\_node() {}  
 heap\_node(double \_h, double \_g, int \_id) : h(\_h), g(\_g), id(\_id) {}  
 bool operator <(const heap\_node& rhs) const {  
 return h + g < rhs.h + rhs.g;  
 }  
 bool operator >(const heap\_node& rhs) const {  
 return h + g > rhs.h + rhs.g;  
 }  
 };  
 priority\_queue<heap\_node, vector<heap\_node>, greater<heap\_node> > Q;  
 vector<pointInt> arr;  
 vector<double> dist;  
 vector<int> pre;  
 vector<pointInt> paths;  
 double min\_dist;  
 map<pair<int, pair<int, int> >, int> record;  
 pair<int, pair<int, int> > point2pair(pointInt p) {  
 return make\_pair(p.x, make\_pair(p.y, p.z));  
 }  
 void \_\_init\_\_() {  
 arr.clear();  
 dist.clear();  
 pre.clear();  
 record.clear();  
 paths.clear();  
 while (Q.size()) Q.pop();  
 }

A\*算法每次从队列中找出f(n)值最小的顶点，将其作为当前顶点，去遍历其周围的顶点，由于顶点到顶点之间的边权是欧几里得距离，故可以应用Dijkstra算法的松弛操作，遍历顶点时计算出该点到已经计算过的点（即arr链表存储的点）之间不经过障碍物的距离，即将原来的拟合算法与寻路算法合并在了一起，代码如代码[[code:astar\_main]](#code:astar_main)所示，\_\_build\_\_是寻路并拟合的函数，\_\_process\_\_是处理最短路径并输出的函数，\_\_main\_\_是算法入口。

bool \_\_build\_\_() {  
 cerr << "A\*算法搜索可达路径中。。。" << endl;  
 arr.push\_back(st); Q.push({0, get\_dist(st, ed), 0});  
 pre.push\_back(-1); dist.push\_back(0);  
 record[point2pair(st)] = 0;  
 while (!Q.empty()) {  
 heap\_node top = Q.top(); Q.pop();  
 pointInt now = arr[top.id];  
 //cerr << top.h << " " << top.g << ": " << now << endl;  
 if (now == ed) {  
 min\_dist = top.h;  
 return true;  
 }  
 for (int i = 0; i < 6; ++i) {   
 pointInt nx(now.x + dir[i][0], now.y + dir[i][1], now.z + dir[i][2]);  
 //check\_is\_out\_of\_bound  
 if (nx.x < 0 || nx.y < 0 || nx.z < 0) continue;  
 if (nx.x > range\_x + 1 || nx.y > range\_y + 1 || nx.z > range\_z + 1) continue;  
 //check\_is\_able\_to\_reach  
 if (check\_all\_insect(now, nx) != -1) continue;  
 if (record.count(point2pair(nx)) == 0) {  
 arr.push\_back(nx);  
 int id = arr.size() - 1;  
 pre.push\_back(top.id);  
 dist.push\_back(dist[top.id] + get\_dist(now, nx));  
 record[point2pair(nx)] = id;  
 for (int pre\_id = 0; pre\_id < id; ++pre\_id) {  
 if (check\_all\_insect(arr[pre\_id], nx) == -1) if (dist[pre\_id] + get\_dist(arr[pre\_id], nx) < dist[id]) {  
 dist[id] = dist[pre\_id] + get\_dist(arr[pre\_id], nx);  
 pre[id] = pre\_id;  
 }  
 }  
 Q.push({dist[id], get\_dist(nx, ed), id});  
 }  
 else {  
 int id = record[point2pair(nx)];  
 bool isup = false;  
 for (int preid = 0; preid < arr.size(); ++preid) if (preid != id && check\_all\_insect(arr[preid], arr[id]) == -1) {  
 if (dist[preid] + get\_dist(arr[preid], arr[id]) < dist[id]) {  
 dist[id] = dist[preid] + get\_dist(arr[preid], arr[id]);  
 pre[id] = preid;  
 isup = true;  
 }  
 }  
 if (isup)  
 Q.push({dist[id], get\_dist(nx, ed), id});  
 }  
 }  
 }  
 return false;  
 }  
 bool \_\_process\_\_() {  
 int ed\_idx = record[point2pair(ed)];  
 for (int i = ed\_idx; i != -1; i = pre[i]) {  
 paths.push\_back(arr[i]);  
 }  
 reverse(paths.begin(), paths.end());  
 cerr << "找到经过" << paths.size() << "个点的路径";  
 for (int i = 0; i < paths.size(); ++i) {  
 cout << paths[i] << endl;  
 if (!i) cerr << paths[i];  
 else cerr << "->" << paths[i];  
 }  
 cerr << endl;  
 cerr << "该条路径的长度为：" << min\_dist << endl;  
 cerr << "起点到终点的欧式距离为：" << get\_dist(st, ed) << endl;  
 return true;  
 }  
 int \_\_main\_() {  
 \_\_init\_\_();  
 bool build\_flg = \_\_build\_\_();  
 if (!build\_flg) {  
 cerr << "未找到从" << st << "到" << ed << "的有效路径！" << endl;  
 return 0;  
 }  
 \_\_process\_\_();  
 return 1;  
 }  
}

由于A\*算法属于启发式算法，对于其复杂度的分析不太准确，但已知的是由于估价函数的存在，可以使路径朝终点方向搜索，有效地减少遍历点的数量，从而在时间和空间复杂度上都有不小的优化。

## 本章总结

本章主要讨论了算法实现当中的重要部分的实现原理，首先分析了常用的最短路径Dijkstra和A\*算法，对其原理和实现方法进行了分析，以及介绍了对障碍物数据进行处理的算法和线段与障碍物相交算法，最后就是讨论关于基于BFS和A\*最短路径算法的实现。 在常见的正权图结构中，Dijkstra算法拥有非常好的时间和空间复杂度，而引入了估价函数的A\*算法，在进行启发式搜索提高搜索算法效率的同时，可以保证找到一条基于评估函数的最优路径。 对于输入的数据，我们需要先对输入的障碍物底面的两中点和z轴范围做一个处理，计算出其的底面四个顶点。在起点、终点和障碍物离散到指定精度的空间后，提出了利用了向量叉积来检查线段是否与障碍物相交的算法。 最后设计和实现了基于BFS和A\*的三维空间最短路径算法，并分析了两者的特点，BFS算法原理较为简单，是基于暴力的系统展开并检查图中的所有顶点；A\*算法采用了估价函数，可以大大提高算法使用的运行时间和减小空间占用，并且能找到一条基于估价函数的最优路径。

# 算法测试

## 本章总结

00

卜月华; 吴建专; 顾国华; 殷翔, 《图论及其应用》 第一版, 东南大学出版社: 1-2, 2007

作者. 书名 [M]. 版次. 出版地：出版单位，出版年份：起止页码.

邓建松等, 《 科技排版指南》, 科学出版社.

吴凌云, 《CTeX FAQ (常见问题集)》, *Version 0.4*, June 21, 2004.

Herbert Voß, Mathmode, <http://www.tex.ac.uk/ctan/info/math/voss/mathmode/Mathmode.pdf>.

感谢你, 感谢他和她, 感谢大家.