人工智能lab8实验报告

学号：17338233 专业：计科 姓名：郑戈涵

本次实验的无信息搜索算法选择**一致代价搜索**，启发式搜索算法选择**A*搜索**。

一致代价搜索

算法原理

一致代价搜索在广度优先搜索上做了改进，扩展节点时优先选择代价最低的节点。

即每次在边界集合中选出用于扩展的节点，标记该节点已访问，并遍历每个可行的动作，找出未访问的节点加入边界集合，访问过的节点如果代价更小，则更新边界集合。

对于本次实验，每个动作的代价一样，因此和BFS的效果一样。

- 一致代价搜索具有**完备性**和**最优性**。
  - 完备性：如果存在满足要求的目标，则最终可以到达
  - 最优性：不存在可以达到目标且开销更小的路径
- 时间复杂度：$O(b^{\lfloor C^*/\epsilon \rfloor+1})$，其中$\epsilon$为状态变化的最小代价，$C^*$为最优解的开销。
- 空间复杂度：$O(b^{\lfloor C^*/\epsilon \rfloor+1})$

本次实验的可行动作数为4，因此b=4。

伪代码

```
function Uniform-Cost-Search(problem) returns a solution or failure
node <-start
frontier<- priority queue ordered by path-cost
push(frontier,node)
loop do
    if empty(frontier) then
        return failure
    node<-pop(frontier)
    if goal(node) return solution(node)
    add node.state to explored
    for each action in problem.actions(node.state) do
        child<-Node(problem,node,action)
        if child.state not in explored or frontier then
            insert(frontier,child)
        else if child.state in frontier(old-child) with higher path-cost
then
            replace(frontier,old-child,child)
```

代码展示

抽象出状态空间的元素，对应节点类。抽象动作为四个方向的移动。

```
class node:
    """
    定义搜索的节点类
    """
```

```
    def __init__(self,pos,cost=0,estimate=0,ancestor=None):
        self.pos=pos
        self.cost=cost
        self.estimate=estimate
        self.ancestor=ancestor
    def __lt__(self,rhs):
        return self.cost+self.estimate<rhs.cost+rhs.estimate
class direction(enum.Enum):
    """
    可以移动的四个方向
    """
    UP=(0,1)
    DOWN=(0,-1)
    RIGHT=(1,0)
    LEFT=(-1,0)
def move(pos,direction):
    """
    移动的结果
    """
    return pos[0]+direction.value[0],pos[1]+direction.value[1]
def isValid(pos):
    """
    判断是否为可行位置
    """
    x,y=pos
    if x<0 or x>=len(maze) or y<0 or y>=len(maze[0]) or maze[x][y] == '1':
        return False
    else:
        return True
```

一致代价搜索根据算法实现，使用堆存储frontier。

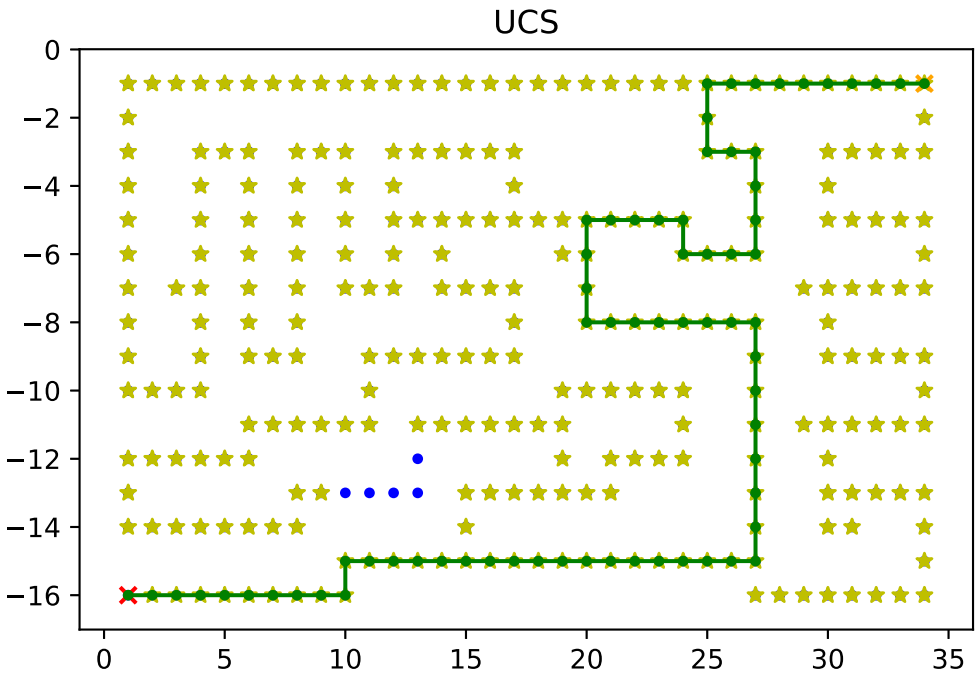
```
def UCS(maze,start,end):
    spatial_complexity=0
    time_complexity=0

    frontier = []
    route=[]
    visited = [[0] * len(maze[0]) for _ in range(len(maze))]
    heapq.heappush(frontier,node(start))
    while True:
        if len(frontier)==0:
            return False

        curr=heapq.heappop(frontier)
        if curr.pos==end:
            print("空间复杂度:{}, 时间复杂度:
{}".format(spatial_complexity,time_complexity))
            return [obj.pos for obj in getRoute(curr)],visited
        visited[curr.pos[0]][curr.pos[1]]=1
        time_complexity+=1
        # 遍历可能的动作
        for d in direction:
            new_pos=move(curr.pos,d)
            # 节点在frontier中，若cost更小，更新frontier
            for Node in frontier:
                if Node.pos==new_pos and Node.cost>curr.cost+1:
```

```
                    Node.cost=curr.cost+1
                    Node.ancestor=curr
                    heapq.heapify(frontier)
                    break
            # 节点不在frontier中，未被访问过，则放入状态空间(堆)中

            if visited[new_pos[0]][new_pos[1]]==0 and isValid(new_pos):

heapq.heappush(frontier,node(new_pos,cost=curr.cost+1,ancestor=curr))
                spatial_complexity=max(spatial_complexity,len(frontier))
```

实验结果以及分析

下面是UCS算法对迷宫的运行结果，绿色线代表路径，黄色的为搜索过的路径，有颜色的点都是迷宫的可行点，蓝色为未被搜索的路径。可以看出，大部分路径都已经被搜索过了，效率比较低。打印的结果为：

```
空间复杂度:9, 时间复杂度:274
```



A*搜索

算法原理

A*搜索是有信息的搜索，信息体现在它会考虑当前已走的开销和未来估计的开销。使用估值函数

$$f(n)=g(n)+h(n)$$

其中$g(n)$为起始点到节点n的代价(cost)，$h(n)$为n到目标的代价，用$f(n)$对边界集合内的节点进行排序。**当$f(n)$相同时，再用$h(n)$进行排序。**

估值函数$f(n)$需要有两个性质，才能保证最优性。

1. 可采纳性

$$Suppose\ c(n1 \to n2) \geq \epsilon > 0,$$
$$h^*(n) := least\ cost\ from\ n\ to\ target$$
$$\forall n : h(n) \leq h^*(n)$$

2. 一致性/单调性

$$h(n_1) \leq c(n_1 \to n_2) + h(n_2)$$

A*搜索的时间复杂度和空间复杂度和UCS是一样的，因为$h(n) = 0$则退化为UCS。

伪代码

```
function Uniform-Cost-Search(problem) returns a solution or failure
node <-start
frontier<- priority queue ordered by heuristic-path-cost
push(frontier,node)
loop do
    if empty(frontier) then
        return failure
    node<-pop(frontier)
    if goal(node) return solution(node)
    add node.state to explored
    for each action in problem.actions(node.state) do
        child<-Node(problem,node,action)
        if child.state not in explored or frontier then
            insert(frontier,child)
        else if child.state in frontier(old-child) with higher heuristic-
path-cost then
            replace(frontier,old-child,child)
```

代码展示

和UCS相比，只有节点的定义和算法需要修改，节点增加记录估计值的域`estimate`。

```python
class node:
    """
    定义搜索的节点类
    """
    def __init__(self,pos,cost=0,estimate=0,ancestor=None):
        self.pos=pos
        self.cost=cost
        self.estimate=estimate
        self.ancestor=ancestor
    def __lt__(self, rhs):
        if self.cost + self.estimate == rhs.cost + rhs.estimate:
            return self.estimate<rhs.estimate
        else:
            return self.cost+self.estimate<rhs.cost+rhs.estimate
```

还需要定义几个启发式函数，我使用了以下几种，其中平方距离不满足可采纳性：

```python
def move(pos,direction):
    """
    移动的结果
    """
    return pos[0]+direction.value[0],pos[1]+direction.value[1]

```

```python
def Manhattan(curr,end):
    """
    曼哈顿距离
    """
    return abs(curr[0]-end[0])+abs(curr[1]-end[1])
def Chebyshev(curr,end):
    """
    切比雪夫距离
    """
    return max(abs(curr[0]-end[0]),abs(curr[1]-end[1]))
def Quadratic(curr,end):
    """
    平方
    """
    return math.sqrt((curr[0]-end[0])**2+(curr[1]-end[1])**2)
heuristic_methods=
{'Manhattan':Manhattan,'Chebyshev':Chebyshev,'Quadratic':Quadratic}
```

A*搜索中生成节点时，需要用启发式函数计算估计值传给构造函数。

```python
def A_star(maze, start, end,heuristic=Manhattan):
    spatial_complexity=0
    time_complexity=0

    frontier = []
    visited = [[0] * len(maze[0]) for _ in range(len(maze))]
    visited_list=[]
    heapq.heappush(frontier,node(start,estimate=heuristic(start,end)))
    while True:
        if len(frontier)==0:
            return False

        curr = heapq.heappop(frontier)
        visited_list.append(curr)
        if curr.pos==end:
            print("空间复杂度:{}, 时间复杂度:
{}".format(spatial_complexity,time_complexity))
            return [obj.pos for obj in getRoute(curr)],visited,visited_list
        visited[curr.pos[0]][curr.pos[1]]=1
        time_complexity+=1
        # 遍历可能的动作
        for d in direction:
            InFrontier=False
            new_pos=move(curr.pos,d)
            # 节点在frontier中，若cost更小，更新frontier
            for Node in frontier:
                if Node.pos == new_pos:
                    InFrontier=True
                    if Node.cost>curr.cost+1:
                        Node.cost=curr.cost+1
                        Node.ancestor=curr
                        # heapq.heapify(frontier)
                        break
            # 节点不在frontier中，未被访问过，则放入状态空间(堆)中

            if (not InFrontier) and isValid(maze,new_pos)  and
visited[new_pos[0]][new_pos[1]]==0:
```

```
35
 heapq.heappush(frontier,node(new_pos,cost=curr.cost+1,estimate=heuristic(ne
w_pos,end),ancestor=curr))
36                 spatial_complexity=max(spatial_complexity,len(frontier))
```

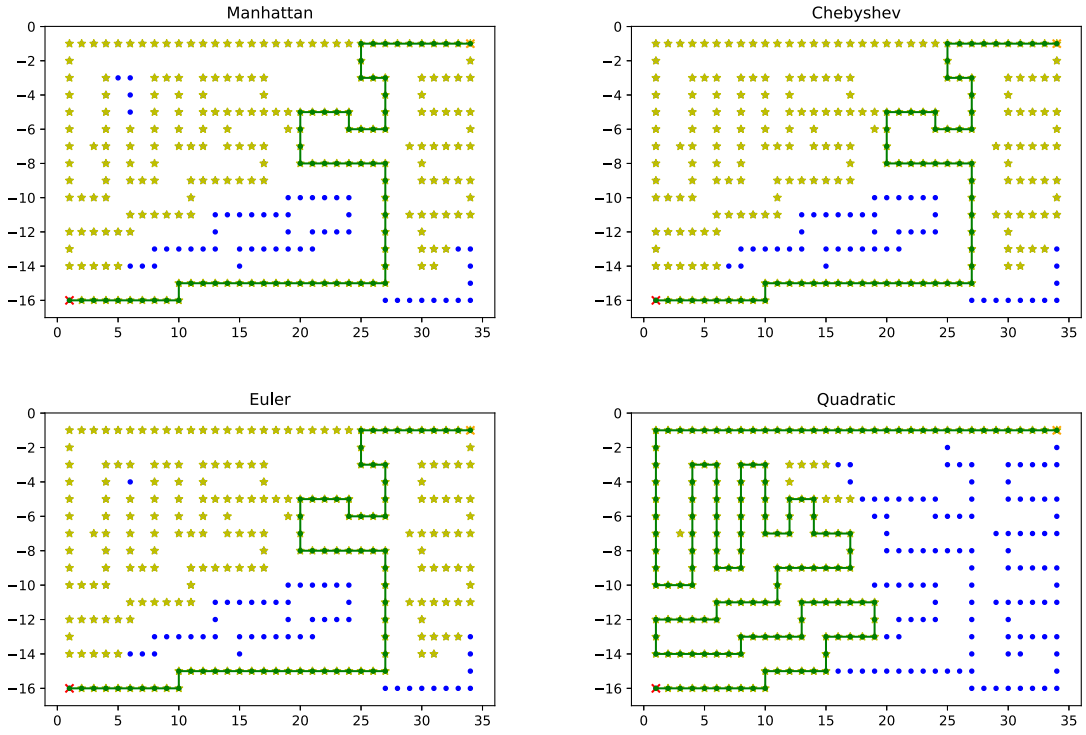
实验结果以及分析

为了说明启发式函数的性质对实验的影响，我也使用了欧氏距离的平方进行了实验。

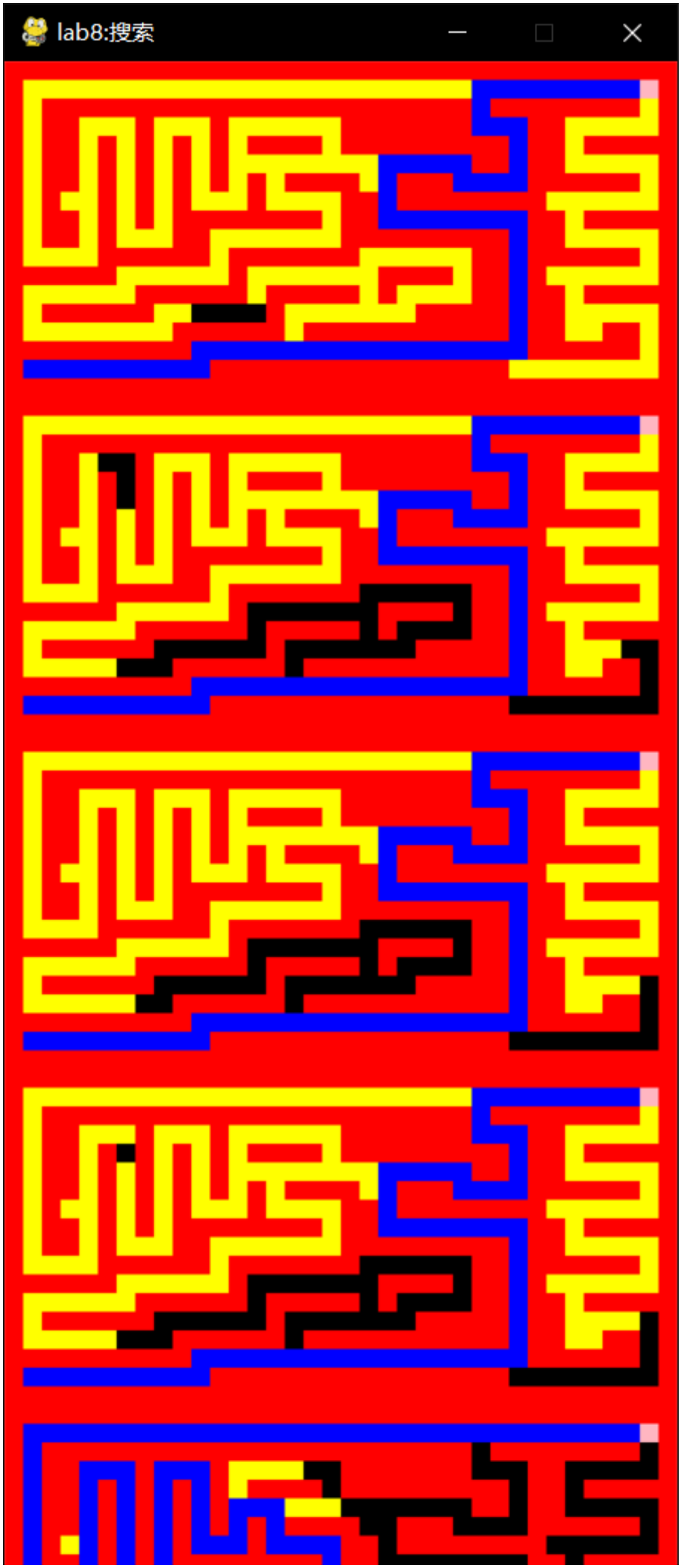
结果如下：

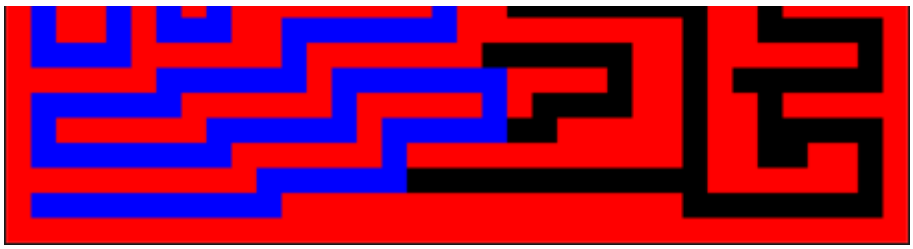
算法	空间复杂度	时间复杂度	完备性	最优性
一致代价搜索	8	269	有	有
A*(曼哈顿距离)	8	220	有	有
A*(切比雪夫距离)	8	226	有	有
A*(欧氏距离)	8	224	有	有
A*(平方距离)	9	161	有	无

用pyplot绘制路径结果如下：



使用pygame库进行可视化，结果更直观（从上到下依次是UCS，A* 曼哈顿，切比雪夫，欧氏，平方距离）：


lab8:搜索



相比一致代价搜索，A*搜索的时间花费明显减少，许多UCS访问的位置都没有访问。空间复杂度则基本没有区别。使用三种不同的启发式函数也没有很大的差别。从数值上来看，曼哈顿距离的时间最短。可以注意到，平方距离得到的结果不是最优的，因为它不满足可采纳性，启发式函数需要一致性和可采纳性才能保证最优性。曼哈顿距离在满足两个特性的前提下数值更大，可能是搜索效率较高的原因之一。

思考题

策略的优缺点

策略	优点	缺点
一致代价搜索	保证完备性，最优性的前提下容易实现	空间复杂度较高，盲目搜索所以效率低
A*搜索	在一致代价搜索的基础上加入启发式函数，搜索速度较快	空间复杂度较高，且搜索速度取决于启发式函数的特点，而且有时候难以找到满足两个性质的启发式函数
迭代加深搜索	空间复杂度低	深度是随迭代逐渐增加的，迭代重新搜索时已访问的点仍会被重新访问。开销不一致时需要满足一定条件才有最优性
IDA*	在迭代加深搜索的基础上加入启发式函数，搜索效率提高	和迭代加深搜索一样，有时候难以找到满足两个性质的启发式函数
双向搜索	搜索深度减半，效率提高	需要维护两个边界集合，空间复杂度高

适用场景

策略	适用场景
一致代价搜索	预估搜索深度较低，或者对空间复杂度要求较低时比较适合，最短和最小问题
A*搜索	有较好的启发式函数，大致知道或者可以确定目标位置时
迭代加深搜索	空间复杂度要求较高的场景
IDA*	空间复杂度要求较高，而且有较好的启发式函数，大致知道或者可以确定目标位置的场景
双向搜索	已知终点和起点的场景