同濟大學

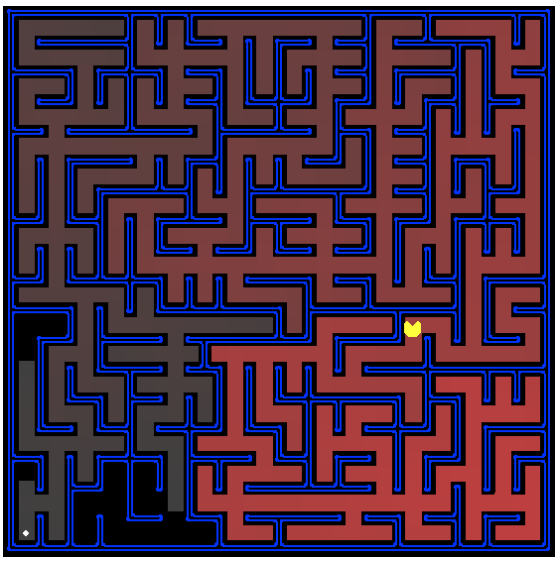
实验报告

实验名称：Pacman Search

学号：2153538

姓名：刘博洋

完成日期：2023年4月3日



1. 项目分析与概述

1.1. 项目总体描述

吃豆人生活在一个蓝色世界里,由弯曲的迷宫构成，迷宫内有分散在各处的食物，有效地收集食物并避开危险，是吃豆人生存的第一要务。

本项目主要是设计程序控制一个吃豆人搜索到达特定目标的路径，收集食物并且避开怪物，并获得更高的分数。本项目有不同的地图资源，同时也有不同的智能Agent，我们将为不同的Agent设计不同搜索算法来解决这些问题。

1.2. 项目已有代码结构及类分析

需要编辑的文件	
search.py	类 Searchproblem，放置所有的搜索算法
searchAgents.py	不同的 Searchproblem 类，不同 Agent 解决不同问题
需要阅读的文件	
pacman.py	GameState 类，pacman 游戏的主要运行文件
game.py	Pacman 游戏运行的底层逻辑 AgentState, Agent, Direction 和 Grid 类.
util.py	编写算法中可以用到的数据结构，如栈、队列和优先队列，以及几个基本的启发式函数，如曼哈顿距离和欧拉距离等等

项目已有代码包括核心的search算法部分和用来解决不同问题的searchAgent部分，同时包括一些便于我们编写算法的工具，除了以上提及的部分之外，同时比较重要的部分还有problem类，类中包含了指向不同问题的接口，以及比较重要的getsuccessor方法和isgoalstate方法用来寻找后继搜索节点和判断是否到达目标，在编写算法前需要对这部分进行仔细阅读。

1.3. 子问题概述及简要分析

1.3.1. Q1：深度优先搜索寻找食物（Finding a Fixed Food Dot using Depth First Search）

应用深度优先搜索，在给定目标坐标和当前坐标的情况下搜索出一条从当前位置到食物坐标处的路径，帮助吃豆人得到食物。深度优先搜索每次深入搜索树的底端，选一条路一直走到底，回溯，遍历所有的子节点，进而达到全局搜索的目的。

1.3.2. Q2：广度优先搜索寻找食物（Breadth First Search）

应用广度优先搜索，在给定目标坐标和当前坐标的情况下搜索出一条从当前位置到食物坐标处的路径。广度优先搜索沿着搜索树的宽度进行遍历，逐层加深，从而遍历整颗搜索树。

1.3.3. Q3：考虑每一步的耗费，运用一致代价搜索寻找食物（uniform-cost search）

虽然BFS可以找到走向食物的最小步数，但是它不一定是最佳的。我们通过改变耗散函数可以对Agent的行动进行一定程度上的控制，例如对充满食物的地方的每一步移动设置比较小的代价进行奖励，而对存在妖怪的危险地区的移动设置较高的代价进行惩罚，这样就可以使agent寻求一条比较安全且合适的路径寻找目标。

1.3.4. Q4：A*搜索（A* search）

不同于前面几种盲目搜索，A*搜索是一种启发式搜索，通过设置启发式函数使Agent的行动更加只能，目标导向更加明确。A*搜索每一步的代价是问题本身的代价加上启发式函数的值，在本项目的情景中，比较好用的启发式函数可能是曼哈顿距离或者欧拉距离，我们只需要在一致代价搜索的基础上将耗散函数进行更改即可。

1.3.5 Q5：找到并遍历所有角落

此问题的目标是找到一个最短路径遍历地图的四个角落，我们采用A*搜索，但是此次要自己编写移动的状态判断，如判断是否撞墙等。一旦访问一个角落，就把它标记为visited，并把访问过的角落数++，最后的截止状态是访问过的角落数等于4

1.3.6. Q6：用启发式函数搜索所有角落

此问题的目标是找到一个最短路径遍历地图的四个角落，我们同样采用A*搜索，不过这次要自己设置启发式函数，要根据具体的问题设置比较一致和完备的启发式函数，就可以显著降低搜索的成本和拓展的节点数。

1.3.7. Q7：找到所有食物

此问题要求控制Agent找到所有的食物，同时要设计一个启发式函数使拓展的节点尽可能少。

1.3.8 Q8：设计一个每次都会吃最近的食物的Agent

有时候很难达到完美的算法，因此本题是要求设计每次吃最近的食物，于是我们可以调用mazedistance方法，获得两个点在地图内的距离并设计启发式函数进行搜索。

2. 算法设计与实现

2.1.1. Q1：深度优先搜索寻找食物（Finding a Fixed Food Dot using Depth First Search）

对于DFS问题，我们采用栈这一数据结构，建立两个表，一个为open表，为栈，存放当前没有拓展的节点，close表存放已经访问过的节点

每一次搜索的步骤如下：

1. 把初始节点S0放入open表
2. 如果open表为空，那么问题无解，退出
3. 从open栈中弹出一个节点p，访问并放入close表
4. 考察节点p是否是目标节点，如果是则退出，否则继续搜索
5. 若节点n不可拓展，回到第2步，
6. 否则拓展节点n，将其子节点压栈，然后再重复2-6步操作，直到求得问题的解。

```
def depthFirstSearch(problem: SearchProblem):

from util import Stack
    from game import Directions
   close = []      # close 表存放已经访问过的节点
    open = Stack()  # open 表存放未拓展的节点
    action=[]     #action 存放访问每一步的方向
    open.push((problem.getStartState(),[])) # 把初始节点放入 open 表 搜索树的节点结构为（当前状态，[actions]从初始状态到达当前状态经过的动作集合）
    while open.isEmpty()==False :
        current,action= open.pop()  # open 表中弹出一个节点
        if problem.isGoalState(current) :   # 判断是否到达目标
            return action
        if current not in close:          # 如果没有访问过
            expand = problem.getSuccessors(current)  # 拓展后继节点
            close.append(current)             # 把 current 加入 close 表
            for nextnode,nextaction,cost in expand:
                newaction=action+[nextaction]
              open.push((nextnode,newaction))
```

深度优先搜索优点是比较简单，容易实现，只要储存当前路径，就比较有希望在更浅处找到问题的解，求解速度相对较快；但是一旦有环，可能会陷入死循环，并且如果不限制深度的话可能会导致大量的资源浪费，并且找到的可能不是最优解，还可能会出现栈溢出问题。如果目标节点更靠近叶子节点，DFS会更好用，对于不要求最优解的情况，DFS内存消耗少，速度较快，较为适用。

2.1.2. Q2：广度优先搜索寻找食物（Breadth First Search）

对于DFS问题，我们采用的是队列这一数据结构，建立两个表，一个为open表，为队列，存放当前没有拓展的节点，close表存放已经访问过的节点

每一次搜索的步骤如下：

1. 把初始节点S0放入open表

2. 如果open表为空，那么问题无解，退出

3. 一个节点p从open队列中出队，访问并放入close表

4. 考察节点p是否是目标节点，如果是则退出，否则继续搜索

5. 若节点n不可拓展，回到第2步，

6. 否则拓展节点n，将其子节点入队，然后再重复2-6步操作，直到求得问题的解。

```
    def breadthFirstSearch(problem: SearchProblem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    from util import Queue
    from game import Directions
    close = []  # close 表存放已经访问过的节点
    open = Queue()  # open 表存放未拓展的节点
    open.push((problem.getStartState(), []))  # 把初始节点放入 open 表 搜索树的节点结构为（当前状态，[actions]从初始状态到达当前状态经过的动作集合）
    while open.isEmpty() == False:
        current, action = open.pop()  # open 表中弹出一个节点
        if problem.isGoalState(current):  # 判断是否到达目标
            return action
        if current not in close:  # 如果没有访问过
            expand = problem.getSuccessors(current)  # 拓展后继节点
            close.append(current)  # 把 current 加入 close 表
            for nextnode, nextaction, cost in expand:
                newaction = action + [nextaction]
                open.push((nextnode, newaction))
```

广度优先搜索从距离起点最近的节点开始搜索，保证找到最短路径，它按照层次顺序搜索，所以可以找到最优解，因为使用队列而不是栈，所以不会 Stack Overflow

缺点： 需要维护一个队列来存储每个节点的邻居节点，所以空间复杂度较高 当图非常大时，BFS可能无法完成搜索

2.1.3. Q3：考虑每一步的耗费，运用一致代价搜索寻找食物（uniform-cost search）

一致代价搜索其实本质上就是广度优先搜索加上最优代价的思想，广度优先搜索每一步的代价为1，而一致代价搜索通过改变代价函数鼓励Agent找到离食物更近的路。

为此我们只需要把open表改为一个fringe表，使其为一个优先队列，进队之后自动按当前代价排序，下一个出队的一定是当前所有拓展节点中代价最小的那个，关键在于调用函数getcostofaction，获得每一步的最小代价。这样就可以实现一致代价搜索。

```
    def uniformCostSearch(problem: SearchProblem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"

    visited = []  # visited 表存放已经访问过的节点
    fringe = util.PriorityQueue()  # 优先队列 fringe 存放未拓展的节点，按耗散排序
    # 把初始节点放入 fringe 表 搜索树的节点结构为（当前状态，[actions]从初始状态到达当前状态经过的动作集合，当前 cost）
    fringe.push((problem.getStartState(), []), 0)

    while fringe.isEmpty() == False:
        current , action = fringe.pop()
        if problem.isGoalState(current):
            return action
        if current not in visited:
           visited.append(current)
           expand=problem.getSuccessors(current)
           for nextnode,nextaction,cost in expand:
               newaction=action+[nextaction]
               newcost=problem.getCostOfActions(newaction)  # 每一步都记录访问耗散最小的节点
               fringe.push((nextnode,newaction),newcost)
    util.raiseNotDefined()
```

一致代价搜索在广度优先搜索的基础上考虑了代价，用改变代价函数来促使Agent改进他的行为，并且一旦检测到更优的新路径，就会丢弃老路径。一致代价搜索时最优的，但是它对路径的步数并不关心，只关心总代价，所以如果存在零代价行动就会出现死循环，而且一致代价搜索具有较高的时间复杂度，因为要想搜索到代价小的步骤，必须搜索很大的搜索树，找到所有节点确定其为最小代价时才会终止。

一致代价搜索适用于每一步代价不等的情况下，亦或是需要对搜索行为进行一定控制的情况，而不

太适用于存在零代价行为的问题。

2.1.4. Q4: A*搜索（A* search）

A*算法是一种启发式算法，其每一步的代价不仅包括实际上的代价，还包括一个启发式函数的值，在方格地图中，通常把启发式函数设置为当前节点到终点的曼哈顿距离

```
    def Manhattan_distance (node) =
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx + dy)
```

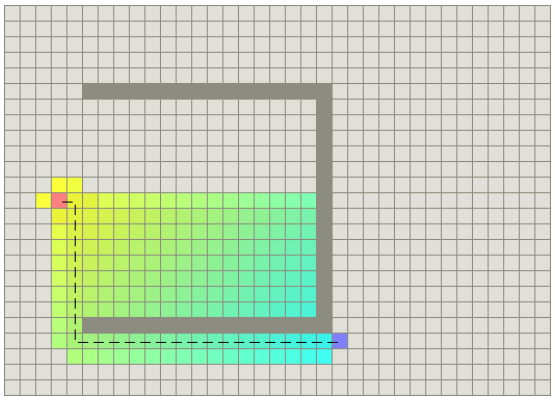


图 1 用 A*算法寻路

A*算法在没有障碍物的情况下的搜索结果与 BFS 一样，而在有障碍物的情况下搜索情况与迪杰斯特拉算法类似，这是因为 A*算法一方面考虑实际距离和代价，另一方面加入启发式函数这一预估代价。

在本项目中，通过 A*算法搜索路径寻找食物，我们只需要把一致代价搜索中的代价函数改为当前代价加上启发式函数的值即可，同样也是通过优先队列每次将代价最小的节点出队进行访问，最终得到目标路径。

```
    def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic
first."""
    "*** YOUR CODE HERE ***"
    visited = []  # visited 表存放已经访问过的节点
    fringe = util.PriorityQueue()  # 优先队列 fringe 存放未拓展的节点，按耗散排序
    # 把初始节点放入 fringe 表 搜索树的节点结构为（当前状态，[actions]从初始状态到达当前
状态经过的动作集合，当前 cost）
    fringe.push((problem.getStartState(), []), 0)

    while fringe.isEmpty() == False:
```

```
        current, action = fringe.pop()
        if problem.isGoalState(current):
            return action
        if current not in visited: # 如果没有访问过
            visited.append(current)
            expand = problem.getSuccessors(current)
            for nextnode, nextaction, cost in expand:
                newaction = action + [nextaction]

newcost=problem.getCostOfActions(newaction)+heuristic(nextnode,problem) # 每一步代价等于行动本身代价加上启发式函数的值
                fringe.push((nextnode, newaction), newcost) # 最小代价节点入队
```

A*算法是目前人们解决实际问题中用的较多的方法，A*算法的效率和优势主要取决于其启发式函数的设计，如果启发式函数设计的越符合问题的实际情况，那么搜索的效率就会越高。并且，在给定一致的启发式函数的情况下，A*算法一定是效率最优的。

A*算法的缺陷一方面在于状态空间中存在多个目标状态时，或者是比较接近最佳目标状态时，搜索有可能因为启发式函数而误入歧途，带来不小的额外代价，而由于较大的时间复杂度，在遇见复杂问题时，用 A*算法可能难以找到一条最佳路径，而是只能满足局部最优解；同时，由于 A*算法在内存中保留了所有已经生成的节点，所以在面对大规模的搜索问题时，A*算法很容易耗尽内存。

2.1.5. Q5：找到并遍历所有角落

这一部分要求遍历所有角落，要补全getStartState，isGoalState，getSuccessors等几个函数，首先本题要自己设置一个好用的State（状态）来记录问题，由于要遍历四个角落，所以可以把state设置为一个元组（当前位置坐标，已经经过的角落数组），据此可以编写getStartState函数如下：

```
    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state space)
        """
        "*** YOUR CODE HERE ***"
        return(self.startingPosition,[]) #返回初始位置及经过的角落数
        #util.raiseNotDefined()
```

而如果要判断是否到达目标状态，可以简单判断为已经经过的角落数组的长度是否为4，因此可以编写函数isGoalState如下：

```
    def isGoalState(self, state: Any):
        """
        Returns whether this search state is a goal state of the problem.
```

```
        """
        "*** YOUR CODE HERE ***"
        des=state[0]
        cornervisit=state[1]
        return len(cornervisit)==4
```

而每一次寻找后继节点中，一个比较关键的内容时判断行动是否合法，即是否撞墙，如果没有撞墙则可以移动并且每走一步都要判断当前位置是否经过了角落点，如果经过了角落则要把当前角落加入已经经过的角落表中予以记录，重复以上操作，直到把一个点的所有后继都加入后继列表。

```
    def getSuccessors(self, state: Any):
        """
        Returns successor states, the actions they require, and a cost of 1.
        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
Directions.WEST]:
            # Add a successor state to the successor list if the action is
legal
            # Here's a code snippet for figuring out whether a new position
hits a wall:
            #   x,y = currentPosition
            #   dx, dy = Actions.directionToVector(action)
            #   nextx, nexty = int(x + dx), int(y + dy)
            #   hitsWall = self.walls[nextx][nexty]

            "*** YOUR CODE HERE ***"
            x,y=state[0] #当前坐标
            cornervisit=state[1] #当前已经访问的角落数组
            dx,dy=Actions.directionToVector(action) #下一步行动
            nextx,nexty= int(x+dx),int(y+dy)
            nextnode=(nextx,nexty) # 下一个点
            hitswall=self.walls[nextx][nexty]
            if not hitswall: #没有撞墙则可以继续移动
                cornerlist=list(cornervisit)
                if nextnode in self.corners: #判断是否经过了角落点
                    if nextnode not in cornerlist:
                        cornerlist.append(nextnode)  #如果当前在角落点且该角落没有访问
过，加入已经访问的角落列表
                successor = ((nextnode, cornerlist), action, 1) #successor是一
个三元组（state，action，cost）
                successors.append(successor) #把当前后继加入后继列表
        self._expanded += 1 # DO NOT CHANGE
        return successors
```

这一部分的代码编写并不涉及算法部分，而是注重与对已有代码、函数和类的理解和阅读，对于游戏逻辑的掌握，只需要按部就班考虑周全即可。

2.1.6. Q6：用启发式函数搜索所有角落

考虑到要设计一个效率较高的算法找到一条相对好的路径，本题要求自己设计启发式函数来提高搜索小了，降低搜索拓展的节点数。

关于启发式函数，我们需要注意的是其可采纳性和一致性，可采纳性是指这个启发式从不会过高估

计到达目标的代价，它认为解决问题的代价比实际代价小；而一致性比可采纳性要强，它要求在图搜索算法中，对于每个节点n和通过任意行动生成的后继节点n’，从n到达目标的估计代价不大于从n到n’的单步代价加上n’到目标的估计代价之和。

在本项目中，由于是地图，一个比较容易想到的点是采用两点之间的曼哈顿距离，但是对于有多个食物目标的问题来说，要想遍历所有的食物，我们最好能够考虑一条遍历所有食物的顺序，其距离以曼哈顿距离来衡量，如现在有ABC三个食物，我们分别求出以顺序ABC，ACB，BAC，BCA，CAB，CBA来访问所有食物所需的曼哈顿距离，从中选出最小的那个，这个距离一定是小于实际距离的，并且也满足一致性，是一个比较好的启发式函数。

```
    def minmanhattan(corners, pos):
    # 遍历所有 corner 的豆子，选择最小的曼哈顿距离作为启发函数
    if len(corners) == 0:
        return 0

    hn = []
    for loc in corners:
        dis = abs(loc[0] - pos[0]) + abs(loc[1] - pos[1])  + minmanhattan([c
for c in corners if c != loc], loc)  # 分别计算每个遍历顺序的曼哈顿距离
        hn.append(dis)

    return min(hn) # 返回最小的距离
```

而cornersHeuristic函数则调用该启发式函数，编写如下即可：

```
    def cornersHeuristic(state: Any, problem: CornersProblem):
    """
    A heuristic for the CornersProblem that you defined.

      state:   The current search state
               (a data structure you chose in your search problem)

      problem: The CornersProblem instance for this layout.

    This function should always return a number that is a lower bound on the
    shortest path from the state to a goal of the problem; i.e.  it should be
    admissible (as well as consistent).
    """
    corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid
(game.py)

    "*** YOUR CODE HERE ***"
    currentpos = state[0] # 返回当前位置
    unvisitedcorner = [i for i in corners if i not in state[1]]
    hn = minmanhattan(unvisitedcorner, currentpos)  # 计算所有未经过的角落的到现在
距离的最小曼哈顿值，一定比实际距离小

    return hn
```

2.3.7. Q7：找到所有食物

此问题要设计一个启发式函数使Agent遍历所有食物并使拓展的节点尽可能少，考虑一致性和可采

纳性，由于是要找到地图中所有食物，所以可以把算法的启发式函数用一个已有的函数Mazedistance来求解，由于要满足一致性，即到终点的代价要小于现在状态到中间状态和中间状态到最终状态的和，所以必须选所有地图距离中最大的一个值作为启发函数，当Agent到所有食物的最大距离变小时，代表其越接近完成目标，Agent就会一直朝着这个方向前进，这样可以显著减少拓展的节点。

```
    def foodHeuristic(state: Tuple[Tuple, List[List]], problem:
FoodSearchProblem):
    """
    Your heuristic for the FoodSearchProblem goes here.

    This heuristic must be consistent to ensure correctness.  First, try to
come
    up with an admissible heuristic; almost all admissible heuristics will be
    consistent as well.

    """
    pos, foodGrid = state
    "*** YOUR CODE HERE ***"
    pos,grid=state
    food_cor=foodGrid.asList()
    if not food_cor:
        return 0
    val=[]
    for tmp in food_cor:
        distance=mazeDistance(tmp,pos,problem.startingGameState)  # 当前点到所有
食物点中地图步数最大的距离
        val.append(distance)
    return max(val)
```

1.3.8 Q8：设计一个每次都会吃最近的食物的Agent

由于A*算法不一定每次都能返回一个最佳路径，我们有时候可以退而求其次，本题要求控制Agent每次都吃最近的食物，而这种情况下需要补全一个找到最近食物的函数和一个判断结束状态的函数。本问题的结束状态可以如下方式表达：

```
    def isGoalState(self, state: Tuple[int, int]):
        """
        The state is Pacman's position. Fill this in with a goal test that
will
        complete the problem definition.
        """
        x,y = state

        "*** YOUR CODE HERE ***"
        foodloc=self.food.asList()
        distance=[]
        for f in foodloc:
            distance.append(util.manhattanDistance(state,f))
        mindis=min(distance)
        return mindis==0
        util.raiseNotDefined()
```

即找到当前节点到所有节点的曼哈顿距离，然后找出其中最小的那个，如果最小的那个为0，说明此次搜索实现了找到当前最近的节点。而对于找路径的算法，我选择的是之前已经编写好的A*算法，于是

只需要补充findPathToClosestDot的返回值即可

```
    def findPathToClosestDot(self, gameState: pacman.GameState):
        """
        Returns a path (a list of actions) to the closest dot, starting from
        gameState.
        """
        # Here are some useful elements of the startState
        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)

        "*** YOUR CODE HERE ***"
        return search.aStarSearch(problem)
```

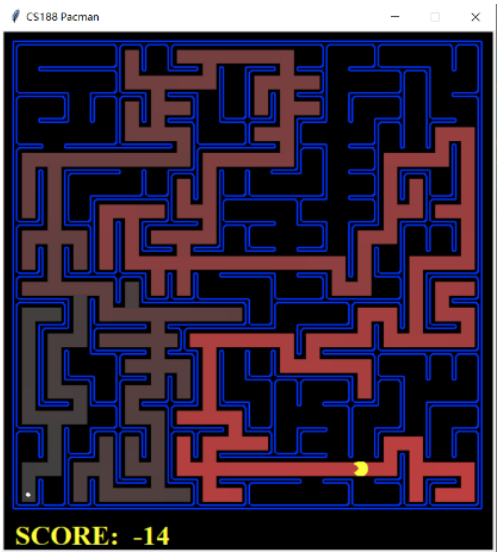
3. 实验结果

Q1：DFS

可以看到最终的路径是一条比较深的路径，说明DFS在较深处找到了目标，拓展的节点并不多

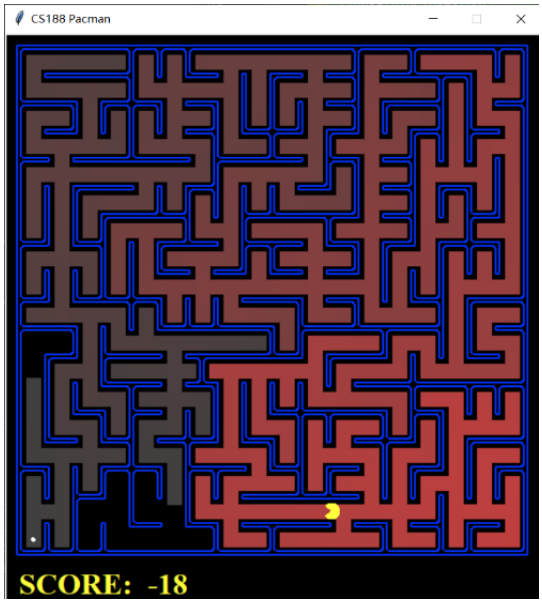
```
Question q1
===========
*** PASS: test_cases\q1\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'D', 'C']
*** PASS: test_cases\q1\graph_bfs_vs_dfs.test
Question q1: 3/3
-----------------
Total: 3/3
```

```
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 390
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```



Q2：BFS

```
Finished at 22:04:25

Provisional grades
==================
Question q2: 3/3
------------------
Total: 3/3
```




```
Question q2
===========
*** PASS: test_cases\q2\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q2\graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases\q2\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q2\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases\q2\pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269

### Question q2: 3/3 ###
```

可以看到BFS拓展的节点更多更广，拓展到比较深的地方才找到正确路径

Q3：UCS

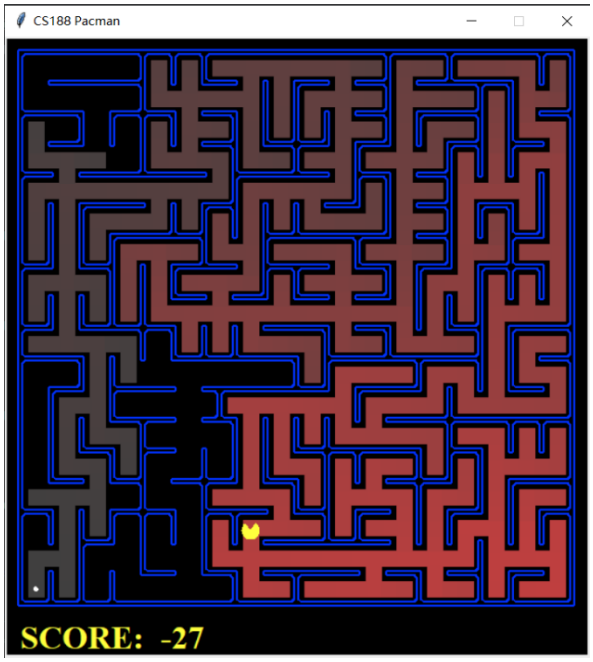
```
Question q3
===========
*** PASS: test_cases\q3\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q3\graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases\q3\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q3\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases\q3\ucs_0_graph.test
***     solution:               ['Right', 'Down', 'Down']
***     nodes expanded:         260
*** PASS: test_cases\q3\ucs_3_problemW.test
***     pacman layout:          mediumMaze
***     solution length: 152
***     nodes expanded:         173
*** PASS: test_cases\q3\ucs_4_testSearch.test
***     pacman layout:          testSearch
***     solution length: 7
***     nodes expanded:         14
*** PASS: test_cases\q3\ucs_5_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']

### Question q3: 3/3 ###
```

```
Finished at 22:07:28

Provisional grades
==================
Question q3: 3/3
------------------
Total: 3/3
```

可以看到UCS通过改变代价函数促使Agent尽量避开妖怪出没的地区而走向食物丰富的地区。

Q4：A*搜索

```
Question q4
===========
*** PASS: test_cases\q4\astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q4\astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
*** PASS: test_cases\q4\astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         221
*** PASS: test_cases\q4\astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q4\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q4\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

### Question q4: 3/3 ###
```




```
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 seconds
Search nodes expanded: 549
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

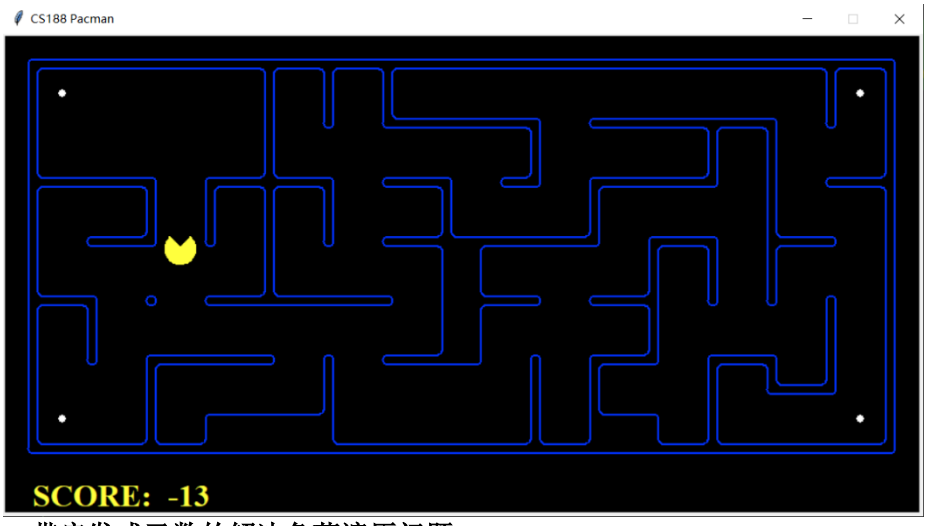
可以观察到A*搜索相比于BFS，通过增加启发式函数减少了很多不必要的节点拓展，提高了搜索效率，使行动目标更加明确。

Q5：找到所有角落

```
Question q5
===========
*** PASS: test_cases\q5\corner_tiny_corner.test
***     pacman layout:          tinyCorner
***     solution length:                28

### Question q5: 3/3 ###


Finished at 22:14:39

Provisional grades
==================
Question q2: 3/3
Question q5: 3/3
------------------
Total: 6/6
```

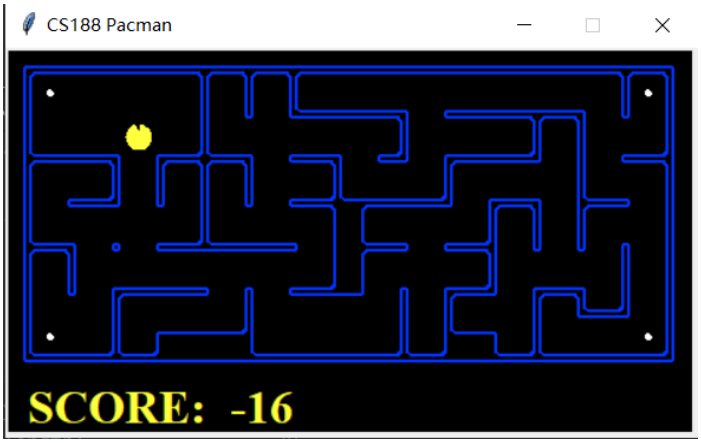
```
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.2 seconds
Search nodes expanded: 2448
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Agent可以以106步的代价完成以下地图中遍历所有角落的任务，但是拓展了2448个节点。



Q6：带启发式函数的解决角落遍历问题

```
Question q6
===========
*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
path: ['North', 'East', 'East', 'East', 'East', 'North', 'North', 'West', 'West', 'West', 'West', 'North', 'Nort
t', 'South', 'South', 'South', 'South', 'South', 'South', 'West', 'West', 'South', 'South', 'South', 'West', 'We
, 'East', 'East', 'East', 'East', 'North', 'North', 'East', 'East', 'North', 'North', 'East', 'East', 'North', '
South', 'South', 'South', 'South', 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'West', 'West'
path length: 106
*** PASS: Heuristic resulted in expansion of 950 nodes

### Question q6: 3/3 ###
```



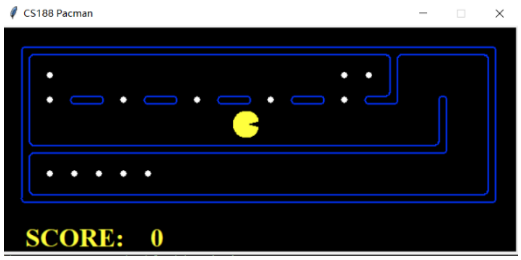

```
Path found with total cost of 106 in 0.1 seconds
Search nodes expanded: 950
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

同样的问题，增加启发式函数后，完成任务只拓展了950个节点，显著提高了搜索效率。

Q7：吃掉所有食物

```
Question q7
===========
*** PASS: test_cases\q7\food_heuristic_1.test
*** PASS: test_cases\q7\food_heuristic_10.test
*** PASS: test_cases\q7\food_heuristic_11.test
*** PASS: test_cases\q7\food_heuristic_12.test
*** PASS: test_cases\q7\food_heuristic_13.test
*** PASS: test_cases\q7\food_heuristic_14.test
*** PASS: test_cases\q7\food_heuristic_15.test
*** PASS: test_cases\q7\food_heuristic_16.test
*** PASS: test_cases\q7\food_heuristic_17.test
*** PASS: test_cases\q7\food_heuristic_2.test
*** PASS: test_cases\q7\food_heuristic_3.test
*** PASS: test_cases\q7\food_heuristic_4.test
*** PASS: test_cases\q7\food_heuristic_5.test
*** PASS: test_cases\q7\food_heuristic_6.test
*** PASS: test_cases\q7\food_heuristic_7.test
*** PASS: test_cases\q7\food_heuristic_8.test
*** PASS: test_cases\q7\food_heuristic_9.test
*** PASS: test_cases\q7\food_heuristic_grade_tricky.test
***     expanded nodes: 4137
***     thresholds: [15000, 12000, 9000, 7000]

### Question q7: 5/4 ###
```

```
Path found with total cost of 60 in 12.1 seconds
Search nodes expanded: 4137
Pacman emerges victorious! Score: 570
Average Score: 570.0
Scores:        570.0
Win Rate:      1/1 (1.00)
Record:        Win
```



可以看到搜索了4000个节点就完成了吃掉所有食物的寻路任务，远少于题目要求的数量7000以内

Q8：次优搜索


```
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with cost 350.
Pacman emerges victorious! Score: 2360
Average Score: 2360.0
Scores:        2360.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
Provisional grades
==================
Question q8: 3/3
------------------
Total: 3/3
```

可以看到A*算法没有找到一条最好的路径，而是把左边食物全都吃完了还在右边剩下一个，不过已经算是能够完成任务，基本符合我们的要求。

最终得分25/23

4. 总结与分析

4.1 算法分析

搜索是Agent通过找到一组行动序列达到目标的过程，数年来，人类开发出多种不同的搜索算法，不断提高搜索的效率、准确性和速度，而这些算法可以分为无信息搜索和有信息搜索两大类，无信息搜索包括宽度优先搜索BFS、统一代价搜索UCS、深度优先搜索DFS、深度受限搜索DLS、迭代加深搜索IDS、双向搜索BS，有信息搜索算法包括贪心搜索、A* 搜索、迭代最佳优先搜索RBFS。

通常一个搜索算法用三个指标来进行评估：

完备性：当问题有解时，这个算法是否可以保证找到解？

最优性：当问题有解时，这个算法是否可以找到使得路径代价最小的解？

复杂性：时空复杂度。

经过学习和实验，我们也可以对以上算法进行一个简单的总结和评价

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes^{a}	Yes^{a,b}	No	No	Yes^{a}	Yes^{a,d}
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes^{c}	Yes	No	No	Yes^{c}	Yes^{c,d}

首先是无信息搜索部分，可以看到对于 DFS 不是完备的，因为可能在遇到无穷深的问题时无法找到解,所以 DFS 一般用于图搜索而不太适合用于大规模的树搜索。

BFS 总是拓展搜索树中深度最浅的节点，算法是完备的，在单位代价下是最优的；而一致代价搜索拓展路径代价最小的节点，对于一般性步骤代价而言算法也是最优的。

而关于 BFS，UCS 和 A*算法的联系，我也有了一些自己的感悟。

UCS 相对于 BFS 来说，实际上就是维护了一个优先队列，通过这个优先队列保证达到目标节点的时候路径一定是最短的。而 UCS 的搜索是像圆一样从一个点蔓延，直到找到了目标节点；而 A*由于有着自己的启发式函数，则是朝着目标节点的方向蔓延，尽管做不到只走一条路径，但也高效的多，形状类似于一个椭圆。

A*搜索时完备的也是最优的，如果启发式函数是可采纳的或者一致的，但是有着很高的空间复杂度。

而关于启发式函数的设计，也就是我们要让这个圆变得越来越扁平的过程，如果启发效果越好，那么搜索的形状就更加具有导向性，更容易到达目标节点，代价也就更小。好的启发式函数还可以通过松弛问题来得到，将子问题的代价记录在模式数据库中，或者对问题进行经验学习来得到。

4.2 项目总结

遇到的困难及解决：

在完成此项目过程中，在代码方面遇到了一些困难，主要还是因为python基础知识比较薄弱导致的，例如我一开始不太明白可以直接用两个变量作为一个元组进行返回，或者是对于一个state型的元组，用state[0]和state[1]去访问其内容，并且关于problem类的继承我一开始也不是特别理解。这就导致我在项目开始看已有代码的过程中用掉了大量时间，不过也得到了相应的成效，花费大量时间把项目要求仔细阅读的代码逻辑理顺后，再补全算法函数就比较简单了。

还有一个比较难的地方是在问题7设计启发式函数的地方，如果仅仅只是用两点之间的曼哈顿距离的话得到的效果不尽人意，这更像是一个头脑风暴的过程，根据问题的结构和形式，最终我是采用了到所有食物的最大距离这一指标，因为不仅要考虑可采纳性，而且要考虑一致性，而且随着到所有食物最大距离的减小，就代表着任务越接近完成，这是一种类似于奖励机制的设定。

项目总结与收获：

通过这次项目，我比较好的完成了项目要求任务，由于对python基础语法不够熟悉，项目实现过程中遇到了不少困难，但是经过仔细思考和反复斟酌，最终还是能够克服这些困难。经过8个问题的循序渐进，我不但增长了我对人工智能搜索问题的理解，而且夯实了我的python程序设计基础，是一次非常有意义的实践。