实验一 启发式搜索

### 姓名：李思博 学号：202108120021011

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。

1. 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。

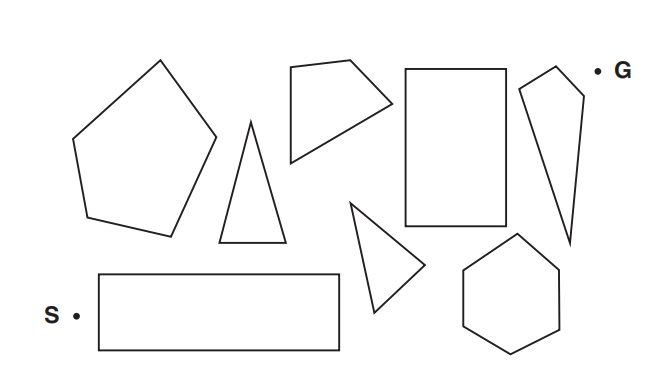


图 1.1: 机器人导航地图

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

1.5 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。

## 1.6 实现结果

1.点位于（34，213），结果如图1：

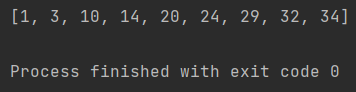


图 1

路径图片，如图2（结果和老师的不同，可能是因为标点误差导致，验算结果也是这一条路线较近，验算结果如图3）：

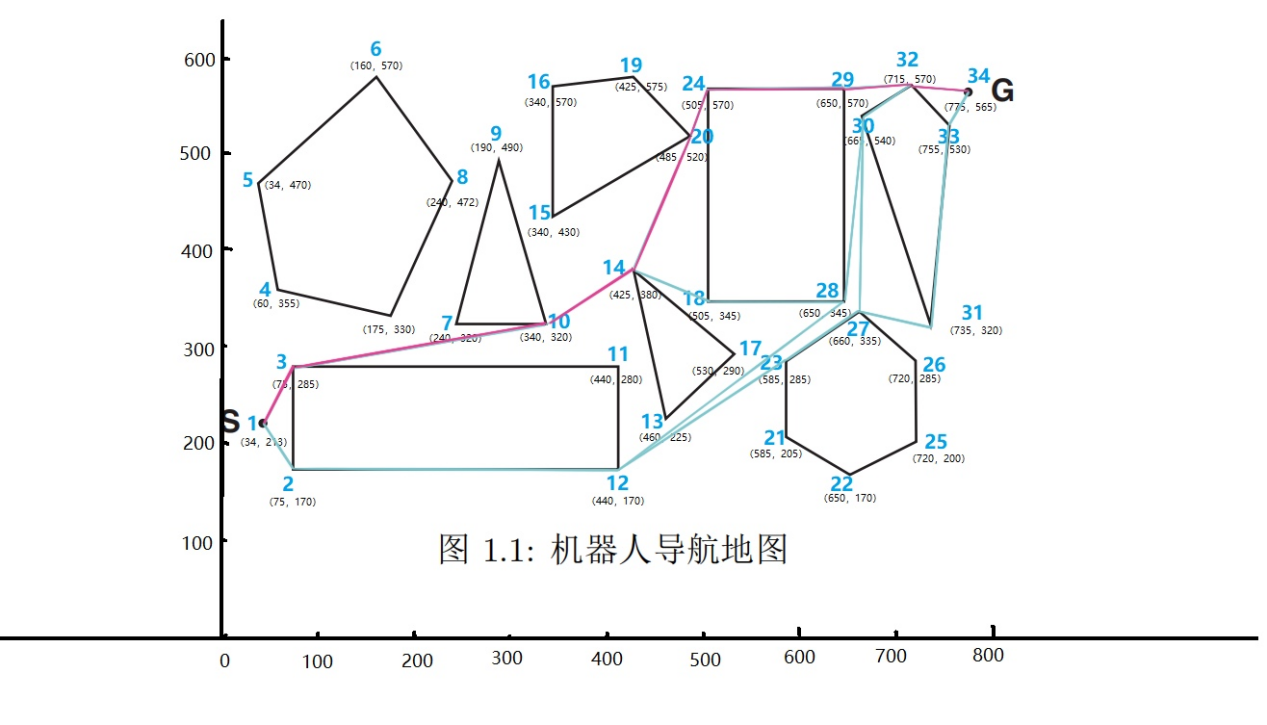


图 2

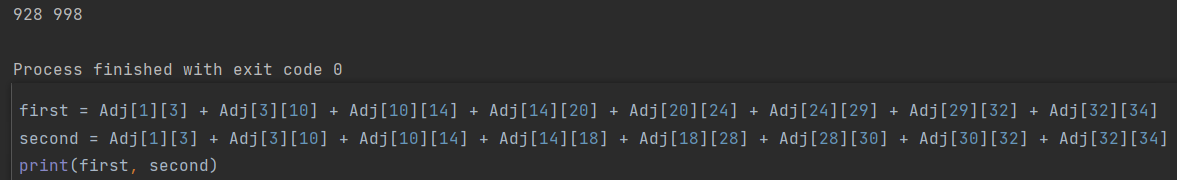


图 3

2.点位于（34，113），结果如图4：

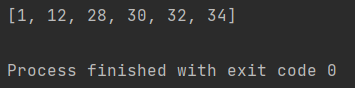


图 4

路径图片，如图5：

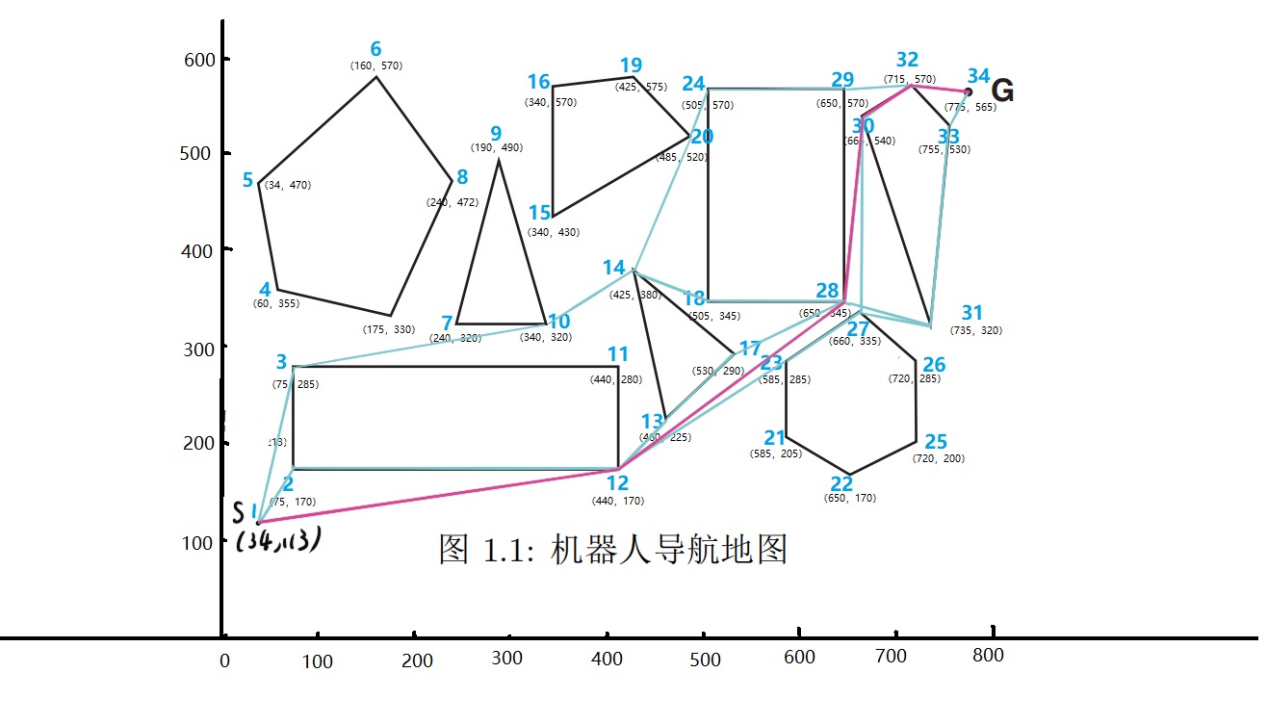


图 5

1. 点位于（260，160）（由于图片比例尺不知道，所以原来的点存在于障碍物内，重新换了一个点），结果如图6：

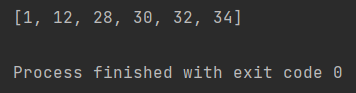


图 6

路径图片，如图7：

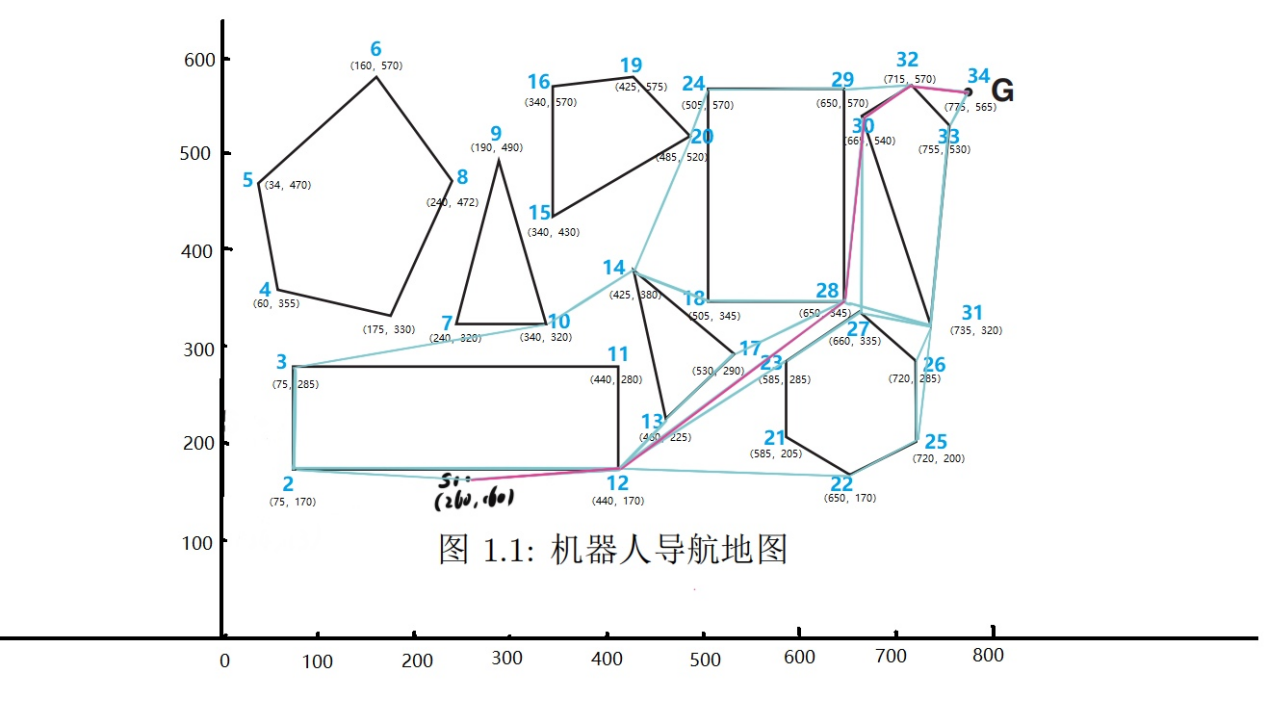


图 7

1.7 Dijakstra算法

整体架构和A\*算法相似，设置一个起始点到各个点最短路径的数组。由起始点出发，可以直接到达的点填写距离，不能到达的点取无穷大。

D = [float('inf') for i in range(35)] # 存放起始点到各个点的最短距离

其余代码大致类似，只是在判断下一步行为节点时有所不同，若节点未曾探索过，则直接用父亲节点的pathcost加上父亲节点到当前节点的cost赋值到D数组对应的下标。

若节点已经探索过，则用D数组中已经记录的pathcost值与父亲节点的pathcost值加上父亲节点到当前节点的cost值进行比较，记录较小的那一个。

if child.State not in explored:  
 D[child.State] = child.PathCost # 未探索过时的D值  
 if problem.GoalTest(child.State):   
 return Solution(child)  
 frontier.put(child)   
 explored.add(child.State)   
elif child.State in explored:  
 D[child.State] = min(D[child.State], D[node.State] + Adj[node.State][child.State]) # 探索过更新D值

Dijakstra算法结果如图8：

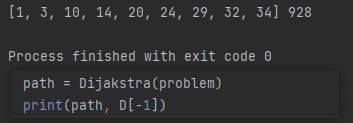


图 8

总结比较：

两种算法进行比较，Dijakstra算法和A\*的时间复杂度进行比较，明显是A\*的时间复杂度更低，因为省去了已经探索过的节点的处理。而Dijakstra算法会对所有的行为进行比较。

而在空间复杂度方面，Dijakstra算法需要较A\*算法额外开辟一个数组用来存储最终的值。A\*代码：

# 点对应状态，边对应行为  
  
from sqlalchemy import intersect  
from queue import PriorityQueue  
  
points = [(0, 0), (34, 213), (75, 170), (75, 285), (60, 355), (34, 470), (160, 570), (240, 320), (240, 472), (190, 490),  
 (340, 320), (440, 280), (440, 170), (460, 225), (425, 380), (340, 430), (340, 570), (530, 290), (505, 345),  
 (425, 575), (485, 520), (585, 205), (650, 170), (585, 285), (505, 570), (720, 200), (720, 285), (660, 335),  
 (650, 345), (650, 570), (665, 540), (735, 320), (715, 570), (755, 530), (775, 565)]  
  
dd = float('inf')  
Adj = [[float('inf') for i in range(35)] for j in range(35)] # 初始化邻接矩阵  
for i in range(35): # A[i][i]=0 ：每个顶点和自身没有边  
 Adj[i][i] = 0  
# 边  
Adj[1][2] = Adj[2][1] = int(pow(pow(points[1][0] - points[2][0], 2) + pow(points[1][1] - points[2][1], 2), 0.5))  
Adj[1][3] = Adj[3][1] = int(pow(pow(points[1][0] - points[3][0], 2) + pow(points[1][1] - points[3][1], 2), 0.5))  
Adj[3][10] = Adj[10][3] = int(pow(pow(points[3][0] - points[10][0], 2) + pow(points[3][1] - points[10][1], 2), 0.5))  
Adj[10][14] = Adj[14][10] = int(pow(pow(points[10][0] - points[14][0], 2) + pow(points[10][1] - points[14][1], 2), 0.5))  
Adj[14][20] = Adj[20][14] = int(pow(pow(points[14][0] - points[20][0], 2) + pow(points[14][1] - points[20][1], 2), 0.5))  
Adj[20][24] = Adj[24][20] = int(pow(pow(points[20][0] - points[24][0], 2) + pow(points[20][1] - points[24][1], 2), 0.5))  
Adj[24][29] = Adj[29][24] = int(pow(pow(points[24][0] - points[29][0], 2) + pow(points[24][1] - points[29][1], 2), 0.5))  
Adj[29][32] = Adj[32][29] = int(pow(pow(points[29][0] - points[32][0], 2) + pow(points[29][1] - points[32][1], 2), 0.5))  
Adj[32][34] = Adj[34][32] = int(pow(pow(points[32][0] - points[34][0], 2) + pow(points[32][1] - points[34][1], 2), 0.5))  
Adj[14][18] = Adj[18][14] = int(pow(pow(points[14][0] - points[18][0], 2) + pow(points[14][1] - points[18][1], 2), 0.5))  
Adj[18][28] = Adj[28][18] = int(pow(pow(points[18][0] - points[28][0], 2) + pow(points[18][1] - points[28][1], 2), 0.5))  
Adj[28][30] = Adj[30][28] = int(pow(pow(points[28][0] - points[30][0], 2) + pow(points[28][1] - points[30][1], 2), 0.5))  
Adj[30][32] = Adj[32][30] = int(pow(pow(points[30][0] - points[32][0], 2) + pow(points[30][1] - points[32][1], 2), 0.5))  
Adj[2][12] = Adj[12][2] = int(pow(pow(points[2][0] - points[12][0], 2) + pow(points[2][1] - points[12][1], 2), 0.5))  
Adj[12][23] = Adj[23][12] = int(pow(pow(points[12][0] - points[23][0], 2) + pow(points[12][1] - points[23][1], 2), 0.5))  
Adj[23][27] = Adj[27][23] = int(pow(pow(points[23][0] - points[27][0], 2) + pow(points[23][1] - points[27][1], 2), 0.5))  
Adj[27][30] = Adj[30][27] = int(pow(pow(points[27][0] - points[30][0], 2) + pow(points[27][1] - points[30][1], 2), 0.5))  
Adj[27][31] = Adj[31][27] = int(pow(pow(points[27][0] - points[31][0], 2) + pow(points[27][1] - points[31][1], 2), 0.5))  
Adj[31][33] = Adj[33][31] = int(pow(pow(points[31][0] - points[33][0], 2) + pow(points[31][1] - points[33][1], 2), 0.5))  
Adj[33][34] = Adj[34][33] = int(pow(pow(points[33][0] - points[34][0], 2) + pow(points[33][1] - points[34][1], 2), 0.5))  
Adj[12][28] = Adj[28][12] = int(pow(pow(points[12][0] - points[28][0], 2) + pow(points[12][1] - points[28][1], 2), 0.5))  
  
  
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): # 测试是否达到目标  
 if state == self.GoalState:  
 return True  
  
 def Action(self, state): # 获取某状态下的行为集合  
 states = []  
 for i in range(state + 1, 35):  
 if Adj[state][i] != float('inf'):  
 states.append(i)  
 return states  
  
 def Result(self, state, action): # 获取在某状态下采取某行为后的新状态  
 return action  
  
 def StepCost(self, state, action): # 获取在某状态下采取某行为需要的费用  
 return Adj[state][action]  
  
  
class Node():  
 def \_\_init\_\_(self, problem, parent=None, action=None):  
 if parent is None:  
 self.State = problem.InitialState  
 self.Parent = None  
 self.Action = None  
 self.PathCost = 0  
 else:  
 self.State = problem.Result(parent.State, action) # 子节点  
 self.Parent = parent # 生成此节点的父亲节点  
 self.Action = action # 生成此节点的行为  
 self.PathCost = parent.PathCost + problem.StepCost(parent.State, action) # 到此节点路径长度  
 self.g = self.PathCost # g信息  
 self.h = distance(problem.Points[self.State], problem.Points[problem.GoalState]) # h信息 # 没有distance方法  
 self.f = self.g + self.h # f信息  
  
 def \_\_lt\_\_(self, other): # 优先队列判别条件  
 return other.f > self.f  
  
  
def distance(a, b): # 距离函数  
 return int(pow(pow(a[0] - b[0], 2) + pow(a[1] - b[1], 2), 0.5))  
  
  
def Solution(node): # 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。  
 res = []  
 while node:  
 res.append(node.State)  
 node = node.Parent  
 return res[::-1]  
  
  
def Astar(problem):  
 node = Node(problem) # 起始节点  
 if problem.GoalTest(node.State):  
 return Solution(node)  
 frontier = PriorityQueue() # frontier:前沿; PriorityQueue():是一个优先队列，f值最小的节点将优先被探索  
 frontier.put(node) # 第一个节点进入  
 explored = set() # 存储正在或者已经探索的状态  
 while frontier.qsize() > 0: # qsize()：队列尺寸  
 node = frontier.get() # 取出前沿中的节点，前沿大小减一  
 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) # 生成子节点  
 if child.State not in explored:  
 if problem.GoalTest(child.State): # 发现目标状态  
 return Solution(child)  
 frontier.put(child) # 子节点进入前沿  
 explored.add(child.State) # 前沿中的节点状态也要记录  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 problem = Problem(points, Adj, 1, 34)  
 res = Astar(problem)  
 print(res)

### Dijakstra代码：

# 点对应状态，边对应行为  
  
from sqlalchemy import intersect  
from queue import PriorityQueue  
  
points = [(0, 0), (34, 213), (75, 170), (75, 285), (60, 355), (34, 470), (160, 570), (240, 320), (240, 472), (190, 490),  
 (340, 320), (440, 280), (440, 170), (460, 225), (425, 380), (340, 430), (340, 570), (530, 290), (505, 345),  
 (425, 575), (485, 520), (585, 205), (650, 170), (585, 285), (505, 570), (720, 200), (720, 285), (660, 335),  
 (650, 345), (650, 570), (665, 540), (735, 320), (715, 570), (755, 530), (775, 565)]  
  
dd = float('inf')  
Adj = [[float('inf') for i in range(35)] for j in range(35)] # 初始化邻接矩阵  
for i in range(35): # A[i][i]=0 ：每个顶点和自身没有边  
 Adj[i][i] = 0  
# 边  
Adj[1][2] = Adj[2][1] = int(pow(pow(points[1][0] - points[2][0], 2) + pow(points[1][1] - points[2][1], 2), 0.5))  
Adj[1][3] = Adj[3][1] = int(pow(pow(points[1][0] - points[3][0], 2) + pow(points[1][1] - points[3][1], 2), 0.5))  
Adj[3][10] = Adj[10][3] = int(pow(pow(points[3][0] - points[10][0], 2) + pow(points[3][1] - points[10][1], 2), 0.5))  
Adj[10][14] = Adj[14][10] = int(pow(pow(points[10][0] - points[14][0], 2) + pow(points[10][1] - points[14][1], 2), 0.5))  
Adj[14][20] = Adj[20][14] = int(pow(pow(points[14][0] - points[20][0], 2) + pow(points[14][1] - points[20][1], 2), 0.5))  
Adj[20][24] = Adj[24][20] = int(pow(pow(points[20][0] - points[24][0], 2) + pow(points[20][1] - points[24][1], 2), 0.5))  
Adj[24][29] = Adj[29][24] = int(pow(pow(points[24][0] - points[29][0], 2) + pow(points[24][1] - points[29][1], 2), 0.5))  
Adj[29][32] = Adj[32][29] = int(pow(pow(points[29][0] - points[32][0], 2) + pow(points[29][1] - points[32][1], 2), 0.5))  
Adj[32][34] = Adj[34][32] = int(pow(pow(points[32][0] - points[34][0], 2) + pow(points[32][1] - points[34][1], 2), 0.5))  
Adj[14][18] = Adj[18][14] = int(pow(pow(points[14][0] - points[18][0], 2) + pow(points[14][1] - points[18][1], 2), 0.5))  
Adj[18][28] = Adj[28][18] = int(pow(pow(points[18][0] - points[28][0], 2) + pow(points[18][1] - points[28][1], 2), 0.5))  
Adj[28][30] = Adj[30][28] = int(pow(pow(points[28][0] - points[30][0], 2) + pow(points[28][1] - points[30][1], 2), 0.5))  
Adj[30][32] = Adj[32][30] = int(pow(pow(points[30][0] - points[32][0], 2) + pow(points[30][1] - points[32][1], 2), 0.5))  
Adj[2][12] = Adj[12][2] = int(pow(pow(points[2][0] - points[12][0], 2) + pow(points[2][1] - points[12][1], 2), 0.5))  
Adj[12][23] = Adj[23][12] = int(pow(pow(points[12][0] - points[23][0], 2) + pow(points[12][1] - points[23][1], 2), 0.5))  
Adj[23][27] = Adj[27][23] = int(pow(pow(points[23][0] - points[27][0], 2) + pow(points[23][1] - points[27][1], 2), 0.5))  
Adj[27][30] = Adj[30][27] = int(pow(pow(points[27][0] - points[30][0], 2) + pow(points[27][1] - points[30][1], 2), 0.5))  
Adj[27][31] = Adj[31][27] = int(pow(pow(points[27][0] - points[31][0], 2) + pow(points[27][1] - points[31][1], 2), 0.5))  
Adj[31][33] = Adj[33][31] = int(pow(pow(points[31][0] - points[33][0], 2) + pow(points[31][1] - points[33][1], 2), 0.5))  
Adj[33][34] = Adj[34][33] = int(pow(pow(points[33][0] - points[34][0], 2) + pow(points[33][1] - points[34][1], 2), 0.5))  
Adj[12][28] = Adj[28][12] = int(pow(pow(points[12][0] - points[28][0], 2) + pow(points[12][1] - points[28][1], 2), 0.5))  
  
  
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): # 测试是否达到目标  
 if state == self.GoalState:  
 return True  
  
 def Action(self, state): # 获取某状态下的行为集合  
 states = []  
 for i in range(1, 35):  
 if Adj[state][i] != float('inf') and Adj[state][i] != 0:  
 states.append(i)  
 return states  
  
 def Result(self, state, action): # 获取在某状态下采取某行为后的新状态  
 return action  
  
 def StepCost(self, state, action): # 获取在某状态下采取某行为需要的费用  
 return Adj[state][action]  
  
  
class Node():  
 def \_\_init\_\_(self, problem, parent=None, action=None):  
 if parent is None:  
 self.State = problem.InitialState  
 self.Parent = None  
 self.Action = None  
 self.PathCost = 0  
 else:  
 self.State = problem.Result(parent.State, action) # 子节点  
 self.Parent = parent # 生成此节点的父亲节点  
 self.Action = action # 生成此节点的行为  
 self.PathCost = parent.PathCost + problem.StepCost(parent.State, action) # 到此节点路径长度  
 self.g = self.PathCost # g信息  
 self.h = distance(problem.Points[self.State], problem.Points[problem.GoalState]) # h信息 # 没有distance方法  
 self.f = self.g + self.h # f信息  
  
 def \_\_lt\_\_(self, other): # 优先队列判别条件  
 return other.f > self.f  
  
  
def distance(a, b): # 距离函数  
 return int(pow(pow(a[0] - b[0], 2) + pow(a[1] - b[1], 2), 0.5))  
  
  
def Solution(node): # 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。  
 res = []  
 while node:  
 res.append(node.State)  
 node = node.Parent  
 return res[::-1]  
  
  
D = [float('inf') for i in range(35)] # 存放起始点到各个点的最短距离  
  
  
def Dijakstra(problem):  
 node = Node(problem) # 起始节点  
 if problem.GoalTest(node.State):  
 return Solution(node)  
 frontier = PriorityQueue() # frontier:前沿; PriorityQueue():是一个优先队列，f值最小的节点将优先被探索  
 frontier.put(node) # 第一个节点进入  
 explored = set() # 存储正在或者已经探索的状态  
 while frontier.qsize() > 0: # qsize()：队列尺寸  
 node = frontier.get() # 取出前沿中的节点，前沿大小减一  
 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) # 生成子节点  
 if child.State not in explored:  
 D[child.State] = child.PathCost  
 if problem.GoalTest(child.State): # 发现目标状态  
 return Solution(child)  
 frontier.put(child) # 子节点进入前沿  
 explored.add(child.State) # 前沿中的节点状态也要记录  
 elif child.State in explored:  
 D[child.State] = min(D[child.State], D[node.State] + Adj[node.State][child.State])  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 problem = Problem(points, Adj, 1, 34)  
 path = Dijakstra(problem)  
 print(path, D[-1])