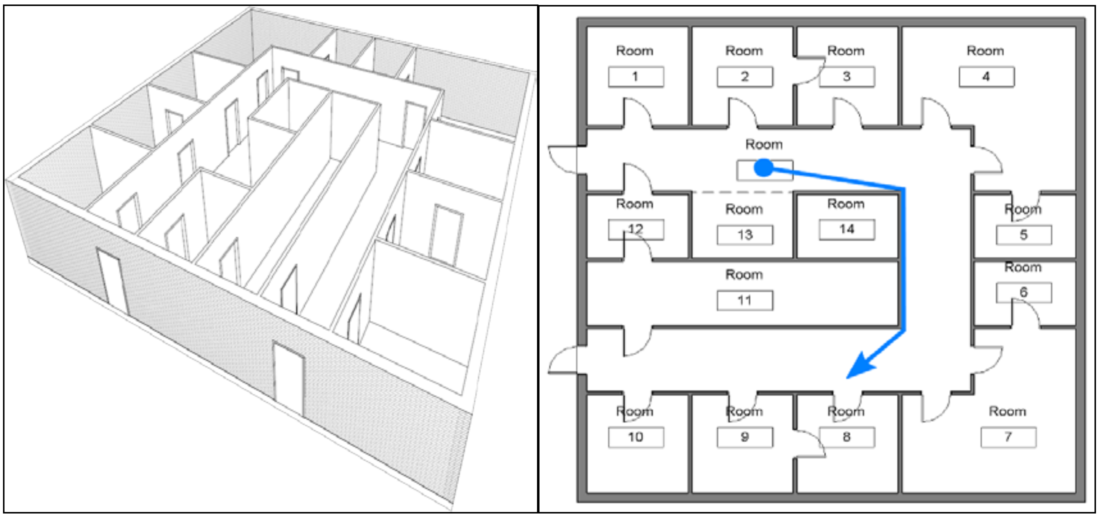
# 平面中多边形障碍下最短路径的求解

## 应用背景

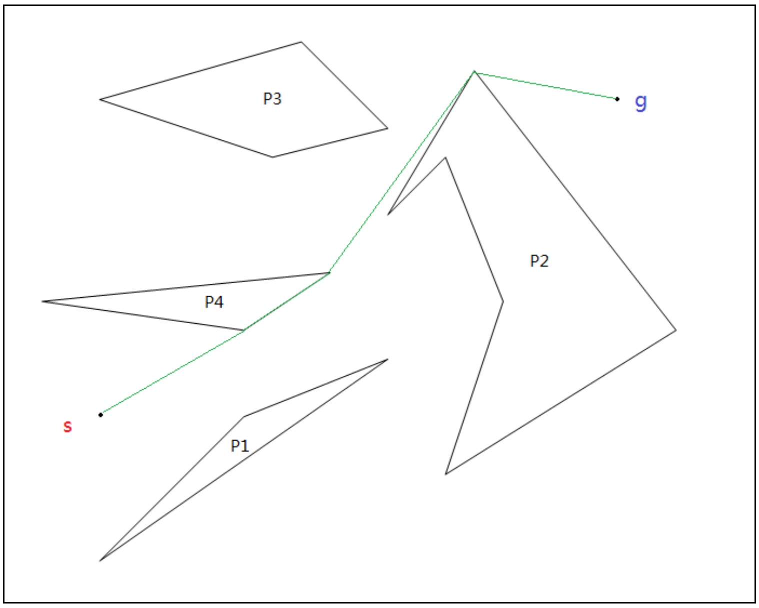
最短路径求解是图论中最常见的问题之一。计算几何中这个问题主要起源于机器人运动规划的问题。在一个平面中，考虑一个机器人从起始位置 s，无碰撞地移动到目标位置 t。假定机器人只有两种运动——直线移动和原地转向，一个常见的目标就是规划一条路径，使得机器人移动距离最短。




左图是一间办公室布局图；右图是其平面图，蓝线显示了 Room13 移动到 Room8 的一条最短路径。[10]

## 问题定义

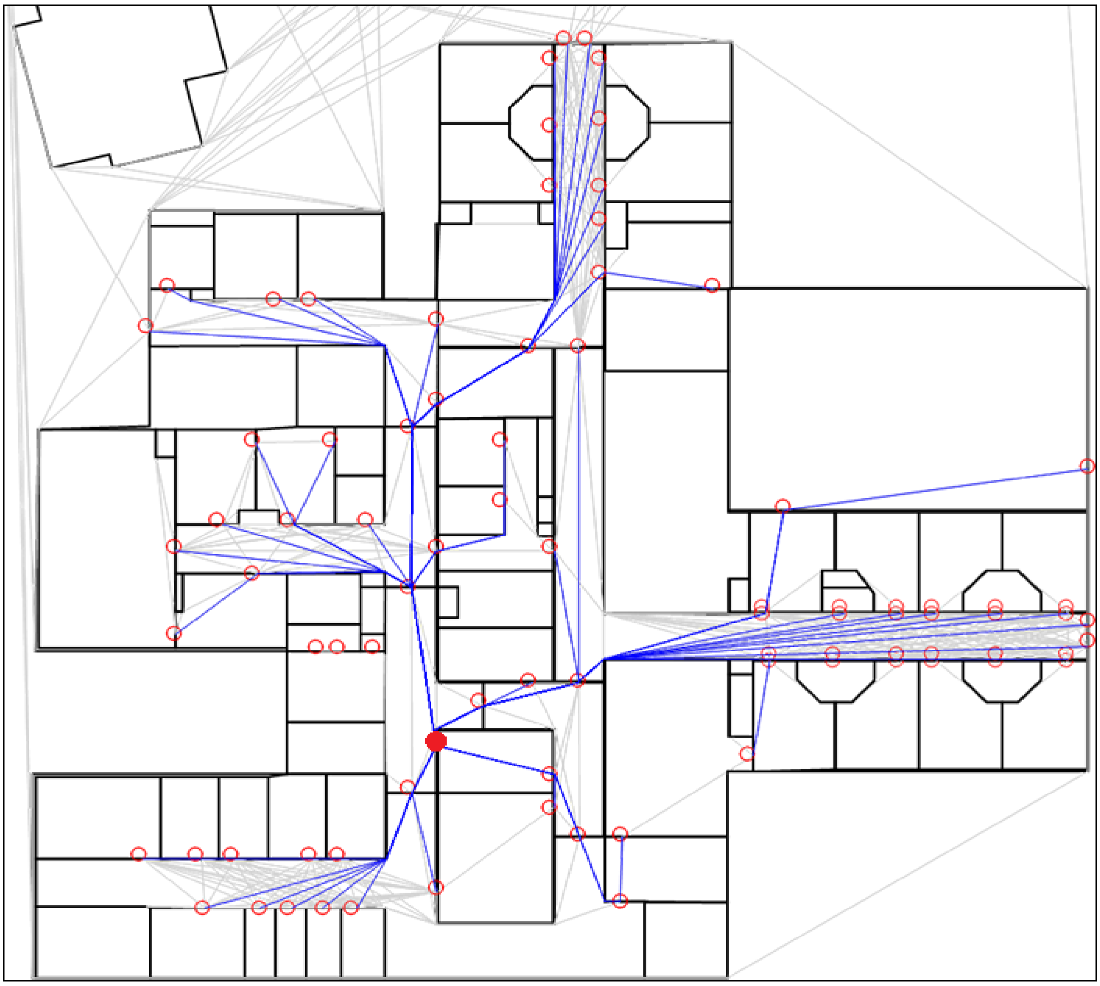
给定平面上一个起始点 s 和一个目标点 g，以及若干多边形障碍物 $P_1$, $P_2$, $P_3$ ... $P_k$，然后在平面上找到一条从 s 到 g 的多边形路径，其距离最短。我们这里限定多边形障碍物都为简单多边形。由于此问题下，非简单多边形可以通过简单的切割合并操作转化为简单多边形，所以我们后面遵循此限定。如下图所示：




该问题的一个常见思路是构造可见性图，然后对可见性图应用最短路径算法。可见性图

构造采用旋转式平面扫描算法[3]，该算法实现上比较简单，复杂度为 $O(N^2\log N)$。在扫描算法上进行改进[4]，可将复杂度降至 O(NlogN+E)。其中 N 是顶点数，E 是可见性图的边数。在生成可见性图后，可直接应用 Dijkstra 最短路径算法，复杂度为 O(NlogN+E)。因此，在这种思路下，算法的最优复杂度即为 O(NlogN+E)。

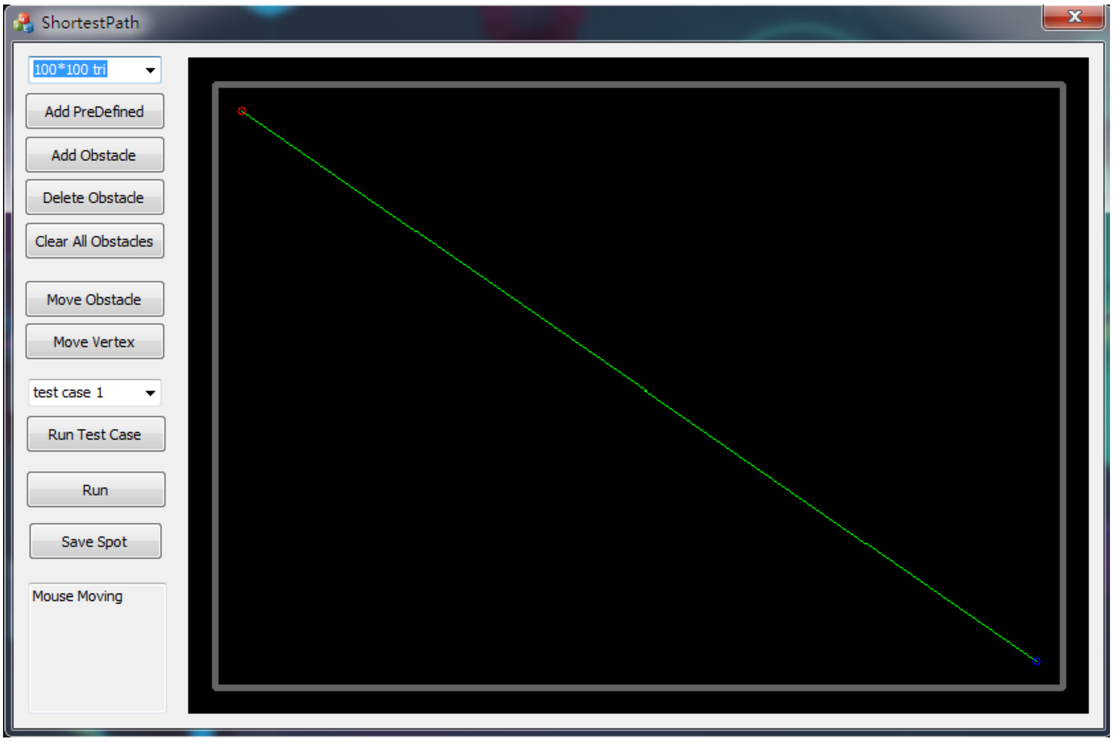
下图所示为该算法的一个图示。从图中可以看到，虽然构造出了可见性图，但是其中的绝大部分线段并不出现在最短路径中，而且在 Dijkstra 算法中也不一定用到。这是一个局限性。



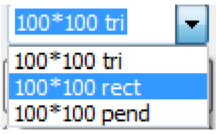
从实心红圈标注的一个门到建筑物内其他所有门的最短路径，用蓝线表示。细灰线表示的是可见性图，用于构造最短路径。[10]

另一种思路是直观形象的波浪线算法，又称为 continuous Dijkstra 算法，由 J.S.B.Mitchell 首先提出[5]，后由 Hershberger 和 Suri 改进[6]，复杂度降至理想的 O(NlogN)，这是目前理论最优的算法。虽然本算法在理解上非常简单形象，但是实现上比较复杂，我们将在最后的部分进行介绍。

# 图形界面




初始界面（左上角的红点为起点，右下角的蓝点为终点）

Add PreDefined：加入预定义的多边形，在下拉菜单

中有三角形、四边形、五边形可选。

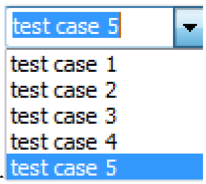
Add Obstacle：加入自定义的多边形，每次左键生成一个顶点，最后右键生成多边形。

Delete Obstacle：删除多边形。左键点击多边形即可删除。

Clear All Obstacles：清除所有多边形。

Move Obstacle：移动多边形。左键可拖动多边形至任意自由区域。

Move Vertex：移动顶点。左键可拖动多边形顶点，任意改变多边形的形状；也可以拖动起点和终点。

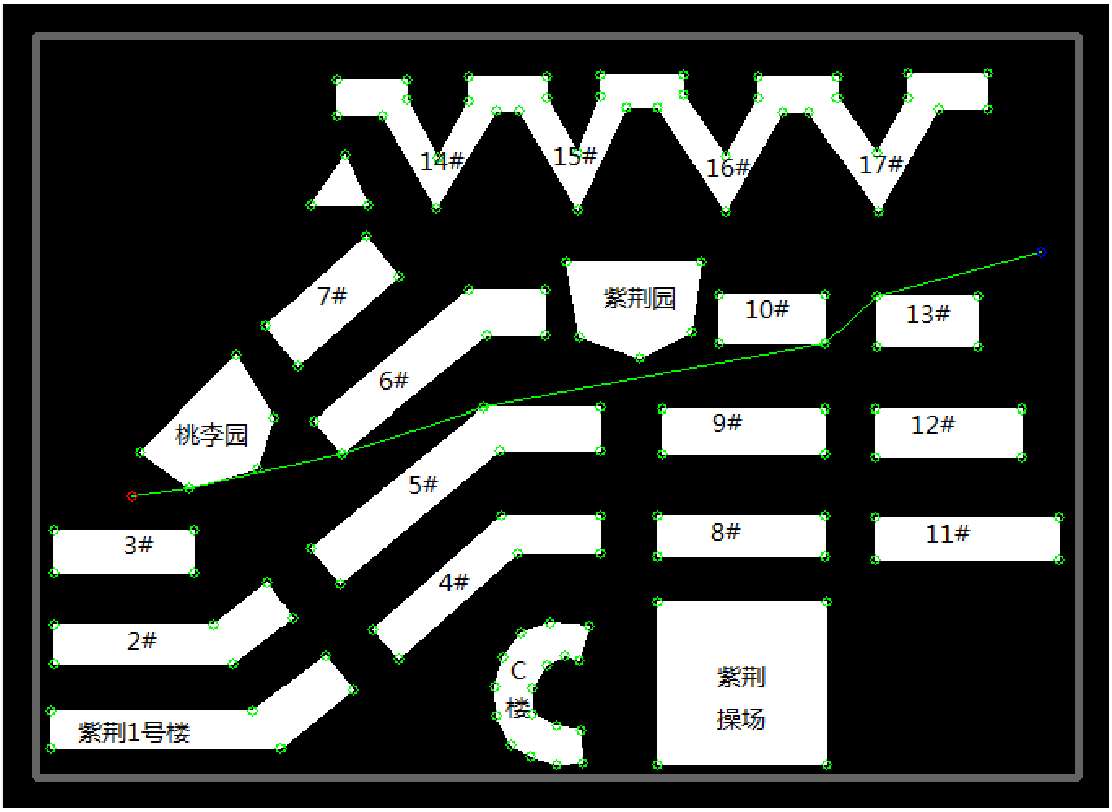
Run Test Case：运行测试样例。在下拉菜单

中有 5 个测试样例，用于系统的各项测试。

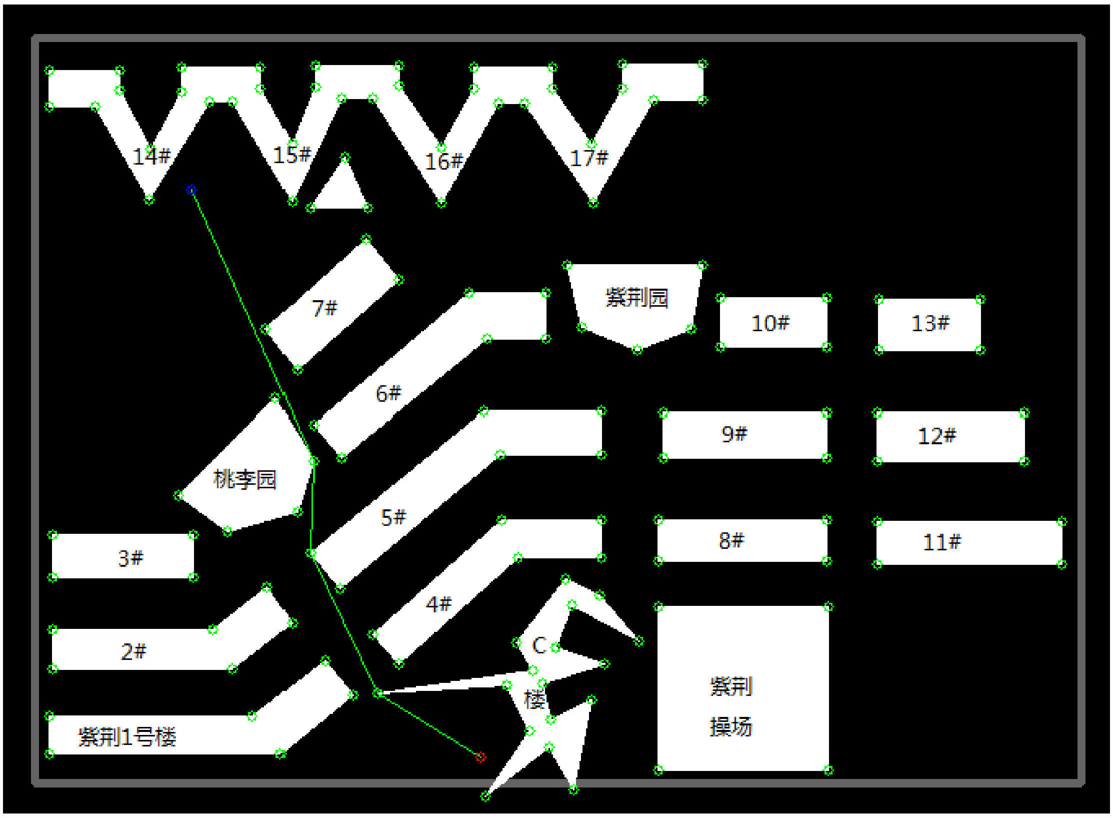
Save Spot：保存。可以保存当前的样例至文件，以供下次重新运行。

我们为紫荆宿舍区画了一个简图，用于实验的简单测试说明：

首先使用画出各栋建筑的形状，其中大部分建筑如紫荆 3 号楼、桃李园等均可使用 Add PreDefined 直接生成，少数如 W 楼、C 楼等则需使用 Add Obstacle 构造；然后使用 Move Obstacle 将各建筑移动至正确位置，还可使用 Move Vertex 对顶点进行移动，以精确描绘建筑物形状。




在给定多边形障碍后，可以使用 Move Vertex 任意拖动起点和终点，系统可以实时响应。也可以拖动和修改多边形，如下图中，使用 Move Obstacle 将 W 楼整体西迁，并使用 Move Vertex 将 C 楼改造成后现代建筑。系统同样是实时给出最短路径。




# 算法实现

构造可见性图的算法框架如下[9]：

**ShortestPath(S, s, g)**

**Input.** A set S of disjoint polygonal obstacles, and two points s, the start position and goal position in the free space.

**Output.** The shortest collision-free path connecting start and goal positions .

1. Gvis ← VISIBILITY_GRAPH(S ∪ {s,g})
2. Assign each arc (v, w) in Gvis a weight, which is the Euclidean length of the segment vw.
3. Use Dijkstra's algorithm to compute a shortest path between the start and final points in Gvis .

**VisibilityGraph(S)**

**Input.** A set S of disjoint polygonal obstacles.

**Output.** The visibility graph Gvis (S).

1. Initialize a graph G = (V, E) where V is the set of all vertices of the polygons in S and E = 0.
2. for all vertices v ∈ V
3. do W ← VisibleVertices(v, S)
4. For every vertex w ∈ W , add the arc (v, w) to E.
5. return G

**VisibleVertices(p, S)**

**Input.** A set S of polygonal obstacles and a point p that does not lie in the interior of any obstacle.

**Output.** The set of all obstacle vertices visible from p.

1. Sort the obstacle vertices according to the clockwise angle that the half line from p to each vertex

makes with the positive x-axis. In case of ties, vertices closer to p should come before vertices farther from p. Let w1 , . . . , wn be the sorted list.

2. Let ρ be the half-line parallel to the positive x-axis starting at p. Find the obstacle edges that are properly intersected by ρ, and store them in a balanced search tree T in the order in which they are intersected by ρ.

3. W ← {}

4. for i ← 1 to n

5. do if Visible(wi ) then Add wi to W.

6. Insert into T the obstacle edges incident to wi that lie on the clockwise side of the half-line from p to wi .

7. Delete from T the obstacle edges incident to wi that lie on the counterclockwise side of the half-line from p to wi.

8. return W

**Visible(wi )**

```
1. if pwi intersects the interior of the obstacle of which wi is a vertex, locally at wi
2.     then return false
3.     else if i = 1 or wi−1 is not on the segment pwi
4.         then Search in T for the edge e in the leftmost leaf.
5.             if e exists and pwi intersects e
6.                 then return false
7.                 else return true
8.     else if wi−1 is not visible
9.         then return false
10.        else Search in T for an edge e that intersects wi−1 wi .
11.            if e exists
12.                then return false
13.                else return true
```

Dijkstra 算法直接使用，不再作说明。

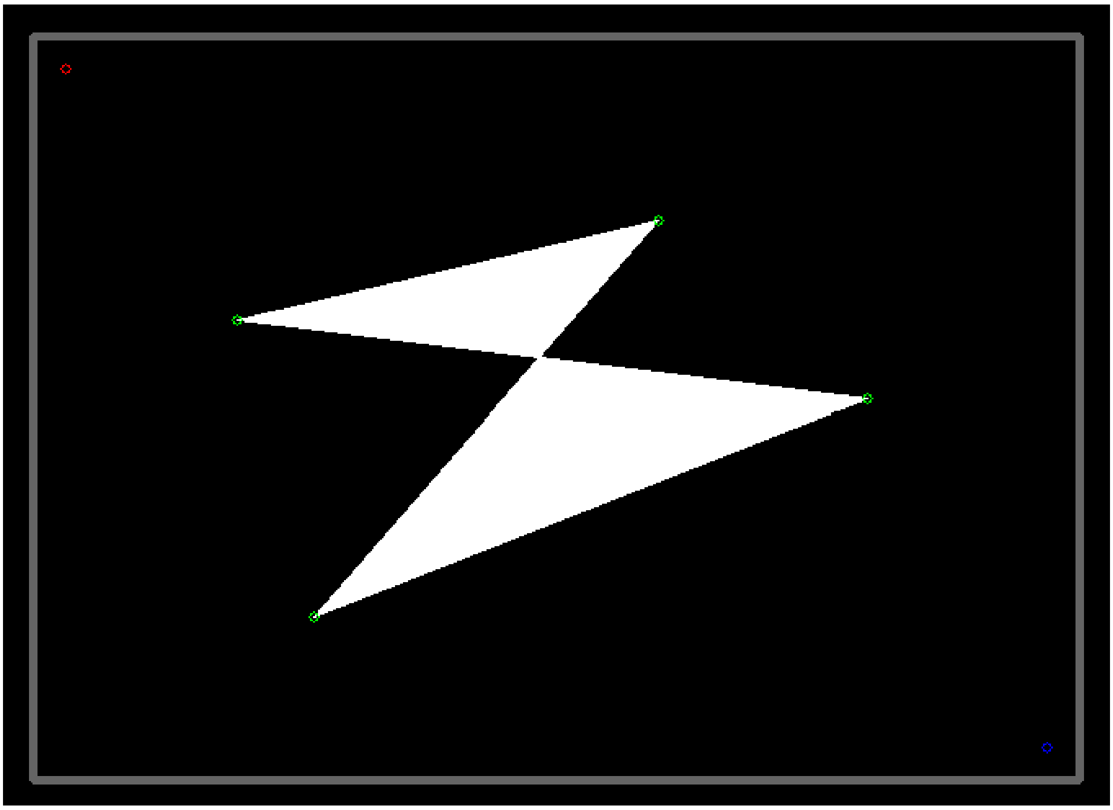
代码实现使用 C++和 MFC，具体代码的细节见附录。

# 实验测试

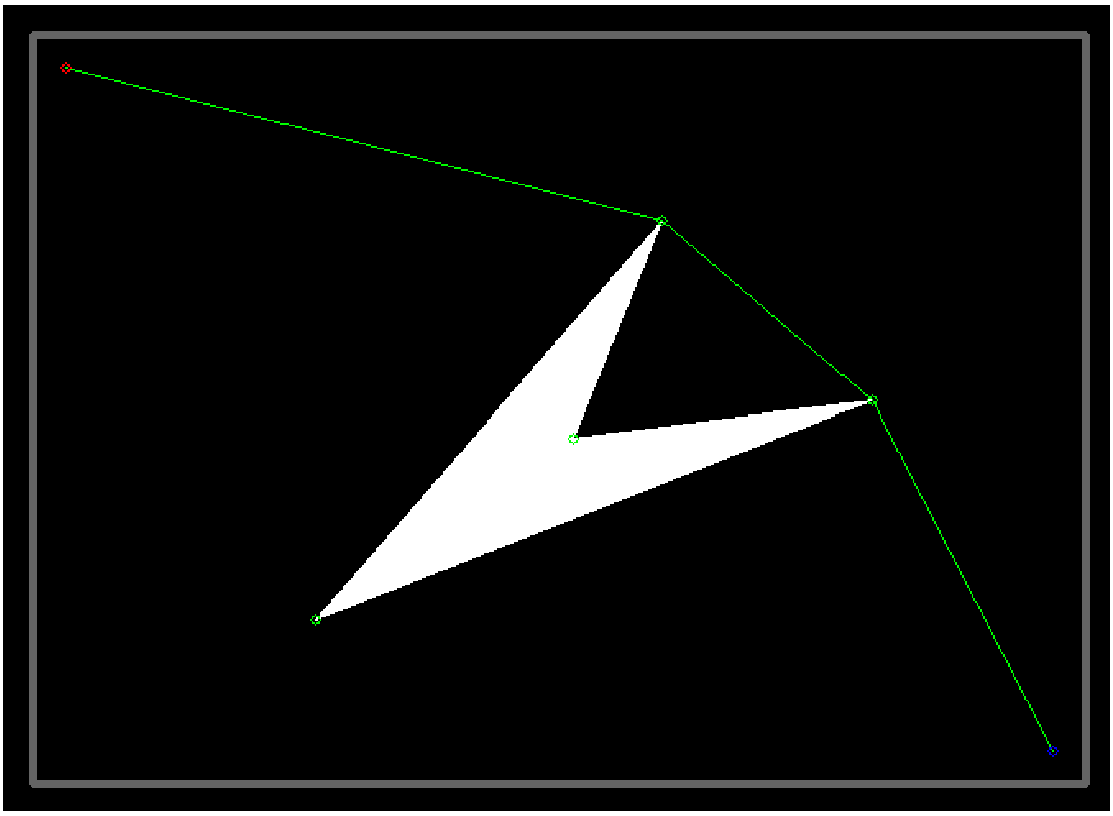
- 非法的输入

  我们在问题定义中限定了**简单多边形且互不相交**的条件。在输入不符合此条件时，系统不输出任何路径。

  下图中，存在非简单多边形以及相交的情况，此时无路径输出。

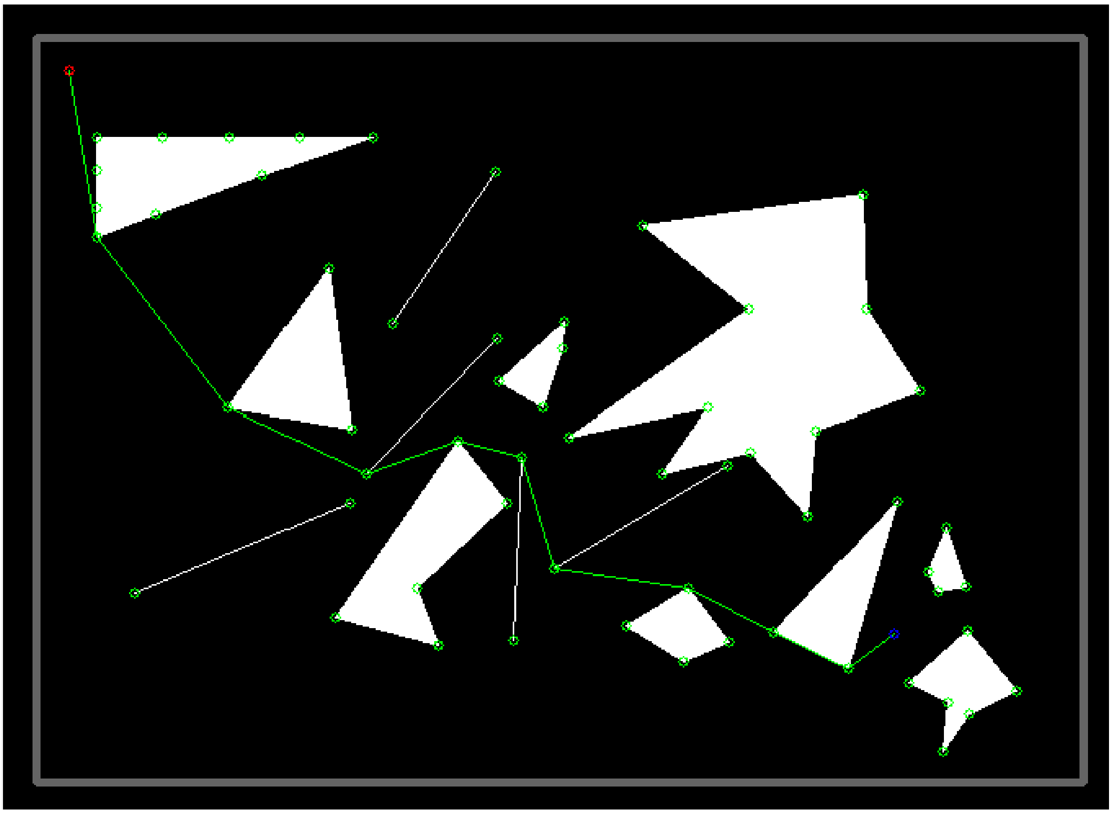


在修改为合法输入后，系统会给出正确的最短路径。



- 边界退化情况

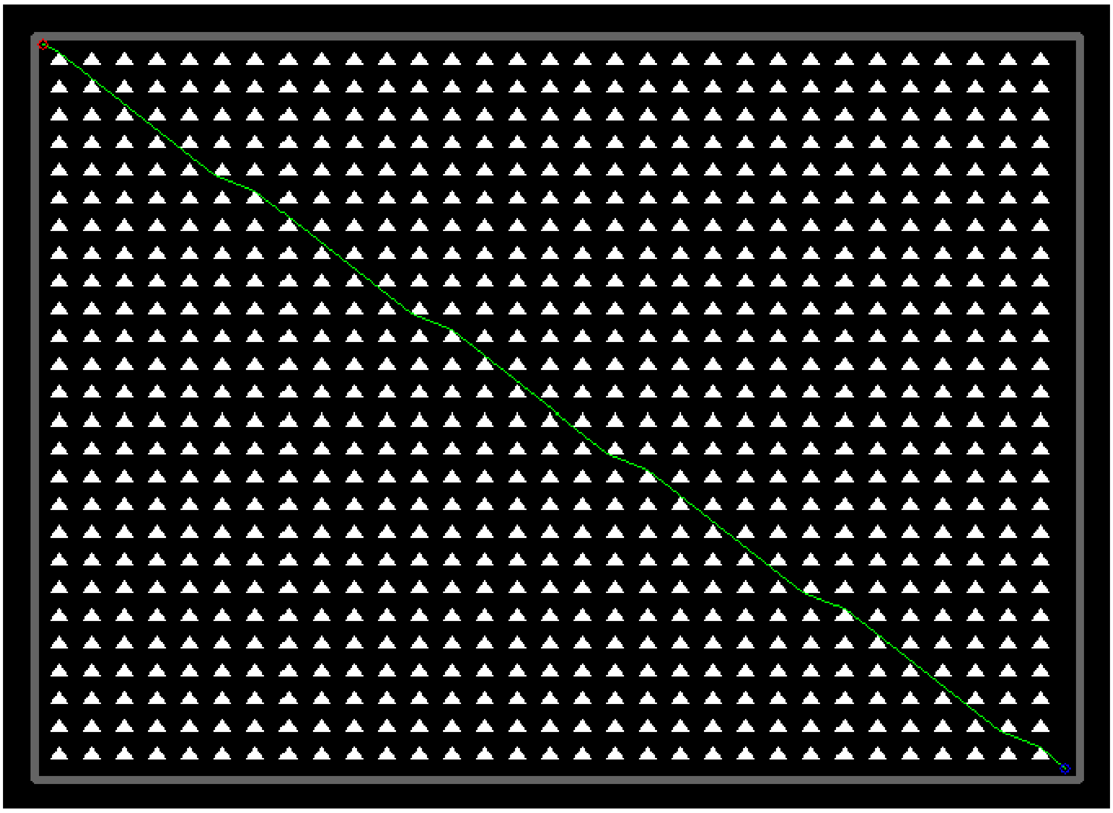
多边形有两种退化的情形：(a)多边形退化为线段(b)多边形的两条边夹角为 180°，退化为一条边。下图为测试样例：



test case 1

- 压力测试

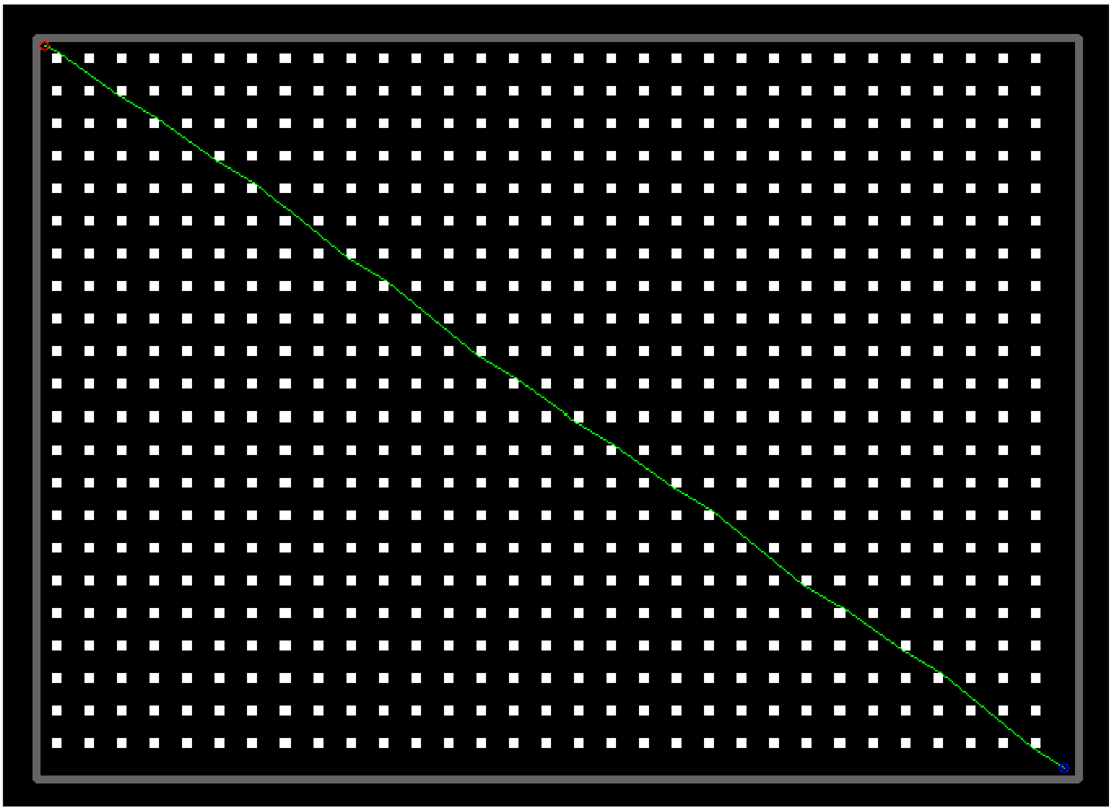
  虽然可见性图构造使用了 $O(N^2\log N)$的算法，但是在实现中，我们使用顶点邻接的两条边对可见的角度进行了限定，从而排除了很大一部分边的相交测试，实际的复杂度并没有理论最坏情况那么差。下图为一个大数据规模的压力测试：



test case 2

此测试样例规模约为 2500 个顶点(800 个三角形),载入后计算可见性图约耗时 5s,之后移动起点、终点,最短路径均可以实时更新。对多边形的更新操作需要更新可见性图,每次耗时 1s 左右。

为了进一步增加测试的压力,我们针对算法,设计了如下的样例,这样,在保持点尽可能多的同时,让点还尽可能的对于其它点可见,所以正方形是一个较好的测试形状。此测试样例规模同为 2500 个顶点(700 个正方形),载入后计算可见性图约耗时 7s,每次更新操作越耗时 4s。而起点、终点的改变仍保持实时更新最短路径。



test case 3

考虑到 10s 为用户忍耐的上限,我们没有给出更大数据规模的压力测试。当多边形障碍物更多时,比如一个城市的平面地图,我们可以让系统进行预处理,计算可见性图,然后再计算最短路径并保存结果,之后用户的查询操作只需要 O(1)的代价,以保证实时的需求。

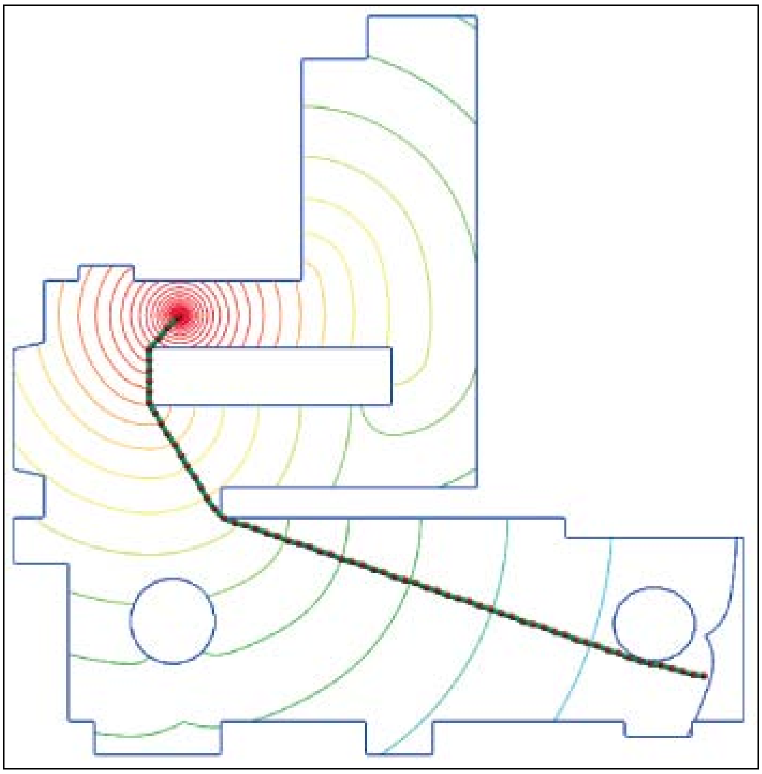
# 思考展望

在问题背景中,还有一个相似的问题:最短链接。即以机器人的原地转向次数最少为目标,来寻找最短路径。这个问题的核心仍然是可见性图的构造,稍微复杂之处在于,需要多次构造可见性图,并对其进行简化处理。

实际上,最短路径和最短链接都有很好的物理演示,可以形象的表现其计算过程。我们各以一组图作简单展示:

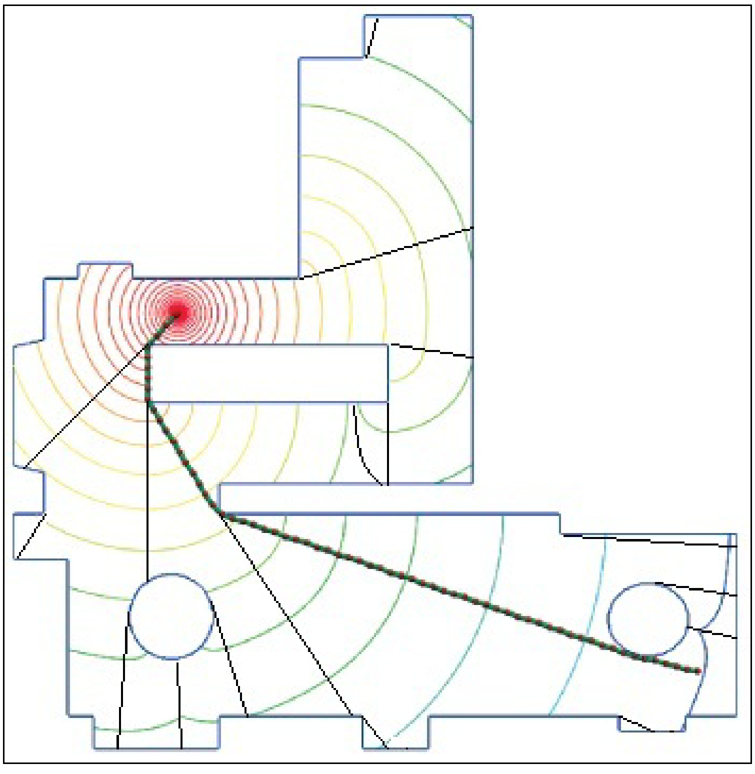
**最短路径**

可以将平面想象为平静的湖面,在起点投入一颗石子,引起的波浪一圈一圈扩散。障碍多边形的边相当于湖岸,波浪在碰到后即消失;两条波浪相碰时也会消失。如图所示,画出了波浪的扩散图。



颜色渐变的弧线就是对应起点的一组等距线[13]

仔细观察可发现，每条波浪都有一个圆心，最初的圆心的起点，后面可能在各个顶点处产生圆心，扩散出新的波浪。下图作了区域划分，每个区域内的波浪对应同一圆心。



每一个区域划分内的等距线是对应同一圆心的圆弧[13]

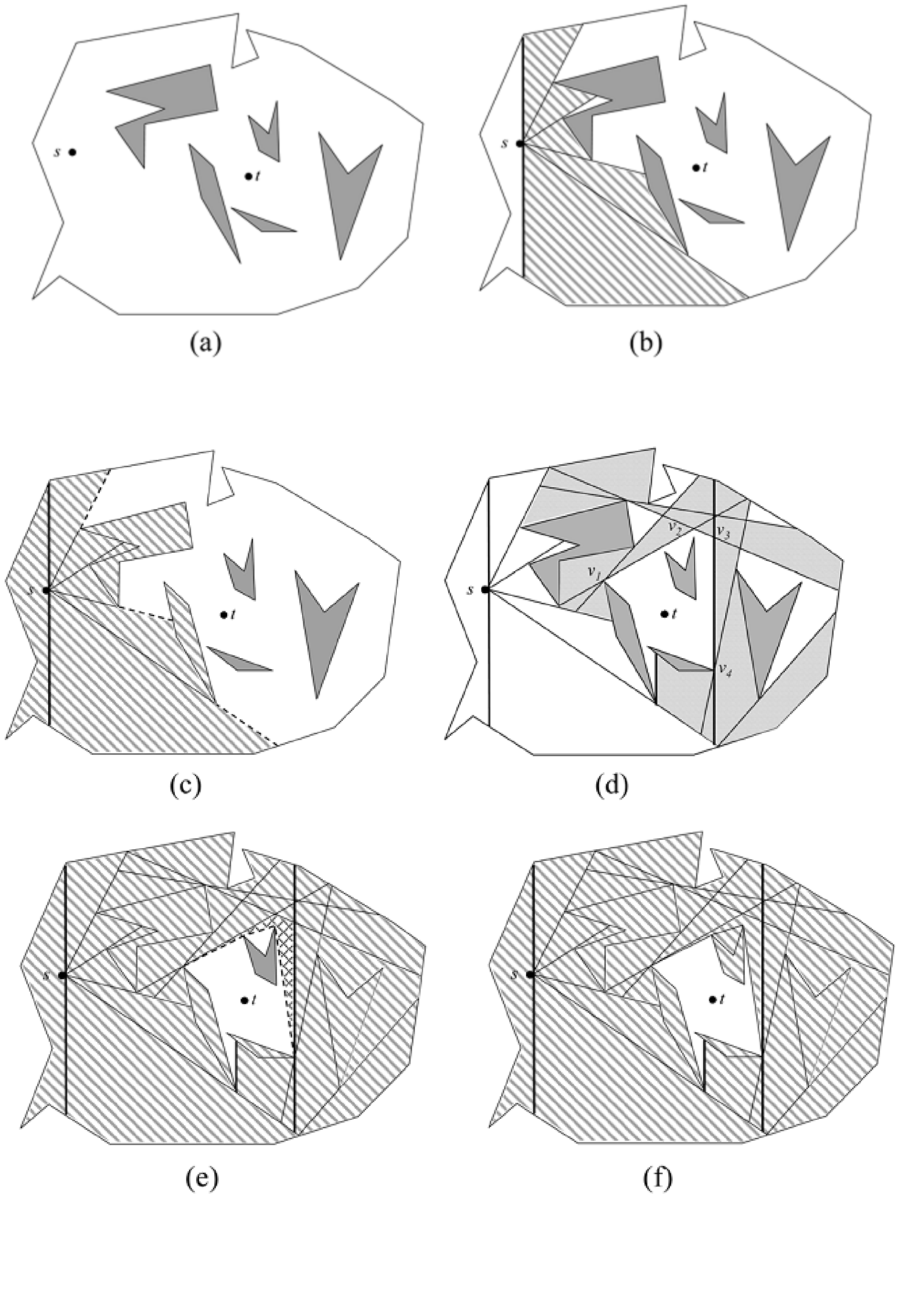
这时的最短路径就非常简单：从终点退回此区域的圆心，即为最短路径的最后一段；再退回上一个圆心，即为倒数第二段……直至退回起点。这些半径的路径即为逆向的最短路径。

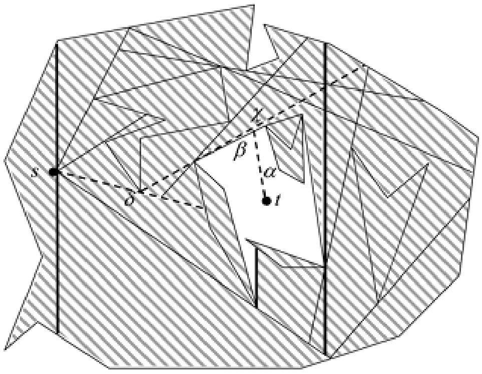
实现的算法也是类似于扫描线算法，以每个顶点为扫描的事件点。虽然可以通过控制每次扫描的复杂度来降低这个算法的复杂度，但是扫描事件点的处理上，程序实现比较复杂，这是该算法在实际应用上的一个问题。

**最短链接**

最短链接是通过可见性图的递归构造来解决。这时可以想象为光源扩散问题。扩散是从起点开始，以射线进行空间扩散。每次扩散后所到达的区域，其内的每个点都是新的扩散源，作为下一次扩散的起点。

如下图(a)所示，为一个简单的最短链接问题。(b)是 1 次可见性图，(c)是对应的简化处理。(d)是 2 次可见性图，(e)(f)再作简化处理。这时(g)再构造可见性图即覆盖至终点。我们注意到，最短链接对应的路径不是唯一的。





(g)

递归构造可见性图 [8]

我们非常希望能将最短路径和最短链接以说明中的“波浪扩散”[5]和“光源扩散”[7]的动态形式做成演示，展现给大家。但是这两种算法的原论文都在 40 页以上，阅读后发现其算法实现很复杂，我们小组的时间和精力不足以将其代码实现。所以仅在文档的最后部分加以说明。有兴趣的朋友可参考原论文。

# 参考文献

1. Mark de Berg, Marc van Kreveld, Mark Overmars, Otfried Schwarzkopf, 邓俊辉(译). *计算几何——算法与应用.*
2. Joseph O'Rourke. *Computational Geometry in C.*
3. D.T.Lee. Proximity and reachability in the plane.
4. S.K.Ghosh, D.M.Mount. An output-sensitive algorithm for computing visibility graphs.
5. Joseph S. B. Mitchell. Shortest paths among obstacles in the plane.
6. J. Hershberger, S. Suri. An optimal algorithm for Euclidean shortest path.
7. Joseph S. B. Mitchell, Gfinter Rote, Gerhard Woeginger. Minimum-link paths among obstacles in the plane.
8. Xiangzhi Wei, Ajay Joneja. On Minimum Link Monotone Path Problems.
9. https://lists-sop.inria.fr/sympa/arc/cgal-discuss/2010-04/msg00089.html Suryajith Chillara 2011-12-12
10. http://blog.bvn-usyd.com/tag/adjacency-graph/ 2011-12-15
11. http://tech.ddvip.com/2006-12/116514526412853.html 2011-12-14
12. http://www.cppblog.com/lzmagic/archive/2009/04/09/79329.html 2011-12-14
13. http://mecca.louisville.edu/~msabry/projects/images/robot/large/global_short.jpg 2011-12-23

# 附录

界面相关的文件

```
CvvImage.h, Resource.h, ShortestPath.h, ShortestPathDlg.h, SPMethod.h,
stdafx.h, targetver.h, CvvImage.cpp, ShortestPath.cpp,
ShortestPathDlg.cpp, SPMethod.cpp, stdafx.cpp
```

算法相关的文件

```
bstree.h        //实现了二叉搜索树的模板类
Geometric.h     //一些几何结构定义和常用几何关系判断及计算、最短路实现的类
```

Geometric.cpp //核心算法具体实现

核心算法的的实现

结构:

```
class Vertex;
class Edge;
class Polygonal;

class Polygonal
{
public:
    vector<Vertex *> vertices;  //多边形顶点集合
    int direction;        //指示沿着顶点集合，多边形的内部
    //0 表示逆时针, 1 顺时针。
    Polygonal(int d = 1) : direction(d)
    {
        vertices.clear();
    }
    void AddVertex(Vertex *v);  //添加顶点
    void CalcDirection();       //计算direction
    vector<Edge *> GetEdges();  //计算边的集合
};

class Vertex
{
public:
    CvPoint pos;       //顶点位置
    Polygonal *pg;     //顶点所属多边形
    Vertex *pre;       //used for shortest path
    double d;          //used for shortest path
    bool dealt;        //used for shortest path
    //...
};

class Edge
{
public:
    Vertex *v1;       //边的一个顶点
    Vertex *v2;       //边的另一个顶点
    //定义扫描状态中边的大小关系
    bool operator<(const Edge &e)
    {
        int x;
        int y;
```

```
        x = Geometric::ToLeft(v1->pos, v2->pos, e.v1->pos)
            + Geometric::ToLeft(v1->pos, v2->pos, e.v2->pos);
        y = Geometric::ToLeft(e.v1->pos, e.v2->pos, v1->pos)
            + Geometric::ToLeft(e.v1->pos, e.v2->pos, v2->pos);
        return x < y;
    }
    bool operator==(const Edge &e)
    {
        return (v1 == e.v1 && v2 == e.v2);
    }
};
```

常用几何关系判断及计算：

```
class Geometric
{
public:
    static const double eps; //定义‘’常量
    static int Sign(int val);   //判断符号
    //定义顶点扫描顺序
    static bool _pre_sweep1(Vertex *v1, Vertex *v2);
    static bool _pre_heap(Vertex *v1, Vertex *v2);
    //两点的欧氏距离
    static double EuclideanDis(CvPoint p1 = cvPoint(0, 0), CvPoint p2 =
cvPoint(0, 0));
    //p2 is to the left of p0->p1 or not
    //0--collinear, 1--to right, -1--to left
    static int ToLeft(CvPoint p0 = cvPoint(0, 0), CvPoint p1 = cvPoint(0,
0), CvPoint p2 = cvPoint(0, 0));
    //p2 is to the left of p0->p1 strictly
    static bool Left(CvPoint p0, CvPoint p1, CvPoint p2);
    //p2 is to the left of p0->p1 or on p0->p1
    static bool LeftOn(CvPoint p0, CvPoint p1, CvPoint p2);
    //the three points are collinear
    static bool Collinear(CvPoint p0, CvPoint p1, CvPoint p2);
    //p2 lies on the segment p0->p1
    static bool Between(CvPoint p0, CvPoint p1, CvPoint p2);
    //the siged area
    static int SignedArea(CvPoint p0 = cvPoint(0, 0), CvPoint p1 =
cvPoint(0, 0), CvPoint p2 = cvPoint(0, 0));
};

//核心算法的实现
class Graph
{
```

```
public:
    vector<Polygonal *> obstacles;      //障碍
    //for graph
    Vertex *s, *g;       //起始顶点和目标顶点
    vector<Vertex *> vertices;      //所有障碍物的顶点
    vector<Edge *> segments;     //障碍物的所有边
    vector<vector<Vertex *> > adjMatrix;   //可视图
    vector<vector<double> > disMatrix;     //可视图权重
    //for sweeping
    BSTree<Edge> sweepState;     //扫描线状态
    Graph(CvPoint _s = cvPoint(0, 0), CvPoint _g = cvPoint(0, 0));
    ~Graph();
    //清楚所有障碍
    void ClearObstacles();
    //添加一个障碍
    bool AddObstacle(Obstacle *obj);
    //添加预定义障碍
    bool AddPreDefined(Obstacle *obj);
    //添加用户画的障碍
    bool AddUserDrawn(Obstacle *obj);
    //删除一个障碍
    void DeleteObstacle(int index);
    //移动一个障碍
    void MoveObstacle(int index, const vector<CvPoint> &vertices);
    //移动一个顶点（障碍顶点或起始或目标顶点）
    void MoveVertex(int obj_index, int ver_index, const CvPoint &p);
    void AddPolygonVertex(Vertex *v);
    //设置起始点和目标点
    void SetSourcePoint(CvPoint p);
    void SetGoalPoint(CvPoint p);
    //初始化扫描线状态
    void InitializeState();
    //计算顶点v的事件（Event）集
    vector<Vertex *> WorkSetVertices(Vertex *v);
    //计算顶点v的可视（visible）点集
    vector<Vertex *> VisibleVertices(Vertex *v);
    //判断顶点v是否可见（在当前扫描线状态下）
    bool Visible(Vertex *v, vector<Vertex *> &workset);
    //构建可视图（visible graph）
    void Construct();
    //赋权重
    void AssignWeightsForVisGraph();
    //找到顶点v在多边形中的两个相邻点
    void GetNeighbors(Vertex *v, Vertex *&n1, Vertex *&n2);
```

```
    //Dijkstra 算法
    void Dijkstra();
    //支持目标点移动查询
    void ReachGoal();
    //获取最短路径
    vector<CvPoint> GetPath();
};
```

算法步骤：

①可视图的构造：

对vertices中的每个顶点v，找出其可见点集VisibleVertices(v)

②顶点v可见点的寻找：

先计算会被处理顶点集合wv = WorkSetVertices(v)，并相对于初始扫描线排好序；

对于wv中每个顶点p_i，判断是否可见Visible(p_i, wv);

③判断点是否可见：

如果i==0或者(v,p_i,p_i-1)不共线

判断扫描线状态sweepState最左边的边是否挡住p_i

挡住，则不可见

否则，可见

否则不可见

④更新扫描线状态：

找到与p_i相邻的处于同一多边形上的顶点GetNeighbors(p_i, n1, n2)而形成的至多两条边e1，e2

如果p_i是起点，则向sweepState中加入对应边

如果p_i是终点，则从sweepState中删除对应边