多边形障碍物环境下多边形机器人平移最短路径规划问题的求解

2004210911 陈婧 计研二 solachen@gmail.com
2004211030 谢婧 计研三 sherry.xie@263.net
2004211023 陈强 计研四 sciurid@gmail.com

目录

一、简要综述

多边形障碍物环境下多边形机器人平移最短路径规划问题可以这样描述：在一个二维有限的环境中，存在一组多边形的障碍物；一个多边形机器人需要从该环境中的一个位置出发，到达环境中的另外一个位置，要求在运动过程中，机器人只能平移，不能转动，机器人所在的多边形和障碍物所在的多边形必须不能有任何的相交；要求所有可能的运动路线中最短的一条。

该问题是机器人路径规划中的一个抽象。在机器人的工作环境中，经常需要机器人从一

个位置移动到另外一个位置上去。在这个过程中，必须保证机器人不能和其它的设施相撞，而且为了使工作更加有效率，往往需要机器人选取最短的一条路径。

实际中的机器人是三维的，很多情况下我们只关心它在水平的二维空间中的运动情况。另外，实际的机器人和障碍物的水平截面可能不会是多边形的（例如圆形或者椭圆形），很多情况下我们可能采用近似的方法逼近，或者膨胀机器人和障碍物，使得他们能有一个安全距离，不至于由于距离太近而相撞。

在本次实验中，我们仅考虑在理论上的求解。也就是说，机器人和障碍物都是多边形的，而且在一个环境包围框里面，不考虑必要的安全距离等问题。

二、算法原理

1. 工作空间、禁止空间和自由空间

一个二维的机器人所在的空间，称为工作空间。我们采用和机器人关联的一个点的位置作为机器人的位置表示，称为参考点；使用机器人的各个定点相对于参考点的坐标表示机器人。直观上讲，相当于我们选定参考点作为原点建立了一个平面直角坐标系，机器人就是用各个顶点的坐标来表示的。

在不考虑旋转，机器人仅仅进行平移运动的情况下，机器人具有两个维度的自由度。我们可以直接使用参考点的坐标（两个参数）来表示机器人的位置[1]。这两个参数所组成的参数空间，称为机器人的C-空间（Configuration Space）[2]。

在 C-空间中，并不是所有的位置都是机器人可以到达的。如果在某个点上，机器人会和障碍物相交，那么这个位置就是不允许的。所有这些不允许的位置的集合，被称为禁止空间；而 C-空间中其余的部分，则称为自由空间。

机器人运动的任何一条路径，都是 C-空间中的一条曲线。而如果机器人要无碰撞地从起点到达终点，那么，机器人的运动路径所对应的曲线必然完全包含在自由空间中。那么，求解最短路径的问题就转化为求解自由空间以及自由空间中两点之间的最短距离。

同时，每一个障碍物都对应到禁止空间中的某一个点集，代表着机器人和该障碍物相交的位置点。我们称这个点集为该障碍物的 C-空间障碍物，简称 C-障碍物。

由上述可知，要计算最短路径，首先要求取自由空间。而自由空间是禁止空间的子集，禁止空间又可以通过计算 C-障碍物来获得，那么，首先我们需要计算出 C-障碍物。对每一个障碍物而言，其对应的 C-障碍物可以通过和机器人的变形进行集合运算获得。这个集合运算就是 Minkowski 和。

2. Minkowski 和

对于任何两个集合 $S_1 \in R^2$ 和 $S_2 \in R^2$，它们的 Minkowski 和（记作 $S_1 \oplus S_2$）定义为：

[1]如果机器人是可以旋转的，那么，除了参考点的位置以外，还要定义一个和参考方向的夹角，范围在[0,2π)，用来表示当前机器人的方向。

[2]如果机器人是可以旋转的，那么，C-空间就由参考点的两个参数和参考方向一个参数共同组成，确切的说，是$R^2 \times [0,2\pi)$。这个空间并不是一个欧氏空间。

$$S_1 \oplus S_2 \overset{def}{=} \{p+q \mid p \in S_1, q \in S_2\}$$

其中， $p+q$ 代表两个向量的向量和。同时我们定义：对于任何向量 $p=(p_x, p_y)$，$-p=(-p_x,-p_y)$；对于任何集合 S，定义 $-S=\{-p \mid p \in S\}$。

我们可以得到：

两个多边形的 Minkowski 和也是一个多边形。以及：

设 R 为沿平面运动的一个机器人， P 为任一障碍物。则 P 对应的 C-障碍物为 $P \oplus (-R)$。

证明直接使用禁止空间和Minkowski和的定义，参见参考文献[1]，第 378 页。

两个凸多边形的 Minkowski 和可以采用下面的算法，两个非凸多边形的 Minkowski 和可以利用三角剖分转化为凸多边形的 Minkowski 和的并，在后面的算法实现中叙述。

```
Poly MinkowskiSum(Poly p1, Poly p2){
//p1, p2 均为凸多边形，逆时针表示
//输出二者的 Minkowski 和，也是凸多边形，逆时针表示
    n, m  分别为 p1, p2 的顶点个数
    i, j  分别为 p1, p2 的 y 坐标最小的顶点的索引值    //若 y 坐标相等，取 x 坐标小的
    endi=i, endj=j       //当 i, j 再次全部回到初始值，循环结束
    Poly sum       //output,初始为空
    do {
        vi, vi+1 分别为 p1 的第 i 个点和第(i+1)%n 个点
        wj, wj+1 分别为 p2 的第 j 个点和第(j+1)%n 个点
        将 vi+wj 加入 sum
        av, aw 分别为向量(vi,vi+1)，(wj,wj+1)与 x 轴正方向的夹角（逆时针）     //
转过一圈之后应在 2pi 的基础上叠加
        if (av<aw) {
            i=(i+1)%n;
        } else if (av>aw) {
            j=(j+1)%m;
        } else {
            i=(i+1)%n;
            j=(j+1)%m;
        }
    } while((i!=endi)||(j!=endj));
    return sum
}
```

通过计算 Minkowski 和，我们就可以获得禁止 C-空间。然后使用包围盒（同样也是经过 Minkowski 和处理过的，比实际的包围盒要小）所包围的点集合禁止 C-空间求差，就可以获得自由空间。

3. 最短路径的特点和可见性图

经过自由空间的计算以后，下一步工作就是计算空间中两点之间的最短路径了。首先，由于自由空间可能是不连通的，如果起点和终点位于自由空间的两个互不连通的子集中，那么最短路径是必然不可能存在的。这种情况下可以直接报错，不在进行后续处理。具体的细节我们将在算法实现中给出。下面我们只处理起点和终点位于同一个连通子集的情况。

在机器人和障碍物都是多边形的情况下，自由空间是一个包含有多边形洞的多边形。在这样的环境中，有一个结论：

穿行于一组互不相交的多边形障碍物之间，经过起点到终点的任何一条最短路径，都是一条多边形路径，其中所有的内部顶点都是多边形障碍物的顶点。

这个结论可以由“全局最短路径必然是局部最短路径”的原理来证明，详细证明参见参考文献[1]，第 420 页。

在这个结论的基础上，我们可以得到，全局最短路径是由直线段的边组成的，这些直线段的起点和终点都是自由空间的顶点或者洞的顶点。而且，只有当两个顶点之间的连线完全位于自由空间中时，这两个顶点才可能成为最短路径上的相邻顶点。

当确定了最短路径的这种特性之后，我们就可以通过构造一个称为可见性图（Visible Graph）的路线图，并借助它找到最短路径。可见性图（V，E），其中 V 是自由空间中的各个顶点（包括洞的顶点）的集合加上起点和终点。如果连接两个顶点的线段不和任何 C-障碍物相交，即完全包含在自由空间内，我们就称这两个顶点是互相可见的，则连接它们的线段被称为可见边。E 就是所有可见边的集合。

为了构造可见性图，我们需要对每一对边，都要进行计算，判断是否相交。构造可见性图的复杂度为$O(n^3)$。

4. Dijkstra 算法

在获得可见性图的基础上，可以使用图论中著名的 Dijkstra 算法来获得起点和终点之间的最短路径。

在计算可见性图和使用 Dijkstra 算法的时候，实际上有两种策略来构造可见性图。

第一种策略是先将所有顶点之间的可见性计算出来，然后再运行 Dijkstra 算法。对这种策略略加改进，将起点和终点和其他顶点之间的可见性单独计算，而保留顶点之间的计算结果的话，那么，在一个固定的环境中，所有障碍物是固定不变的，那么顶点之间的可见性也不会发生变化，这样在起点和终点发生变化的时候，可以只计算起点和终点和其他顶点之间的可见关系，可以大大减少计算量。

第二种策略是只有在需要的时候才计算顶点的可见性。也就是说，只有某个点是当前距离起点（终点）距离最近的一个点的时候，才计算它和其他点的可见性并更新各点距离。这种策略的时间消耗取决于路径的长度，如果路径本身就跨过整个环境或者大部分环境的话，这样的策略并不能获得任何性能的优势，但是如果路径本身局限于环境的一个很小部分的话，那么实际上需要计算的顶点数量就会非常的少，可以在很短的时间内计算获得。

5. 整体算法流程

A. 通过输入输出接口，获得机器人多边形和障碍物多边形的数据；

B. 计算机器人多边形和各个障碍物多边形的 Minkowski 和；
C. 将所有的 Minkowski 和取并，得到禁止空间；
D. 计算外包围盒对应的禁止空间；
E. 检查机器人的起点和终点位置，如果起点和终点都位于禁止空间内，则路径不可能存在，返回错误信息；
F. 根据禁止空间计算自由空间，得到用多边形表示的自由空间；
G. 分离自由空间中的各个连通集；
H. 检查机器人的起点和终点未知，如果起点和终点位于不同的连通集内，则路径不可能存在，返回错误信息；
I. （策略 1）计算起点和终点所在的自由空间连通子集内的可见性图；
J. 使用 Dijkstra 算法求解起点和终点之间的最短路径。
K. （策略 2）使用扩展的 Dijkstra 算法求解起点和终点之间的最短路径。

三、系统设计

系统主要分为以下几个部分：

1. 基础运算类。这部分主要是定义一些最基本的数据类型、运算和结构。在系统中，该部分包含 CGPoint 点类，CGVector 向量类，CGSimplePolygon 简单多边形类和 CGComplexPolygon 复杂多边形类。
2. 多边形运算类。提供了多边形的存储、表示，多边形的运算、以及其他必要的操作。
3. 输入输出类。这部分类包含了对用户界面的响应，数据的存储等输入输出操作。
4. 用户界面。用户界面使用 MFC 的 SDI 框架，使用 OpenGL 进行图形显示。OpenGL 使用 Windows 提供的程序库，并采用了双缓冲的机制消除闪烁。

四、数据结构及算法实现

程序中使用的主要数据结构和算法实现描述如下。

1. 浮点运算、点和向量

点和向量均为 2 维表示，分别用 CGPoint 和 CGVector 类来表示，除了必要的构造、拷贝、赋值等操作以外，还提供了向量的加减乘除、内积等操作。此外，为了方便使用，还提供了计算距离、计算点和线、线和线之间几何位置关系等等函数。

浮点运算主要是比较运算，在比较运算中引入了容错程度（Tolerance）的概念，当两个浮点数差的绝对值小于制定的 TOL 值得时候，就认为两个浮点数是相等的，避免由于浮点数计算误差而在退化情况下出现问题。

其中，计算点和线之间的集合关系，主要指点和有向线段之间的关系。如果点在有向线段的左侧或者上面（逆时针方向不超过 180 度），函数返回 1；点在右侧或者下面（顺时针方向不超过 180 度），函数返回-1；如果点在有向线段所在直线上，则返回 0。

线和线之间的关系，分为真相交、不相交、相切等情况，分别返回 1，-1，0。

2. 简单多边形（CGSimplePolygon）

简单多边形指的是只有一条连续边界，内部没有洞的多边形。简单多边形使用节点存储的方式，按照顺序将节点依次存储在一个链表（数组）里面。

简单多边形提供了输入的接口，AppendVertex 和 FinishAppend，分别向当前的节点列表末尾增加节点和检查当前的节点列表是否可以成为一个简单多边形。为了减少后续计算中出现的特殊情况，简单多边形不允许自相交情况的出现。AppendVertex 和 FinishAppend 中采取了必要的检查和控制。

```
AppendVertex
多边形的列表为 vertices，当前点为 p；
检查线段(p,vertices.last())和(vertices.at(i), vertices.at(i+1))的关系， i 从 0 到 l-2
如果都不相交，则将 p 加入 vertices，返回 true
否则返回 false，p 不加入 vertices
```

执行 AppendVertex 之后，并不能保证当前的节点列表就一定是可以构成简单多边形的。例如下图中的情况：

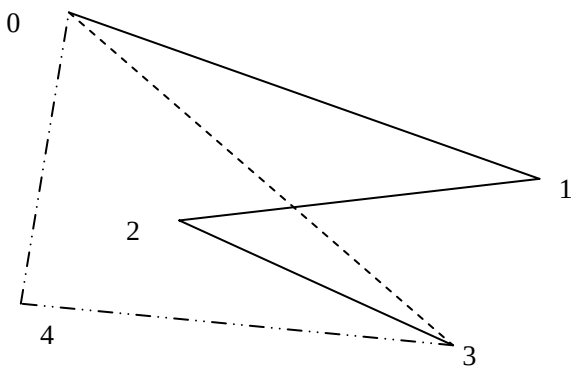


图 1 节点添加过程中的检测

节点 3 可以使用 AppendVertex 加入节点列表中，但是当前的节点列表仍然是非法的。但是如果因此不允许 3 加入列表也是不合理的，例如当后续的点是节点 4 的话，这 5 个节点完全可以成为一个合法的多边形。因此，在添加完所有的节点后，必须使用 FinishAppend 来判断列表所代表的多边形是否合法。FinishAppend 将最后一个节点和第一个节点的连线依次和列表中的各个边进行测试，如果发现自相交，那么节点列表将根据参数选择被清空，并返回失败，如果均没有自相交，则返回正确，简单多边形构造完成。

此外，简单多边形还提供了很多反映多边形属性的函数。

函数 IsClockWise 用来判断多边形的顶点顺序是顺时针还是逆时针的。判断的依据是多边形的外角和的符号。将每一个边都认为是有向的，并且是沿着列表顺序，前一个指向后一个，最后一个指向第一个。由于多边形的外角和必然是 360 度，所以，如果顶点顺序是逆时针的话，各条边之间的角度和必然是 360 度，否则就是-360 度。通过判断外角和的正负性质就可以得知列表的时针顺序。

得知列表的时针顺序以后，同样通过计算外交和就可以得知某个顶点是不是位于一个凸角上：如果该点处的外角和整个多边形的外角和同号，那么就是一个凸角，否则就是一个凹角。如果多边形的各个顶点都是凸角的话，那么该多边形就是一个凸多边形。函数 IsConvex 实现了这个计算过程。

检测点是否在简单多边形内是一个非常重要的操作。程序中使用从测试点向 x 轴正方向发出的射线和多边形的边相交的点数来确定测试点是否在多边形内部的：奇数次说明测试点在多边形内部，偶数次说明测试点不在多边形内部。测试算法并没有使用测试线段相交的方法。因为可能会出现顶点恰好和测试点位于同一水平线上的情况。算法根据线段的两个端点位于测试点 x 轴正方向的位置和测试点相对于线段两个端点的位置共同判断测试点和线段的相交情况。为了处理前面所述的退化情况，我们将线段端点和测试点位于同一水平线上的情况视为在 x 轴上方。这种处理方式，利用了多边形的连续性，在几种退化情况下都可以工作的很好。图 2 中 A、B、C、D、E、F 五种情况对相交点的贡献分别为 1、0、0、1、0、1，对检测没有产生错误的影响。

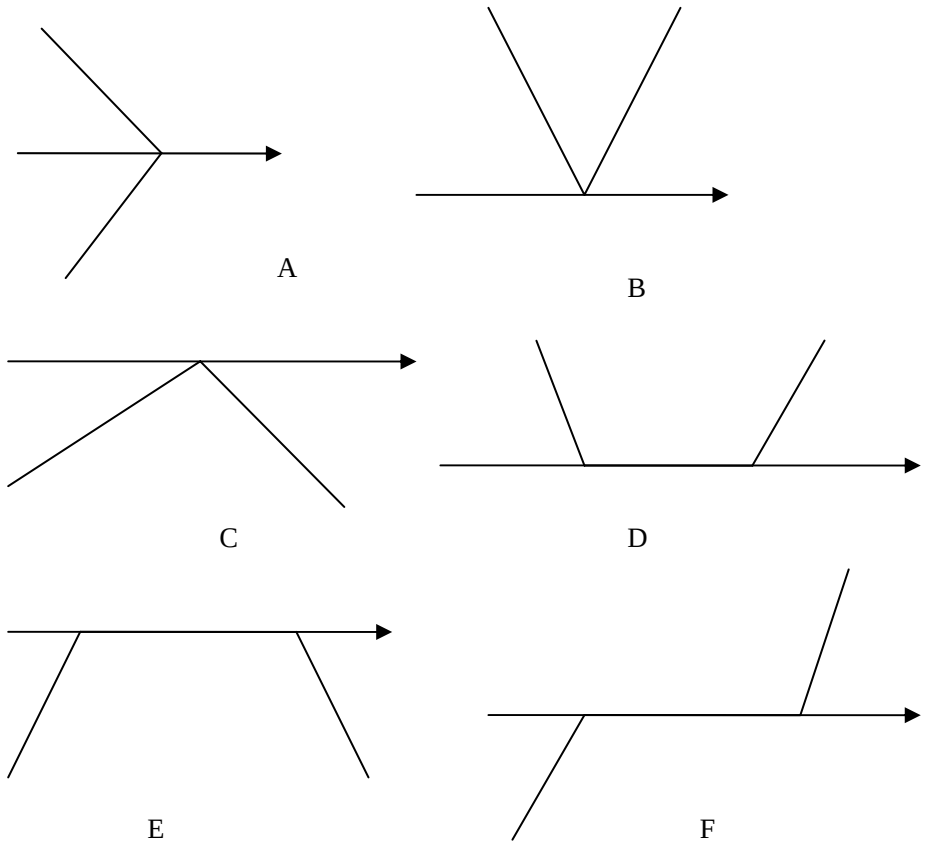


图 2 检测点是否在多边形内的几种退化情况

CGSimplePolygon 类中提供的另外一个重要功能是判断顶点之间的可见关系。该功能主要应用在可见性图的计算中。同一个多边形上的顶点不能通过判断是否和多边形的边相交来判断，需要进行特殊处理。

程序中采用的算法如下：

IsVisible

输入：简单多边形 Polygon，两个点的索引 i 和 j

输出：两个点是否可见（连线不穿过多边形）

如果 i、j 两点相邻，则返回 true；

如果 i、j 两点不相邻，分别沿着顶点列表，按照从 i 到 j 和从 j 到 i 的顺序，构造两个包含有边(i,j)的简单多边形；

如果其中一个多边形出现了自相交，不能构成多边形，则返回 false；
如果两个多边形的时针顺序一致，则返回 false，否则，返回 true。

i、j 不相邻情况下的几种状态如图 3 所示。

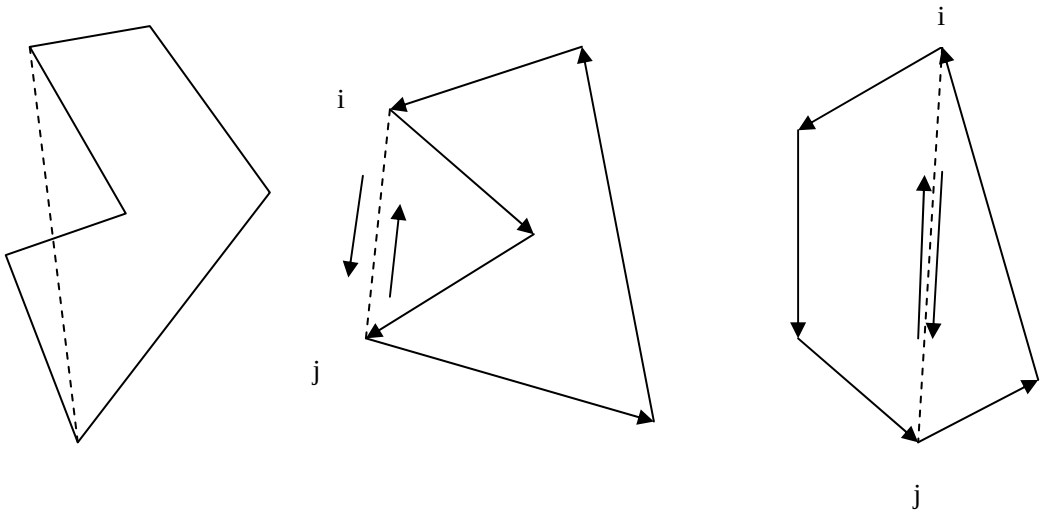


图 3 同一多边形上两点之间的可见性判断

此外，CGSimplePolygon 类还提供了旋转、平移、缩放等简单操作。

3. 复杂多边形（CGComplexPolygon）

复杂多边形是指包含有洞的多边形。复杂多边形未必只有一个连通集，可能会分成多个连通集。如图 4 中的情况都是复杂多边形。

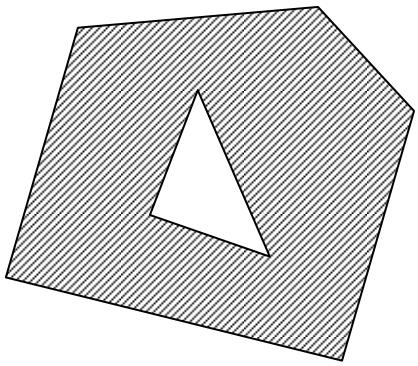


图 4-1

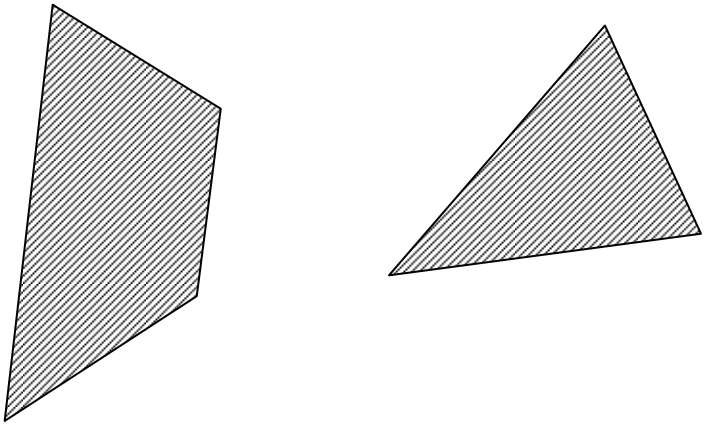


图 4-2

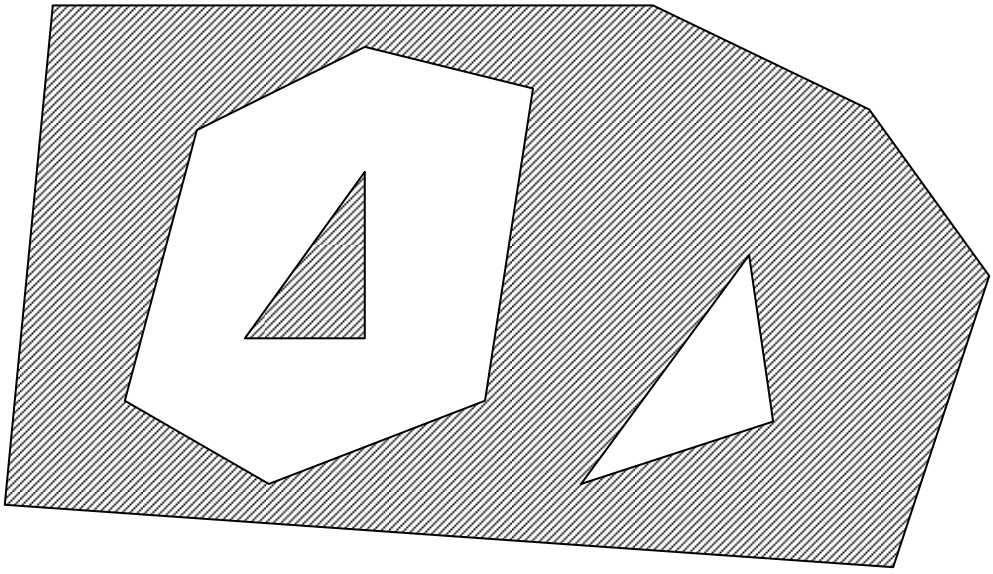


图 4-3

在复杂多边形中，外边界和内边界（洞的边界）分别使用简单多边形来表示。为了区别，外边界使用逆时针顺序的简单多边形，内边界使用顺时针顺序的简单多边形。

对复杂多边形进行各种运算，是后续算法的前提条件。其中，交、并、差、异或和三角剖分的算法使用了网上提供的运算库 GPC（Generic Polygon Clipper）。程序中有若干个类负责进行 GPC 数据结构和系统数据结构之间的转换工作。

一个对复杂多边形重要操作是获得复杂多边形中的不同连通集。我们根据复杂多边形的特点，设计的如下算法来分离复杂多边形中的连通集。

通过观察我们可以得到这样的结论：**每个连通集必然有且仅有一个外边界**，如果定义外边界之间的“小于”关系为：如果一个外边界被另外一个外边界包围，也就是说，它的所有顶点都位于另外一个外边界内部，那么这个外边界就“小于”包围它的那个外边界。在这个定义下，我们会发现**内边界只能和包围它的最小外边界在同一个连通集内**。

根据这一点，算法可以设计如下：

```
GetConnectedComponents
初始化一个复杂多边形的列表 connected 为空
将每一个外边界加入到 connected 中，并按照从小到大的顺序排列。
```

依次取出每一个内边界，遍历 connected 列表，并将此内边界加入第一个包围它的外边界中

返回 connected

实际的计算结果证明，这种方法是正确和有效的。

Minkowski和的计算，核心是两个凸多边形之间的Minkowski和计算。计算的具体算法在参考文献[1]中已经介绍，这里不再赘述。如果多边形的凸凹性质不明确，则需要首先进行三角剖分，将剖分后的结果分别进行Minkowski和，然后将结果求并，就可以得到结果。

算法的描述如下：

MinkowskiSum

输入：多边形 A、B

输出：多边形 C，表示 A 和 B 的 Minkowski 和

如果A、B都是凸多边形，使用参考文献[1]的算法计算Minkowski和并返回。

如果 A 不是凸多边形，

对 A 进行三角剖分，得到 A1、A2、……An，分别递归调用 MinkowskiSum 计算和 B 的 Minkowski 和，将所有的结果求并并返回；

如果 B 不是凸多边形，

对 B 进行三角剖分，得到 B1、B2、……Bn，分别递归调用 MinkowskiSum 计算和 A 的 Minkowski 和，将所有的结果求并并返回。

在系统中，障碍物、禁止空间和自由空间都是使用 CGComplexPolygon 来表示的。

4. 可见性图

可见性图中最核心的部分是两个点之间可见性的判断。

判断两个点之间的可见性，参考文献[1]中给出了一个优化的算法，使用了角度排序和平衡树来减少比较的次数。但是由于该算法中使用了大量的三角运算，在普通的PC上运行效率会比较低，因此没有在程序中采用。程序采用的是Brute-force的列举算法。对两个点之间的可见性采用如下算法进行判断。

IsVisible

输入：两个点 i，j

输出：i 和 j 的可见性

如果 i 和 j 都在自由空间的外边界上

且如果 i 和 j 在外边界代表的多边形上判断可见，那么返回 false

如果 i 和 j 都在自由空间的同一个内边界（洞的边界，或者说是禁止空间的边界）上

且如果 i 和 j 在内边界代表的多边形上判断不可见，那么返回 false

对环境中所有的多边形边界，如果连接 i 和 j 的线段和它们中任意一条相交，那么返回 false，否则返回 true

在实际的运行中会遇到退化情况，因此在可见性判断中，采取了比较严格的标准，即真相交和相切都认为是相交，除非两条线段有着公共的端点，否则就判定为不可见。由于在这些退化情况下，两个点之间的距离等于两个端点和退化点距离之和，因此判定为部可见不会影响到最短路径的检测结果。如图所示：

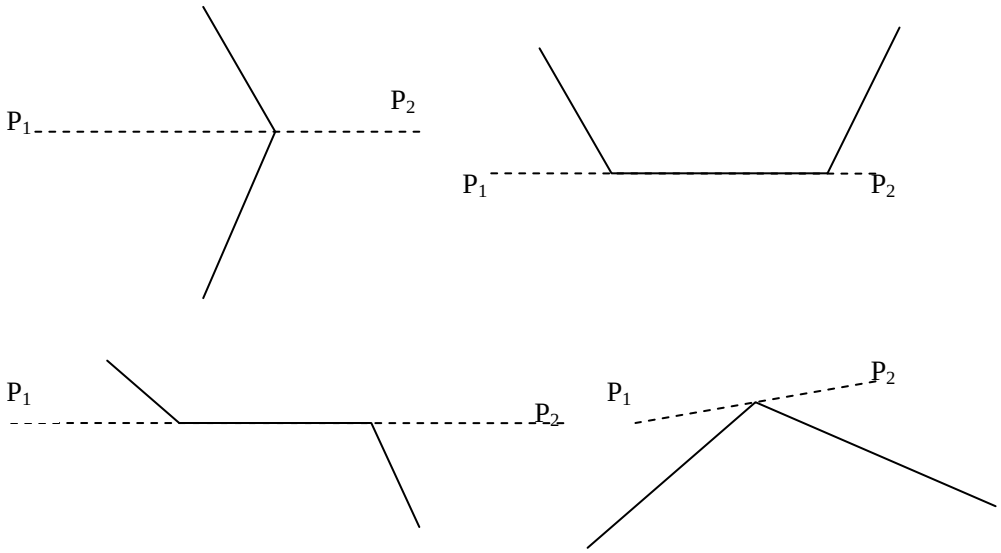


图 5 可见性检测中的退化情况（以上情况将一律按照不可见处理）

5. Dijkstra 算法

Dijkstra

输入：自由空间 FreeSpace、起点 v0，终点 v1

输出：不存在最短路径或者一条以 v0 为起点，v1 为终点的最短路径

数据结构：一个结构的数组

结构中包括：起点到当前点的距离（－1 表示距离无穷大）；当前点的前驱结点；以及标记（0 表示未访问、1 表示访问过、2 表示访问过且找到最短路径的点）

①初始化——设置所有点与起点的距离为无穷大，标记为未访问；

②设置起点的距离为 0，标记为 2；

③将所有与起点有边的点标记为访问过，距离为该边的权，前驱设为起点；

④求所有访问过的点中距离最短的点 g，若 g 不存在，则转⑧；

⑤将点 g 的标记设为 2，若 g 为终点，则转⑧；

⑥遍历所有标记为 1 的点，若它的距离大于 g 的距离＋g 到该点的距离，则修改其距离，并且将其前驱设为 g；

⑦转④；

⑧根据结构数组求最短路径。

扩展的 Dijkstra 算法结合了可见性分析和 Dijkstra 算法，使用的是策略 2 来进行计算。

ExtendedDijkstra

输入：自由空间 FreeSpace、起点 v0，终点 v1

输出：一条最短路径，以 v0 为起点，v1 为终点

设置一个距离向量，代表每个点当前到 v0 的最短路径距离；设置一个路径向量，代表该最短路径的上一个节点。为每个点初始化距离为-1，代表无穷大，将 v0 的距离设为 0，将所有的路径向量设为-1，代表尚未访问到；设置一个可见性表，代表各个点之间的可见关系，将所有的关系初始化为尚未检测；设置一个最小堆，以到 v0 的路径距离为排序准则，

用来存放当前已经访问到但尚未检测的点，将 v0 加入堆中。

取出堆顶的元素作为当前节点，该元素为堆中所有具有最小距离的点。

如果当前节点为 v1，则沿路径向量向前寻找，将路径上的点都加入结果列表，直到找到 v0，将结果列表反向后返回。

遍历所有节点

如果当前节点的距离值小于遍历点的距离值，不做任何操作；

如果当前节点和遍历点的可见关系为可见或者不可见，不做任何操作；

如果当前节点和遍历点的可见关系未知，则计算节点和该点之间的可见关系

如果不可见，则标记可见性表；

如果可见，则标记可见性表并计算当前节点和遍历点之间的距离与当前节点距离值的和为 newdistance

如果遍历点的距离为-1 或者遍历点的距离值大于 newdistance，则遍历点的距离值设为 distance，并将路径向量的值设为当前节点

否则，不进行操作。

五、测试统计、比较结果和理论分析

测试环境：P4M1.8G，640M DDR266，Windows XP Home Edition with Service Pack 2

测试数据选择：

由于程序主要的运行时间消耗在计算 Minkowski 和以及可见性图上，因此我们特别针对这两项计算消耗时间进行了测试。

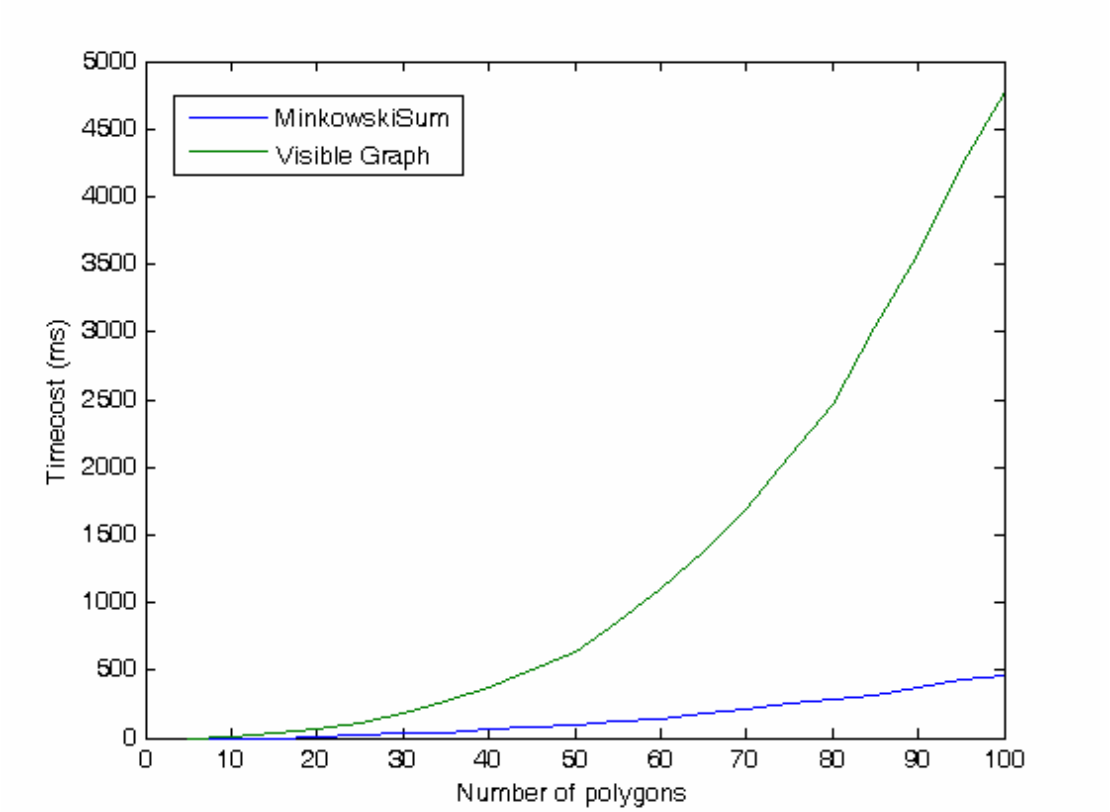
理论上，设机器人的边数为m，障碍物的边数为n的话，如果两个都是凹的话，进行Minkowski和的复杂度将会是$O(m^2n^2)$，并得到mn个点；如果两个中有且仅有一个是凸的话，复杂度将会是O(mn)；如果两个都是凸的话，复杂度是O(m+n)，因此，这一步的操作将会是$O(m^2n^2)$。

由于多边形的形状会影响到计算的结果，不便于分析。为了体现性能的变化，更好地寻找其中的规律，在测试中，机器人和障碍物全部都采用三角形。以障碍物的个数为自变量，以 5 为步长进行测试，由程序记录每一步的执行时间。

这样的话，机器人的复杂度为常数，那么自由空间应该会在$O(n^2\log n)$的时间内构造出来，并且有O(n)个点。这样，使用我们的算法计算可见性图的时间复杂度应该是$O(n^3)$，计算最短路径的复杂度应该是O(nlogn)。

采用上面叙述的配置策略进行测试。测试采用逐步增加多边形的方式，每一种配置计算 5 次求最小值。这里采用的是求最小值而不是求平均值的方式，就是因为可能会出现在某些时刻计算机操作系统的调度出现随机的不平衡，而产生非常慢的结果，造成结果的失常。实际上的测试结果分布很窄，平均值和最小值没有明显差别。

性能测试结果如下：



可以看出，随着三角形个数的增加，Minkowski 和的执行过程比较慢，可见性图的计算时间随着障碍物个数的增加而迅速上升。因为我们采用的算法中，可见性图的计算是立方级别的增长，图中的曲线也验证了这个结论。

六、根据检查的结果进行的改进

根据检查的结果，我们的工作主要集中在了界面的更新和部分退化情况的处理上。本次提交的程序，图形显示部分已经使用 OpenGL，不再使用 Windows GDI 了。图像显示部分性能提高了很多，已经消除了闪烁的问题。

在退化情况上，我们取消了对退化通过情况的支持，即机器人刚好和两个或者两个以上的障碍物相切的情况[3]。在自由空间刚好能满足机器人通过的情况下，现在程序的处理按照实际标准结果，不允许机器人通过。

[3] 从理论上讲，如果机器人的两个位置存在一个最短路径的话，机器人和障碍物必然全部都是开的或者只好有一个是开的。否则，机器人和障碍物之间必然会有一个足够小的空隙，最短路径是理论不存在的。如果机器人和障碍物边界是开得话，当出现机器人恰巧和两个以上的障碍物有接触的位置的时候，机器人是可以通过或者存在在那个位置上的。在工程实现的过程中，Generic Polygon Clipper提供的运算都是基于闭多边形的。因此在通过对包围盒和这样的禁止C-空间求差以后，这样的一个位置会分力成两个独立的顶点，虽然它们的位置一致，但是并没有路径连接它们。
为了解决这个问题，我们曾经在计算 Minkowski 和的时候采取了一点小小的手段来解决这个理论的矛盾。当我们在运算中发现两个 Minkowski 和有退化情况的接触（边和边，边和点，点和点）时候，我们将接触点向障碍物内部退后一个刚好可以被浮点运算检测出来的距离。这样就可以生成一条退化情况下的路径让机器人通过。
但是在实际的工程中，这样做是有问题的。工程中的策略应当更多考虑到实际的需要，机器人本身的行动安全比一条略短的路径要更加重要。因此我们接受了在检查过程中的质疑，并将这部分代码从程序中取消。

七、进一步工作的方向和注意的问题

机器人路径规划问题，更高一个层次的工作是允许机器人旋转的情况下进行路径规划。精确的算法参见[2]。该论文中论述的算法非常复杂，如果要实现的话，需要从最基础的数据结构做起，因为该算法不仅需要计算出机器人在某个关键角度的时刻自由空间的情况，还要记录下此时机器人和障碍物之间的关系，以及这些信息在多边形的运算中的变换等等，需要对数据结构有非常好的理解和实现能力。

机器人路径的规划问题，在理论上并不是一个很困难的问题，但是由于退化情况非常多，使得工程实现并不是非常容易。在多边形的各种操作中都会遇到退化情况和特殊情况，需要小心讨论和分类实现，通过不断的测试发现问题是一个很好的办法。

参考文献

[1] 《计算几何——算法与应用》，Mark de Berg，Marc van Kreveld，Mark Overmars，Otfried Schwarzkopf 著，邓俊辉译。

[2] K. Kedem and M. Sharir. An motion planning algorithm for a convex rigid polygonal object in 2-dimensional polygonal space. Discrete Comput. Geom., 5:43-75, 1990.