第一章　启发式搜索

1.1　教学目标

1. 让学生能够应用基于搜索的技术来实现问题求解智能体。能够学会进行问题建模，数据结构设计及A*搜索算法设计。

2. 学会使用Python工具完成计算及结果展示。

1.2　实验任务

图1.1是机器人导航问题的地图。机器人需要从起点 S出发，搜索目标点G. 图中存在一些凸多边形障碍，设计算法寻求从S点到G点的一个路径。

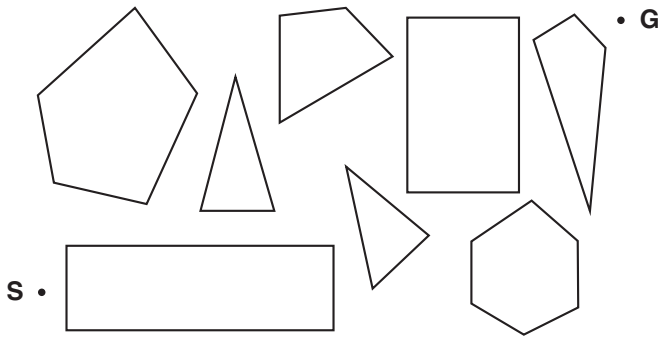

图 1.1: 机器人导航地图

1.3　实验环境

Windows/Linue操作系统，Python编译环境，queue, math，numpy, matplotlib等程序库

1.4　模型与方法

假设一条能绕过障碍物的到达路径是一个橡皮筋，我们在S点和D点用力拉，则直觉可以告诉我们路径最终拉成几段互联的直线段构成，折线的端点就是某些多边形障碍物的顶点。我们省略证明直接应用这个直觉。因此需要事先将多边形的顶点以及S点和G点互联，而这些连线不会穿过任何障碍物。

将多边形的各顶点以及S,G的集合定义为顶点集，将连线定义为边，将连线的长度定义为边的权值，问题成为一个求解S到G的最短路径问题。本实验应用A*算法进行求解。

1.4.1　线段相交

连线过程中，如果连线与障碍物相交，则线段一定与障碍物的某个边相交。线段AB和CD相交如图1.2所示.

相交的判断算法为线段AB的两个端点须位于CD的两侧，同时CD的两个端点也须位于AB的两侧，故需满足$\overrightarrow{AC} \times \overrightarrow{AD} > 0$且$\overrightarrow{CA} \times \overrightarrow{CB} > 0$

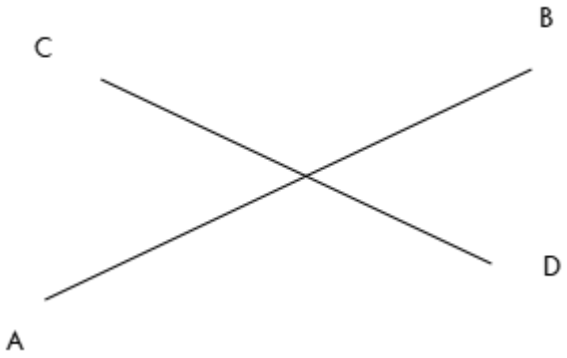

图 1.2: 线段交叉

1.4.2　邻接矩阵

邻接矩阵是图模型的一种表达方式。对于一个N个顶点的图，定义一个二维矩阵A, 其中$A[i,j]$为顶点i和j构成的边(i,j)的权值。A满足：

1. $A[i,i]=0$, 即每个顶点和自身没有边；
2. 如果边(i,j)不存在，则$A[i,j]=MAX$；
3. 如果边(i,j)存在, 则$A[i,j]<MAX$；

其中MAX为表示正无穷大的常数。

1.4.3　AI问题求解模型

邻接矩阵模型是图论中常用的模型，但本实验中要求将邻接矩阵模型转换为Agent的问题求解模型。在问题模型中，图的顶点对应Agent所历经的状态，边对应某状态下可以选择的行为，边的另一个顶点则是采取行为Agent达到的新状态，边的权重则是采取行为的费用。

```
class Problem():
def __init__(self,points,Adj,start, goal):
    self.Adj=Adj    #记录邻接矩阵备用
    self.Points=points    #记录每个顶点坐标备用
    self.InitialState=start #起始状态
    self.GoalState=goal   #目标状态
    def GoalTest(self,state)    #测试是否到达目标
    def Action(self,state)        #获取某状态下的行为集合
    def Result(self,state, action)
    #获取在某状态下采取某行为后的新状态
    def StepCost(self,state,action) #获取在某状态下采取某行为需要的
    费用
```

1.4.4　A*算法

A*算法是一种图搜索技术。图搜索过程中，从起点开始，将待搜索的结点加入一个前沿集合（frontier）。每次从前沿中取出一个点访问，并将该点的邻居结点加入前沿。访问在遇到目标状态或者前沿为空时结束。Agent对一个图结点定义启发式信息$f(node)$，作为从前沿中优先选择访问

结点的依据。本实验中$f(node) = g(node) + h(node)$, 其中$g(node)$到$node$所在状态下agent 已走过路径的总长，$h(node)$是$node$状态到目标状态的预测路径长度，本实验中$h$定义为到目标状态的欧式距离。Agent优先选择$f$值更小的结点进行探索。结点的定义如下：

```
class Node():
    def __init__(self,problem, parent=None,action=None):
        if parent == None:                    #起始结点
            self.State=problem.InitialState
            self.Parent=None
            self.Action=None
            self.PathCost=0
        else:
            self.State=problem.Result(parent.State,action)
            #子结点
            self.Parent=parent   #生成此结点的父亲结点
            self.Action=action    #生成此结点的行为
            self.PathCost=parent.PathCost+problem.StepCost(
    parent.State,action)
            #到此结点路径长度
        self.g=self.PathCost    #g信息
        self.h=intersect.distance(problem.Points[self.State],
    problem.Points[problem.GoalState])
        #h 信息
        self.f=self.g+self.h    #f信息
    def __lt__(self, other):
      return other.f > self.f
```

从node的定义可知，根据node.parent可以回溯出整个解决方案所到达的state和相应的action序列，因此可设计一个函数Solution(node)获得这些序列。

A*算法的关键在于使用优先队列来存储待探索的前沿结点，f值更小的结点将优先被探索。具体算法定义如下：

```
def Astar(problem):
    node=Node(problem)   #起始结点
    if problem.GoalTest(node.State):
        return Solution(node)
    frontier=PriorityQueue()
    #前沿是一个优先队列，f值最小的结点将优先被探索
```

```
    frontier.put(node)    #第一个结点进入
    explored=set()        #存储正在或者已经探索的状态
    while frontier.qsize()>0:
        node=frontier.get()           #取出结点进入前沿
        explored.add(node.State)      #记录探索过的状态
        for action in problem.Action(node.State):
        #遍历对可采取的行为
            child=Node(problem,node,action)
            #生成子结点
            if child.State not in explored:
                if problem.GoalTest(child.State):
                #发现目标状态
                    return Solution(child)
                frontier.put(child)       #子结点进入前沿
                explored.add(child.State)
                #前沿中的节点状态也要记录
```

1.5　实验步骤

1. 使用一个图像处理软件，获取任务图片中各障碍物顶点、S和G的像素坐标。假设S的坐标为(34.1, 213)，用S坐标对其他所有点坐标进行标定。

2. 建立图模型，获得邻接矩阵 Adj.

3. 实现problem, 将邻接矩阵模型转化成Agent问题求解模型.

4. 实现A*算法.

5. 应用A*算法获得最优解，并根据solution的路径序列画图。

6. 更改S的坐标为(34.1, 113), (160, 260)，获得不同的路径图。

7. 实现一个Dijakstra算法，对A*算法和Dijakstra的性能进行比较分析。

图1.1的最优路径图如1.3所示。

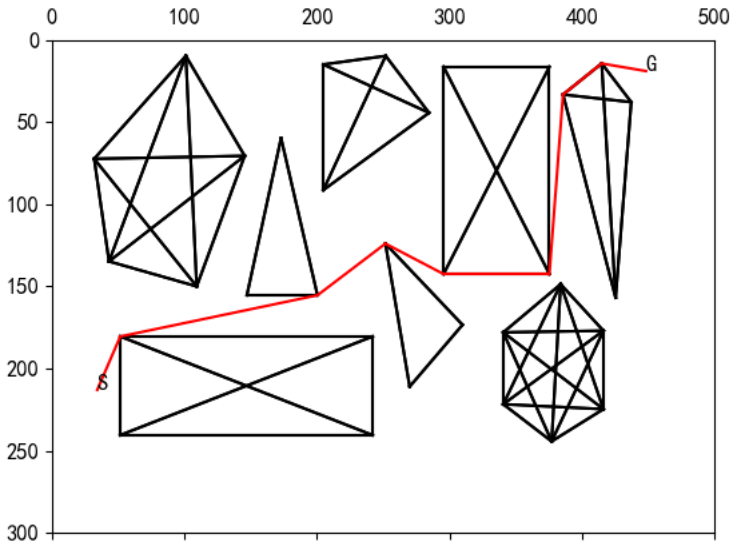

图 1.3: 最短路径

1.6　实验报告

按实验步骤完成实验后，提供数据和代码，并完成对实验结果的分析和总结，撰写实验报告。