A\* 算法的简单实现，在网上找的罗马尼亚度假问题。下面算法的描述等自己写的，代码多有借鉴。

描述：A\*算法解决自动寻路的算法，不是完美的但比较接近。

算法将每一个节点看做一个正方形（也可以是其他形状，但这里我用的正方形），每个节点有八个不同方向的相邻节点，假设在同行同列的行走代价是10，斜对角线的就是14。

算法中需要用到两个列表，一个open一个close，open list用来装即将用到的节点，close装入已经确定的节点。

算法的灵魂是一个公式F=G+H（G:节点运动的代价H为此节点到终点的理想距离（无视障碍物））

每一个节点都有自己的F，节点寻路过程就是一个寻找相邻最小F节点的过程

算法实现：代码基本上借鉴来的，注释自己写的

A=0  
B=1  
C=2  
D=3  
E=4  
F=5  
G=6  
H=7  
I=8  
L=9  
M=10  
N=11  
O=12  
P=13  
R=14  
S=15  
T=16  
U=17  
V=18  
Z=19  
PLACES\_NUM=20*#构建一个20\*20的矩阵  
#地图***class** Graph:  
 **def** \_\_init\_\_(self):  
 self.remania\_graph=[[0]\*PLACES\_NUM **for** i **in** range(PLACES\_NUM)]

#【0】\*x为构建一个有x个字符串的的列表  
 **def** add\_edge(self,\_from,\_to,\_value):  
 **if** \_from < PLACES\_NUM:  
 **if** \_to < PLACES\_NUM:  
 self.remania\_graph[\_from][\_to]=\_value  
 self.remania\_graph[\_to][\_from]  
 **def** get\_edge(self,\_from,\_to):  
 **return** self.remania\_graph[\_from][\_to]  
**class** Stack:*#问题的最终解(构建一个栈)* **def** \_init\_\_(self):  
 self.stack=[]  
 **def** push(self,value):  
 self.stack.append(value)  
 **return  
 def** pop(self):  
 **return** self.stack.pop()  
 **def** is\_empty(self):  
 **if** self.stack:  
 **return False  
 return True***#close表***class** Queue:  
 **def** \_\_init\_\_(self):  
 self.queue=[]  
 **def** put(self,value):  
 self.queue.append(value)  
 **return  
 def** get(self):  
 **return** self.queue.pop(0)  
 **def** contain(self,value):  
 **return** value **in** self.queue  
 **def** is\_empty(self):  
 **if** self.queue:  
 **return False  
 return True***#open表***class** PriortyQueue:  
 **def** \_\_init\_\_(self):  
 self.queue=[]  
 **def** put(self,node\_cost):  
 self.queue.append(node\_cost)  
 **def** get(self):#寻找邻点花费最小值  
 **if** self.queue:  
 min\_i=0  
 min\_cost=self.queue[min\_i][1]  
 **for** i **in** range(len(self.queue)):  
 **if** self.queue[i][1]<min\_cost:  
 min\_i=i  
 min\_cost=self.queue[i][1]  
 **return** self.queue.pop(min\_i)  
 **def** contain(self,value):  
 **for** i **in** range(len(self.queue)):  
 **if** self.queue[i][0]==value:  
 **return** self.queue[i],**True  
 return None**,**False  
 def** is\_empty(self):  
 **if** self.queue:  
 **return False  
 return True***#初始化无向图***def** init\_graph():  
 graph =Graph()  
 graph.add\_edge(O,Z,71)  
 graph.add\_edge(O,S,151)  
 graph.add\_edge(A,Z,75)  
 graph.add\_edge(A,S,140)  
 graph.add\_edge(A,T,118)  
 graph.add\_edge(T,L,111)  
 graph.add\_edge(L,M,70)  
 graph.add\_edge(M,D,75)  
 graph.add\_edge(D,C,120)  
 graph.add\_edge(S,R,80)  
 graph.add\_edge(S,F,99)  
 graph.add\_edge(R,C,146)  
 graph.add\_edge(F,B,211)  
 graph.add\_edge(R,P,97)  
 graph.add\_edge(C,P,138)  
 graph.add\_edge(P,B,101)  
 graph.add\_edge(B,G,90)  
 graph.add\_edge(B,U,85)  
 graph.add\_edge(U,H,98)  
 graph.add\_edge(U,V,142)  
 graph.add\_edge(V,I,92)  
 graph.add\_edge(I,N,87)  
 graph.add\_edge(H,E,86)  
  
 **return** graph  
**def** a\_star(graph,h,\_root,\_goal):  
 g=[0]\*PLACES\_NUM  
 parents=[0]\*PLACES\_NUM  
 pq\_open=PriortyQueue()  
 q\_close=Queue()  
 s\_path=Stack()  
 s\_parent=Stack()  
 pq\_open.put([\_root,0])  
  
 **while** pq\_open.is\_empty()==**False**:  
 parrent\_node=pq\_open.get()  
 **if** parrent\_node==\_goal:  
 **break** q\_close.put(parrent\_node[0])  
 **for** i **in** range(20):  
 length=graph.get\_edge(parrent\_node[0],i)  
 **if** length!=0:  
 node,result=pq\_open.contain(i)  
 f=parrent\_node[1]-h[parrent\_node[0]]+length+h[i]  
 **if** q\_close.contain(i):  
 **continue  
 elif** result==**True**:  
 **if** node[1]>f:  
 node.pop()  
 parents[i]=parrent\_node[0]  
 pq\_open.put([i,f])  
 **else**:  
 parents[i]=parrent\_node[0]  
 pq\_open.put([i,f])  
 path=[]  
 cost=0  
 print(**"parents:"**,parents)  
 p=\_goal  
 **while** p!=\_root:  
 cost +=graph.get\_edge(p,parents[p])  
 path.append(p)  
 p=parents[p]  
 length=len(path)-1  
 print(**'path:'**,\_root,end=**''**)  
 **for** i **in** range(length+1):  
 print(**"-->"**,path[length-i],end=**''**)  
 print()  
 **return** cost  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 graph=init\_graph()  
 h=(366,0,160,242,161,178,77,151,226,244,241,234,380,98,193,253,329,80,199,374)  
 print(**"node:"**,**"A"**,**"B"**)  
 cost=a\_star(graph,h,eval(**'A'**),eval(**'B'**))  
 print(**'cost:'**,cost)  
 print()  
 print(**"node"**,**"C"**,**"B"**)  
 cost=a\_star(graph,h,eval(**'C'**),eval(**'B'**))  
 print(**'cost:'**,cost)  
 print()  
 print(**"node"**,**"U"**,**"B"**)  
 cost=a\_star(graph,h,eval(**'U'**),eval(**'B'**))  
 print(**'cost:'**,cost)  
 print()

