哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验报告

开课学期：大三上学期

课程名称：人工智能

实验名称：搜索策略

实验性质：设计

实验时间：2019.10.21 地点：T2109 实验台号 21

学生专业：计算机类

学生学号：SZ170110311

学生姓名：廖思瑀

评阅教师：

报告成绩：

实验与创新实践教育中心印制
2019年10月

一． 简介/问题描述

1.1 待解决问题的解释

吃豆人：

1)问题 1

利用深度优先搜索算法找到一个特定位置的豆，帮助吃豆人规划路线，找到特定位置的豆

2)问题 2

利用宽度优先搜索算法找到一个特定位置的豆，帮助吃豆人规划路线，找到特定位置的豆

3)问题 3

利用代价一致算法，根据路径中的不同代价，帮助吃豆人规划路线，找到代价最优的路径并吃到豆子

4)问题 4

利用 A*搜索算法，利用曼哈顿距离作为启发函数，帮助吃豆人规划路线，找到特定位置的豆

5)问题 5

在角落迷宫的四个角上面有四个豆，找到一条最短的路径能访问吃到四个角落中的豆子

6)问题 6

找到合适的启发式函数，使得吃豆人能够访问迷宫图中的每一个角落，同时不会发生“穿墙”

7)问题 7

构造合适的启发函数，用尽可能少的步数吃掉所有的豆子

8)问题 8

进行算法优化，定义一个优先吃最近的豆子的函数，从而提高搜索速度

1.2 问题的形式化描述

吃豆人：

定义谓词：吃豆人位置、豆子位置、迷宫道路信息、墙面信息

定义相应的算法操作：深度优先搜索、广度优先搜索、一致代价搜索算法、A*算法、曼哈顿距离算法等

初始状态：吃豆人在地图中随机的某个位置，地图中其它的某些位置上有豆子

终止状态：吃豆人吃掉地图中所有的豆子，游戏终止

1.3 解决方案介绍（原理）

利用一系列的搜索策略：广度优先搜索、深度优先搜索、代价一致搜索算法、A*算法 ，同时利用各类启发式函数：如曼哈顿距离函数等，实现对：次优先搜索、吃到所有的“豆”、角落问题

二． 算法介绍

2.1 所用方法的一般介绍

1)问题 1

利用深度优先搜索算法，深度优先搜索属于图算法的一种，英文缩写为 DFS（Depth First Search.）其过程简要来说是对每一个可能的分支路径深入到不能再深入为止，而且每个节点只能访问一次。

2)问题 2

利用广度优先搜索算法，广度优先搜索算法（Breadth-First Search，BFS）是一种盲目搜寻法，目的是系统地展开并检查图中的所有节点，以找寻结果。换句话说，它并不考虑结果的可能位置，彻底地搜索整张图，直到找到结果为止。BFS 并不使用经验法则算法。

3)问题 3

利用一致代价算法，在 BFS 的基础上，一致代价搜索不在扩展深度最浅的节点，而是通过比较路径消耗，并选择当前代价最小的节点进行扩展，因此可以保证无论每一步代价是否一致，都能够找到最优解。

4)问题 4

A*算法，A*（A-Star)算法是一种静态路网中求解最短路径最有效的直接搜索方法，也是解决许多搜索问题的有效算法。算法中的距离估

算值与实际值越接近，最终搜索速度越快。

5)问题 5

利用定义好的广度优先搜索算法，广度优先搜索算法（Breadth-First Search，BFS）是一种盲目搜寻法，目的是系统地展开并检查图中的所有节点，以找寻结果。换句话说，它并不考虑结果的可能位置，彻底地搜索整张图，直到找到结果为止。BFS 并不使用经验法则算法。

6)问题 6

利用曼哈顿距离作为启发式寻路算法，曼哈顿距离：两点在南北方向上的距离加上东西方向上的距离，就曼哈顿距离的概念来说，只能上、下、左、右四个方向进行移动，而且两点之间的曼哈顿距离是两点之间的最短距离（在只能向上、下、左、右四个方向进行移动的前提下）。

7)问题 7

A*算法，A*（A-Star)算法是一种静态路网中求解最短路径最有效的直接搜索方法，也是解决许多搜索问题的有效算法。算法中的距离估算值与实际值越接近，最终搜索速度越快。

8)问题 8

递归利用 A*算法，利用完一次 A*算法后，将路径终点作为新的路径源点，再进行 A*算法搜索，A*（A-Star)算法是一种静态路网中求解最短路径最有效的直接搜索方法，也是许多其他问题的常用启发式算法。

2.2 算法伪代码

问题一：深度优先搜索：

（1）初始化一个栈，并将这个顶点入栈

（2）判断栈是否为空

若栈非空{

访问栈顶点（记为 p);

在全局数组中将这个顶点标记为已访问；

将栈顶顶点出栈

遍历 p 的所有邻接顶点{

若该顶点没被访问过入栈}}

问题二：广度优先搜索

（1）初始化一个队列，并将这个顶点入队列

（2）判断队列是否为空

若队列非空{

访问队头顶点 s

标记 s 为已遍历；

出队 pop()

遍历 p 的所有邻接顶点{

若该顶点没被访问过，入队}}

问题三：一致代价搜索算法

（1）如果边缘为空，则返回失败。操作：EMPTY?(frontier)

（2）否则从边缘中选择一个叶子节点。操作：POP(frontier)

（3）遍历叶子节点的所有动作

1）每个动作产生子节点

2）如果子节点的状态不在探索集或者边缘，则插入到边缘集合。操作：INSERT(child, frontier)

3）否则如果边缘集合中如果存在此状态且有更高的路径消耗，则用子节点替代边缘集合中的状态

问题四：A*算法

（1）将起始点加入 open 表

当 open 表不为空时：

寻找 open 表中 f 值最小的点 current

它是终止点，则找到结果，程序结束。

否则，Open 表移出 current，对 current 表中的每一个临近点

若它不可走或在 close 表中，略过

若它不在 open 表中，加入。

若它在 open 表中，计算 g 值，若 g 值更小，替换其父节点为 current，更新它的 g 值。

若 open 表为空，则路径不存在。

问题七：吃掉所有的豆子

（1）遍历地图找到所有有豆子的位置，并进行存储为 dot

（2）遍历 dot,利用曼哈顿距离求解源点离所有豆子的 distance

（3）求取最大 distance 值，将其作为第一个搜索点

（4）不断循环直到吃掉所有豆子

问题八：次优先搜索

（1）找到所有有豆子的节点，并进行存储 dot

（2）将豆子起始点作为源点

（3）利用 A*算法，找到离源点最近的豆子，并吃掉该豆子

（4）重复（2）（3）步骤，直到吃掉所有的豆子

源代码：

问题一：深度优先搜索

```python
def depthFirstSearch(problem):
    s = problem.getStartState()      #初始节点
    closed = []     #建立一个closed表, 置为空
    open = util.Stack()
    open.push((s,[]))    #将初始节点放入open表 (栈)
    while not open.isEmpty():   #检查open表是否空
        cnode,action = open.pop()
        if problem.isGoalState(cnode):      #到达目标节点, 退出
            return action
        if cnode not in closed:
            closed.append(cnode)
            successor = problem.getSuccessors(cnode)     #将子节点放入open表
            for location,direction,cost in successor:
                if(location not in closed):
                    open.push((location,action+[direction]))
    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()
```

问题二：广度优先搜索

```python
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    s = problem.getStartState()      #初始节点
    closed = []      #标记已经遍历过的节点, 置为空
    q = util.Queue()      #建立队列来保存拓展到的各节点
    q.push((s,[]))
    while not q.isEmpty():
        state,path = q.pop()
        if problem.isGoalState(state):   #如果是目标状态, 则返回当前路径, 退出函数
            return path
        if state not in closed:
            closed.append(state)     #标记其为已经遍历过的状态
            for node in problem.getSuccessors(state):
                n_state = node[0]
                direction = node[1]
                if n_state not in closed:    #如果后继状态未被遍历过, 将其入队列
                    q.push((n_state, path + [direction]))
    return path
```

问题三：代价一致算法

```python
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    #初始状态
    start = problem.getStartState()
    #标记已经搜索过的状态集合exstates
    exstates = []
    #用优先队列PriorityQueue实现ucs
    states = util.PriorityQueue()
    states.push((start,[]),0)
    while not states.isEmpty():
        state,actions = states.pop()
        #目标测试
        if problem.isGoalState(state):
            return actions
        #检查重复
        if state not in exstates:
            #拓展
            successors = problem.getSuccessors(state)
            for node in successors:
                coordinate = node[0]
                direction = node[1]
                if coordinate not in exstates:
                    newActions = actions + [direction]
                    #ucs比bfs的区别在于getCostOfActions决定节点拓展的优先级
                    states.push((coordinate,actions + [direction]),problem.getCostOfActions(newActions))
        exstates.append(state)
    return  actions
    util.raiseNotDefined()
```

问题四：A*算法

```python
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"
    start = problem.getStartState()     #初始状态
    exstates = []        #是否访问过该节点，初始为空
    states = util.PriorityQueue()
    states.push((start,[]),nullHeuristic(start,problem))        #初始节点入栈
    nCost = 0
    while not states.isEmpty():
        state,actions = states.pop()
        if problem.isGoalState(state):      #到达目标节点，退出
            return actions
        if state not in exstates:
            successors = problem.getSuccessors(state)        #查找子节点
            for node in successors:
                coordinate = node[0]
                direction = node[1]
                if coordinate not in exstates:
                    newActions = actions + [direction]
                    newCost = problem.getCostOfActions(newActions) + heuristic(coordinate,problem)
                    states.push((coordinate,actions + [direction]),newCost)
        exstates.append(state)
    return  actions
    util.raiseNotDefined()
```

问题五：查找所有角落

```python
    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        "*** YOUR CODE HERE ***"
        return (self.startingPosition, [])

    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"
        location = state[0]
        corner_state = state[1]
        return len(corner_state) == 4       #角落状态表中不含重复状态，包含四个状态时达到目标
        util.raiseNotDefined()
```

```python
def getSuccessors(self, state):
    """
    Returns successor states, the actions they require, and a cost of 1.

     As noted in search.py:
        For a given state, this should return a list of triples, (successor,
        action, stepCost), where 'successor' is a successor to the current
        state, 'action' is the action required to get there, and 'stepCost'
        is the incremental cost of expanding to that successor
    """

    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 ***"
        corner_state = state[1]
        list_corner_state = list(corner_state)
        x,y = state[0]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)     #计算下一个状态的位置坐标
        hitsWall = self.walls[nextx][nexty]
        if not hitsWall:
            if ((nextx,nexty) in self.corners) & ((nextx,nexty) not in list_corner_state):      #是一个角落位置并且未被遍历过，将其加入角落状态表中
                list_corner_state.append((nextx, nexty))
            successor = (((nextx,nexty), list_corner_state), action, 1)
            successors.append(successor)
    self._expanded += 1 # DO NOT CHANGE
    return successors
```

问题六：角落问题——启发式

```python
def cornersHeuristic(state, problem):
    """
    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).
    """
    #角落坐标
    corners = problem.corners
    #墙壁坐标
    walls = problem.walls
    #起始源点
    node = state[0]
    un_viscorner = [item for item in corners if item not in state[1]]
    #计算所有未经过的角落的到现在距离的最小曼哈顿值
    hn = minmanhattan(un_viscorner,state[0])
    return hn
    #"*** YOUR CODE HERE ***"
    #return 0 # Default to trivial solution

#遍历所有的corner的豆子，选择最小的曼哈顿距离作为启发函数
def minmanhattan(corners,pos):
    #corners长度为0，返回0
    if len(corners) == 0:
        return 0
    hn = []
    #循环计算每一个点的曼哈顿距离
    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))
        hn.append(dis)
    #返回最小的曼哈顿值
    return min(hn)
```

问题七：吃掉所有的“豆”

```python
"""
position, foodGrid = state
food_available = []
hvalue = 0
for i in range(0,foodGrid.width):
    for j in range(0,foodGrid.height):
        if(foodGrid[i][j] == True):
            food_location = (i,j)
            food_available.append(food_location)
if(len(food_available) == 0):
    return 0
#初始化距离为0
max_distance=((0,0),(0,0),0)
for current_food in food_available:
    for next_food in food_available:
        if(current_food==next_food):
            pass
        else:
            #使用曼哈顿距离构造启发式函数
            distance = util.manhattanDistance(current_food,next_food)
            if(max_distance[2] < distance):
                max_distance = (current_food,next_food,distance)
#把起点和第一个搜索的食物连接起来
#处理只有一个食物的情况
if(max_distance[0]==(0,0) and max_distance[1]==(0,0)):
    hvalue = util.manhattanDistance(position,food_available[0])
else:
    d1 = util.manhattanDistance(position,max_distance[0])
    d2 = util.manhattanDistance(position,max_distance[1])
    hvalue = max_distance[2] + min(d1,d2)
#Breaking Tie
hvalue *=(1.0+0.75)
return hvalue
"*** YOUR CODE HERE ***"
return 0
```

问题八：次优先搜索

```python
def findPathToClosestDot(self, 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)
    return search.aStarSearch(problem)      #调用A*算法，寻找最短路径
```

```python
def __init__(self, gameState):
    "Stores information from the gameState.  You don't need to change this."
    # Store the food for later reference
    self.food = gameState.getFood()

    # Store info for the PositionSearchProblem (no need to change this)
    self.walls = gameState.getWalls()
    self.startState = gameState.getPacmanPosition()
    self.costFn = lambda x: 1
    self._visited, self._visitedlist, self._expanded = {}, [], 0 # DO NOT CHANGE

def isGoalState(self, state):
    """
    The state is Pacman's position. Fill this in with a goal test that will
    complete the problem definition.
    """
    x,y = state
    foodGrid = self.food    #找到食物所在位置
    if(foodGrid[x][y] == True)or(foodGrid.count() == 0):        #判断该点是否有食物
        return True
    # util.raiseNotDefined()
```

三. 算法实现

3.1 实验环境与问题规模

实验环境：python2.7.3

问题规模：不需要占用过多的内存运行

3.2 数据结构

数组：一维数组用于标记搜索过的状态集合，二维数组用于存储豆子的二维位置坐标等信息

栈：利用栈实现 dfs 等

队列：利用优先队列实现 A*算法等

3.3 实验结果

（1）深度优先与宽度优先

问题	地图类型	搜索耗时	路径代价	拓展节点数	得分
深度优先	tinyMaze	0.0	10	15	500

	mediumMaze	0.0	130	146	380
	bigMaze	0.0	210	390	300
宽度优先	tinyMaze	0.0	8	15	502
	mediumMaze	0.0	68	269	442
	bigMaze	0.0	210	620	300

实验结果分析：深度优先搜索不能保证找到最优解，是不完备的搜索策略。广度优先搜索得到的解是搜索树中路径最短的解（最优解），但搜索效率较低。

（2）代价一致+A*

问题	地图类型	搜索耗时	路径代价	拓展节点数	得分
代价一致	tinyMaze	0.0	68	269	442
	mediumMaze	0.0	1	186	186
	bigMaze	0.0	68719479864	108	418
A*	tinyMaze	0.0	8	14	502
	mediumMaze	0.0	68	221	300
	bigMaze	0.0	210	549	300

实验结果分析：代价一致算法找到的是最小代价路径，而最小代价路径不一定是最短路径。A*算法一定能结束在最佳路径上。

(3)角落问题

问题	地图类型	搜索耗时	路径代价	拓展节点数	得分
查找所有角落（BFS）	tinyCorners	0.0	28	435	512
	mediumCorners	0.3	106	2448	434
	bigCorners	4.9	162	9904	378
角落问题（启发式A*）	tinyCorners	0.0	28	311	512
	mediumCorners	0.3	106	1636	434
	bigCorners	2.0	162	5052	378

实验结果分析：在角落问题中，广度优先和 A*算法都可以找到最短路径，但是 A*算法效率更高。此处 A*算法采用的启发函数是曼哈顿距离。

3.4 系统中间及最终输出结果（要求有屏幕显示）

问题一：广度优先搜索——三种地图分别的测试结果

```
D:\HIT\AI\lab\search>python2 pacman.py -l tinyMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 10 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 500
Average Score: 500.0
Scores:        500.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
D:\HIT\AI\lab\search>python2 pacman.py -l mediumMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 130 in 0.0 seconds
Search nodes expanded: 146
Pacman emerges victorious! Score: 380
Average Score: 380.0
Scores:        380.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
D:\HIT\AI\lab\search>python2 pacman.py -l bigMaze -z .5 -p SearchAgent
[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
```

问题二：宽度优先搜索——三种地图分别的测试结果

```
C:\Users\Lenovo\Desktop\flr\search>python2 pacman.py -l tinyMaze -p SearchAgent
-a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 502
Average Score: 502.0
Scores:        502.0
Win Rate:      1/1 (1.00)
Record:        Win
```


```
C:\Users\Lenovo\Desktop\flr\search>python2 pacman.py -l tinyMaze -p SearchAgent
-a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 502
Average Score: 502.0
Scores:        502.0
Win Rate:      1/1 (1.00)
Record:        Win
```


```
C:\Users\Lenovo\Desktop\flr\search>python2 pacman.py -l bigMaze -p SearchAgent -
a fn=bfs -z .5
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

问题三：代价一致算法——三种地图分别的测试结果

```
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Lenovo\Documents\Tencent Files\1271600927\FileRecv\search\search>Python
 pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
C:\Users\Lenovo\Documents\Tencent Files\1271600927\FileRecv\search\search>Python
 pacman.py -l mediumDottedMaze -p StayEastSearchAgent
Path found with total cost of 1 in 0.0 seconds
Search nodes expanded: 186
Pacman emerges victorious! Score: 646
Average Score: 646.0
Scores:        646.0
Win Rate:      1/1 (1.00)
Record:        Win
          半:
```

```
C:\Users\Lenovo\Documents\Tencent Files\1271600927\FileRecv\search\search>Python
 pacman.py -l mediumScaryMaze -p StayWestSearchAgent
Path found with total cost of 68719479864 in 0.0 seconds
Search nodes expanded: 108
Pacman emerges victorious! Score: 418
Average Score: 418.0
Scores:        418.0
Win Rate:      1/1 (1.00)
Record:        Win
```

问题四：A*算法

```
D:\HIT\AI\lab\search>python2 pacman.py -l bigMaze -z .5 -p SearchAgent
[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
```

问题五：找到所有的角落——三种地图分别的测试结果

```
D:\HIT\AI\lab\search>python2 pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 28 in 0.0 seconds
Search nodes expanded: 252
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:        512.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
D:\HIT\AI\lab\search>python2 pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.3 seconds
Search nodes expanded: 1966
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
D:\HIT\AI\lab\search>python2 pacman.py -l bigCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 162 in 4.8 seconds
Search nodes expanded: 7949
Pacman emerges victorious! Score: 378
Average Score: 378.0
Scores:        378.0
Win Rate:      1/1 (1.00)
Record:        Win
```

问题六：角落问题——启发式（两种角落问题）

```
D:\HIT\AI\lab\search>python2 pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 106 in 0.1 seconds
Search nodes expanded: 692
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win

D:\HIT\AI\lab\search>python2 pacman.py -l tinyCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 28 in 0.0 seconds
Search nodes expanded: 154
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:        512.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
D:\HIT\AI\lab\search>python2 pacman.py -l bigCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 162 in 0.5 seconds
Search nodes expanded: 1725
Pacman emerges victorious! Score: 378
Average Score: 378.0
Scores:        378.0
Win Rate:      1/1 (1.00)
Record:        Win
```

问题七：吃掉所有的“豆”

```
D:\HIT\AI\lab\search>python2 pacman.py -l trickySearch -p AStarFoodSearchAgent
Path found with total cost of 60 in 5.6 seconds
Search nodes expanded: 7101
Pacman emerges victorious! Score: 570
Average Score: 570.0
Scores:        570.0
Win Rate:      1/1 (1.00)
Record:        Win
```

问题八：次优先搜索

```
C:\Users\Lenovo\Documents\Tencent Files\1271600927\FileRecv\search\search>python
 pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
[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
```

四． 总结及讨论（对该实验的总结以及任何该实验的启发），

此次实验通过吃豆人的经典游戏作为实验背景，通过实验设计相应

的情景问题：如找到所有角落的豆子、吃最近的“豆”等，实现我们对于相关算法的理解与运用，同时让我们了解到人工智能具体的运用实例以及相应的运用效果。

五、实验贡献度

姓名	贡献度	工作
韩雪婷	25%	实验 2 问题 147
陈抒语	25%	实验 1 大部分
廖思瑀	25%	实验 2 问题 368
卢茉莉	25%	实验 2 问题 25