一、问题描述

1.1 待解决问题

实验要求采用且不限于课程第四章内各种搜索算法此编写一系列吃豆人程序解决以下列出的问题 1-8，包括到达指定位置以及有效的吃豆等。

问题 1：应用深度优先算法找到一个特定的位置的豆；

问题 2：应用宽度优先算法实现以上问题；

问题 3：完成代价一致搜索算法；

问题 4：完成 A*搜索算法，利用曼哈顿距离作为启发函数；

问题 5：迷宫的四个角上有四个豆，要求找到访问所有四个角落的最短路径。

问题 6：构建合适的启发函数，完成启发式角落搜索问题。

问题 7：用尽可能少的步数吃掉所有的豆子；构建合适的启发函数，完成启发式豆子搜索问题。

问题 8：定义一个优先吃最近的豆子函数以实现次最优搜索。

1.2 解决方案介绍

问题 1、2、3、4 分别考察深度优先、广度优先、代价一致、A*算法，它们的结构相似，不同的地方在于 util.py 中数据结构的选择，即对于 open 表的排序不同：深度优先用到的是栈 Stack，广度优先用到的是队列 Queue，代价一致和 A*算法用到的是优先队列 PriorityQueue。4 个算法的具体介绍见第二部分。

问题 5 用状态压缩判断哪些豆子没吃。状态压缩具体指，四个角落用四位二进制数表示，初始状态为 0000，该角落的豆子被吃了就置 1,最终状态为 1111。

问题 6 从当前位置出发，以某种顺序吃掉剩余所有豆子所需要走的总距离最小，这个最小的距离即为当前位置的启发函数。吃掉剩余所有豆子的顺序可用枚举法，假设剩余 n（n≤4）颗豆子，即枚举 n！种情况，取这 n！种情况中总距离最小的情况作为启发函数。其中估计距离时采用的是曼哈顿距离。

问题 7 以当前位置为原点，用水平线和垂直线将整个图划分为四个象限，取每个象限中距离当前位置最远的豆子，启发函数为这四个最远的豆子中与当前位置最近的豆子的距离。当前位置与每个豆子的距离是用的实际距离，而不是用曼哈顿距离直接估计的，其中实际距离是由 BFS 实际走了一遍得到的。

问题 8 采用广度优先算法，找到的第一个豆子就是距离当前位置最近的豆子。

二、算法介绍

（一）问题 1~4

下面是前 4 个问题的算法的统一**结构**：

(1) 把初始节点放入 open 表中，建立一个 close 表，置为空；

(2) 检查 open 表是否为空表，若为空，则问题无解，失败退出；

(3) 把 open 表的第一个节点取出放入 closed 表，并记该节点为 n；

(4) 考察节点 n 是否为目标节点，若是则得到问题的解成功退出；

(5) 若结点 n 不可扩展，则转第二步；

(6) 扩展节点 n，将其子节点按照不同的排列顺序放入 open 表中，并为每个子节点设置指向父节点的指针，转向第二步。

该结构用**伪代码**表示如下：

```
function Graph-Search(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]),fringe)
    loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST(problem, STATE[node]) then return node
      if STATE[node] is not in closed then
          add STATE[node] to closed
          for child-node in EXPAND(STATE[node], problem) do
              fringe ← INSERT(child-node, fringe)
          end
    end
```

对于深度优先搜索：

(6a) 扩展节点 n，将其子节点放入 open 表的首部，并为每个子节点设置指向父节点的指针，转向第二步。

对于广度优先搜索：

(6b) 扩展节点 n，将其子节点放入 open 表的尾部，并为每个子节点设置指向父节点的指针，转向第二步。

对于代价一致搜索：

(6c) 扩展节点 n，生成子节点 n_i(i=1,2,...)，将其子节点放入 open 表，并为每个子节点设置指向父节点的指针，计算各个节点的代价 $g(n_i)$，将 open 表内的节点按 $g(n_i)$从小到大排序，转向第二步。

对于 A*算法：

(6d) 扩展节点 n，生成其子节点 n_i (i=1,2,..)，计算每一个节点的估计值 $f(n_i) = g(n_i) + h(n_i)$ (i=1,2,..)，并为每一个子节点设置指向父节点的指针，然后根据各节点的估价函数值，对 open 表中的全部节点从小到大进行排序，转第二步。

（二）问题 5~8

(1) 问题 5：找到所有的角落

用状态压缩判断哪些豆子没吃。状态压缩具体指，四个角落用四位二进制数

表示，初始状态为 0000，该角落的豆子被吃了就置 1,最终状态为 1111。

(2) 问题 6：角落问题（启发式）

从当前位置出发，以某种顺序吃掉剩余所有豆子所需要走的总距离最小，这个最小的距离即为当前位置的启发函数。吃掉剩余所有豆子的顺序可用枚举法，假设剩余 n（n≤4）颗豆子，即枚举 n！种情况，取这 n！种情况中总距离最小的情况作为启发函数。其中估计距离时采用的是曼哈顿距离。

(3) 问题 7：吃掉所有的豆子

以当前位置为原点，用水平线和垂直线将整个图划分为四个象限，取每个象限中距离当前位置最远的豆子。当前位置与每个豆子的距离是用的实际距离，而不是用曼哈顿距离直接估计的，其中实际距离是由 BFS 实际走了一遍得到的。

(4) 问题 8：次最优搜索

BFS 找到的第一个豆子就是距离当前位置最近的豆子。

三、算法实现

3.1 实验环境和问题规模

实验环境：Python 3.7(64 bit)，PyCharm 2019.3.3

Python interpreter: Python 3.7 E:\APP\anaconda\python.exe

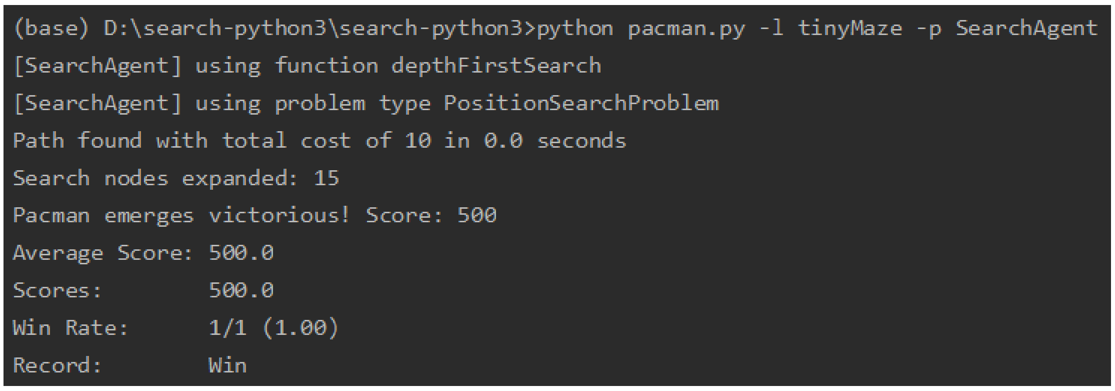
问题复杂度大部分由于 open 表的排序，我们的代码中时间复杂度 O(n^2)，空间复杂度为 O(n)，n 为地图中全部结点数。

3.2 数据结构

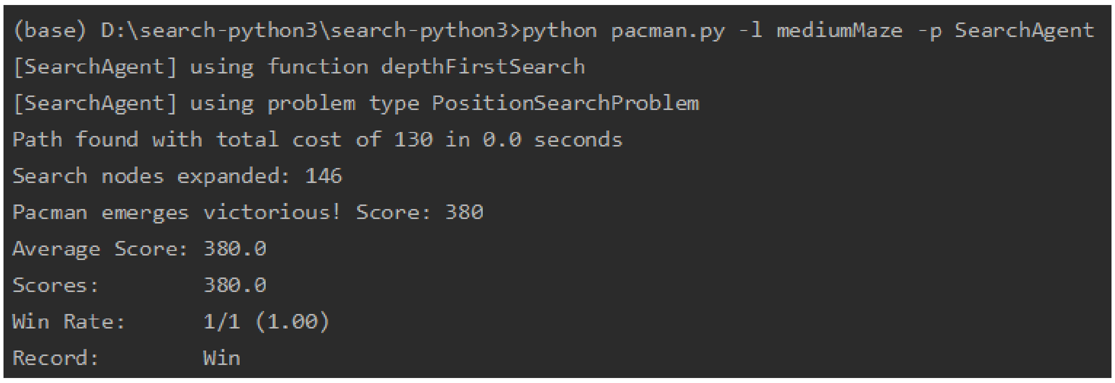
util.py 文件中提供的 Stack, Queue, PriorityQueue。

3.3 实验结果

问题 1：python pacman.py -l tinyMaze -p SearchAgent

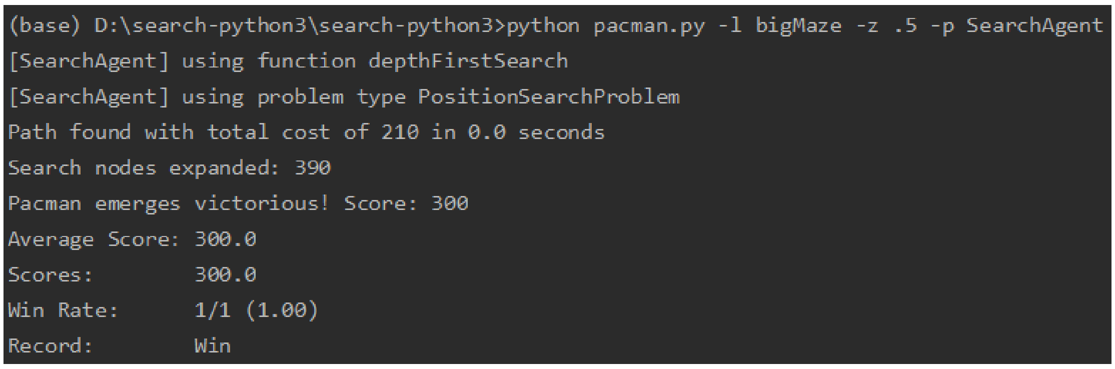

```
(base) D:\search-python3\search-python3>python pacman.py -l tinyMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 10 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 500
Average Score: 500.0
Scores:        500.0
Win Rate:      1/1 (1.00)
Record:        Win
```

python pacman.py -l mediumMaze -p SearchAgent



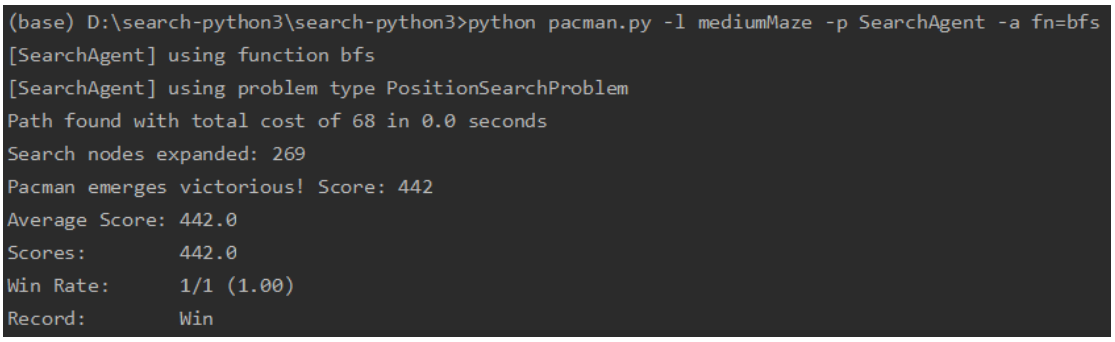
```
(base) D:\search-python3\search-python3>python pacman.py -l mediumMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 130 in 0.0 seconds
Search nodes expanded: 146
Pacman emerges victorious! Score: 380
Average Score: 380.0
Scores:        380.0
Win Rate:      1/1 (1.00)
Record:        Win
```

python pacman.py -l bigMaze -z .5 -p SearchAgent



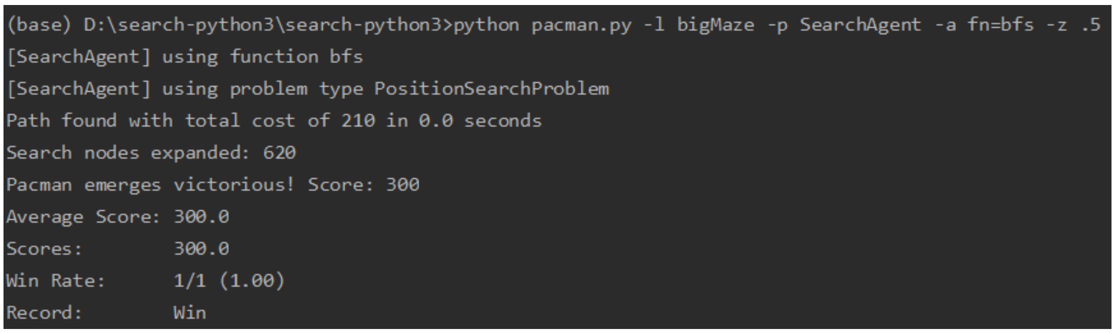
```
(base) D:\search-python3\search-python3>python pacman.py -l bigMaze -z .5 -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 390
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

问题 2：python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs



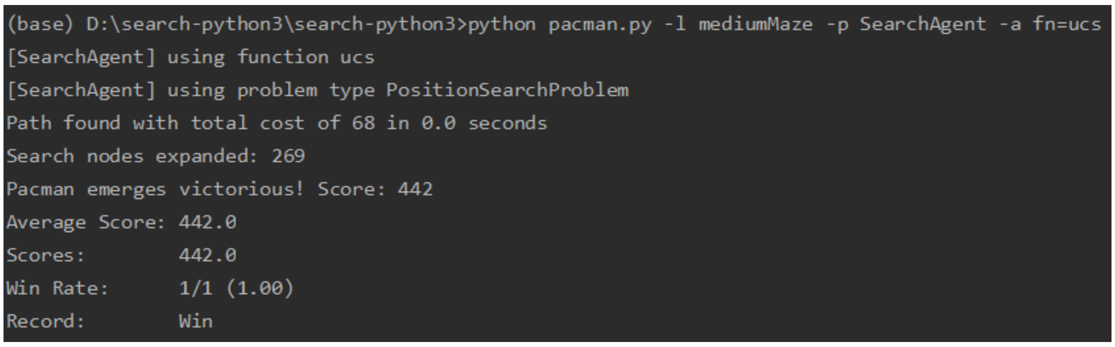
```
(base) D:\search-python3\search-python3>python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5



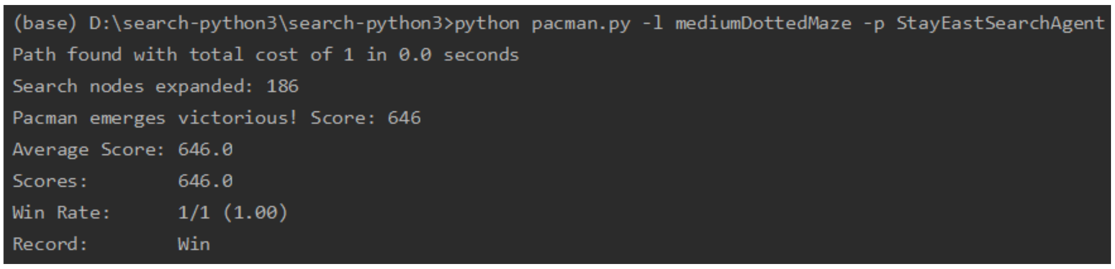
```
(base) D:\search-python3\search-python3>python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

问题 3：python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs



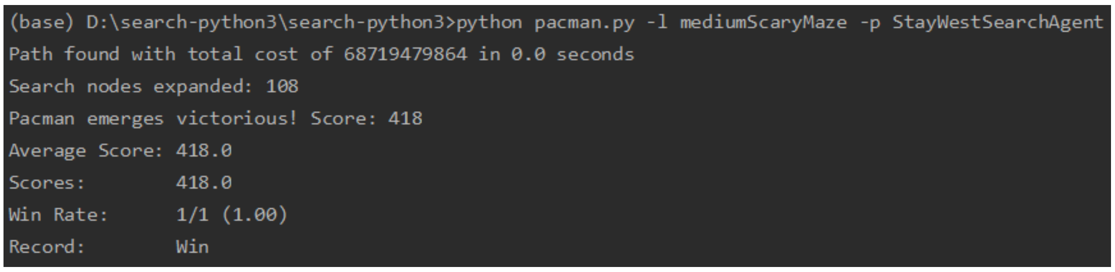
```
(base) D:\search-python3\search-python3>python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

python pacman.py -l mediumDottedMaze -p StayEastSearchAgent



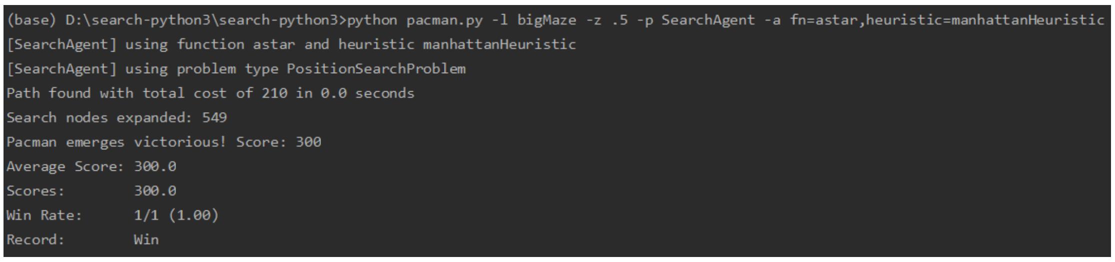
```
(base) D:\search-python3\search-python3>python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
Path found with total cost of 1 in 0.0 seconds
Search nodes expanded: 186
Pacman emerges victorious! Score: 646
Average Score: 646.0
Scores:        646.0
Win Rate:      1/1 (1.00)
Record:        Win
```

python pacman.py -l mediumScaryMaze -p StayWestSearchAgent



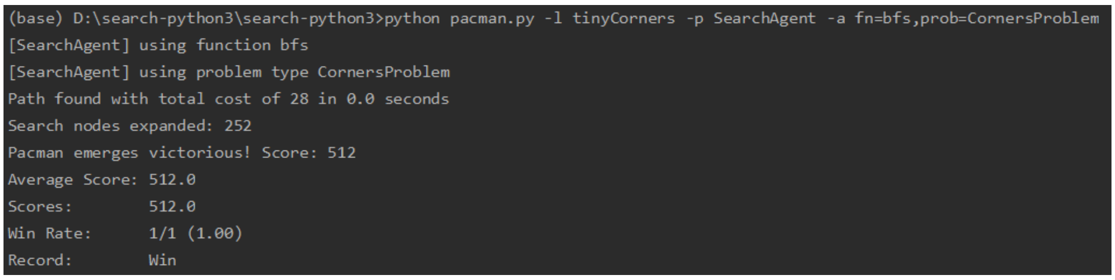
```
(base) D:\search-python3\search-python3>python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
Path found with total cost of 68719479864 in 0.0 seconds
Search nodes expanded: 108
Pacman emerges victorious! Score: 418
Average Score: 418.0
Scores:        418.0
Win Rate:      1/1 (1.00)
Record:        Win
```

问题 4： python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic



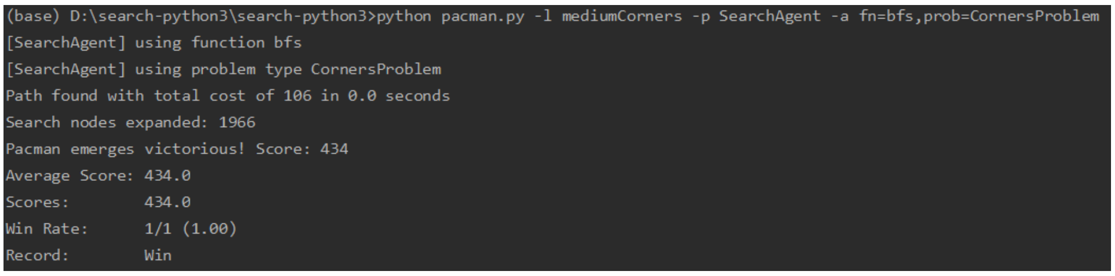
```
(base) D:\search-python3\search-python3>python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 549
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

问题 5： python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem



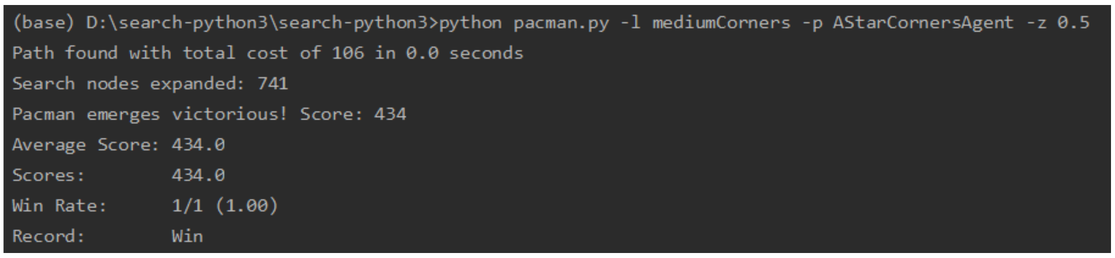
```
(base) D:\search-python3\search-python3>python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 28 in 0.0 seconds
Search nodes expanded: 252
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:        512.0
Win Rate:      1/1 (1.00)
Record:        Win
```

python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem



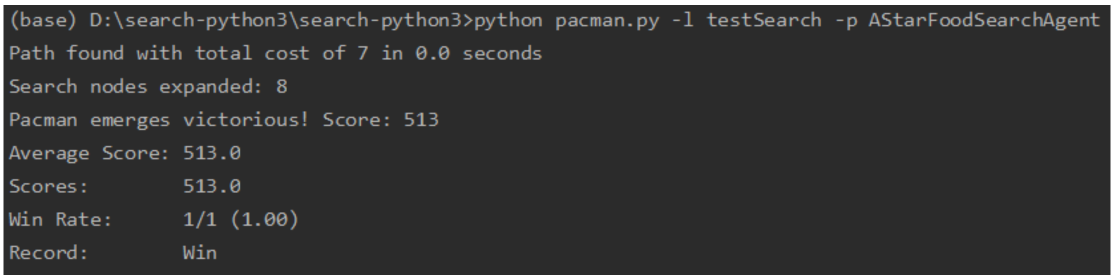
```
(base) D:\search-python3\search-python3>python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.0 seconds
Search nodes expanded: 1966
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

问题 6：python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5



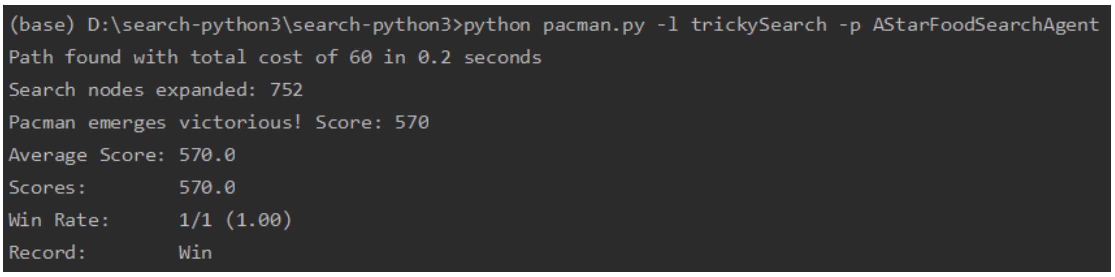
```
(base) D:\search-python3\search-python3>python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 106 in 0.0 seconds
Search nodes expanded: 741
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

问题 7：python pacman.py -l testSearch -p AStarFoodSearchAgent



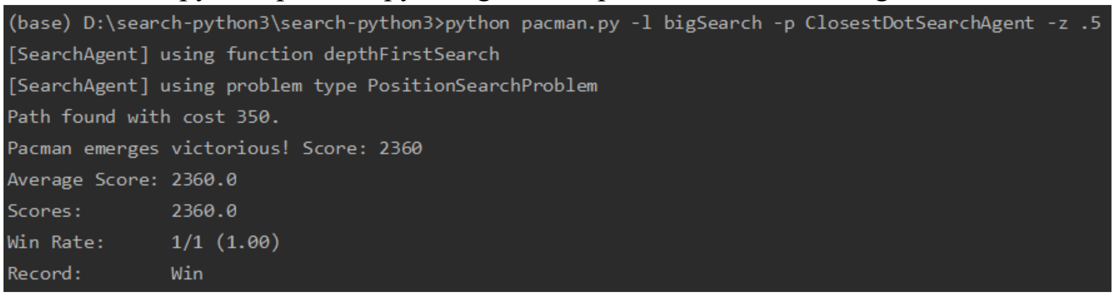
```
(base) D:\search-python3\search-python3>python pacman.py -l testSearch -p AStarFoodSearchAgent
Path found with total cost of 7 in 0.0 seconds
Search nodes expanded: 8
Pacman emerges victorious! Score: 513
Average Score: 513.0
Scores:        513.0
Win Rate:      1/1 (1.00)
Record:        Win
```

python pacman.py -l trickySearch -p AStarFoodSearchAgent



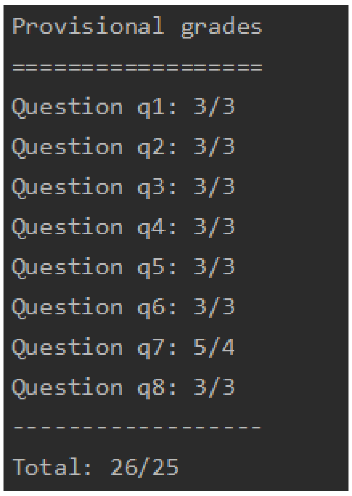
```
(base) D:\search-python3\search-python3>python pacman.py -l trickySearch -p AStarFoodSearchAgent
Path found with total cost of 60 in 0.2 seconds
Search nodes expanded: 752
Pacman emerges victorious! Score: 570
Average Score: 570.0
Scores:        570.0
Win Rate:      1/1 (1.00)
Record:        Win
```

问题 8：python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5



```
(base) D:\search-python3\search-python3>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
```

总评分：python autograder.py


```
Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 3/3
Question q6: 3/3
Question q7: 5/4
Question q8: 3/3
------------------
Total: 26/25
```

四、总结讨论

本次实验吃豆人游戏很有趣，动手实现了 4 个经典的搜索策略：深度优先、广度优先、代价一致和 A*算法，学习并实践了启发式函数的选取要求。在已有的比较成熟的代码上补全 2 个.py 文件，需要我们阅读并理解不止这两个原有程序代码，也是非常锻炼自身能力的一次实验。

附录：源代码及其注释

（一）search.py

（1）关于广度优先搜索算法的代码如下：

```
def depthFirstSearch(problem):
    "*** YOUR CODE HERE ***"
    open = util.Stack()
    open.push(problem.getStartState())#初始节点入栈
   close = {}
    pre = {}
    ans = []
    while not open.isEmpty():
        u = open.pop()
        if u in close:
            continue
        close[u] = True
        if problem.isGoalState(u):#如果节点是目标节点，根据路径记录返回到初始节点并
记录路径
          p = u
            while p in pre:
                fa, Dir = pre[p]
                ans.append(Dir)
                p = fa
            ans.reverse()
            break
        for v, dir, cost in problem.getSuccessors(u):#得到后继节点
          if v in close:#如果当前节点已在 close 表中，则直接退出
             continue
            pre[v] = (u, dir)#记录扩展节点的父节点
          open.push(v)
    return ans
```

（2）关于宽度优先搜索算法的代码如下：

```
def breadthFirstSearch(problem):
    "*** YOUR CODE HERE ***"
    open = util.Queue()
    open.push(problem.getStartState())#初始节点入队列
   close = {}
    pre = {}
    ans = []
    close[problem.getStartState()] = True
    while not open.isEmpty():
        u = open.pop()
        if problem.isGoalState(u):#如果节点是目标节点，根据路径记录返回到初始节点并
记录路径
```

```
            p = u
            while p in pre:
                fa, Dir = pre[p]
                ans.append(Dir)
                p = fa
            ans.reverse()
            break
        for v, dir, cost in problem.getSuccessors(u):#得到后继节点
          if v in close:#如果当前节点已在 close 表中，则直接退出
              continue
            close[v] = True
            pre[v] = (u, dir)#记录扩展节点的父节点
          open.push(v)
    return ans
```

（3）关于代价一致搜索算法的代码如下：

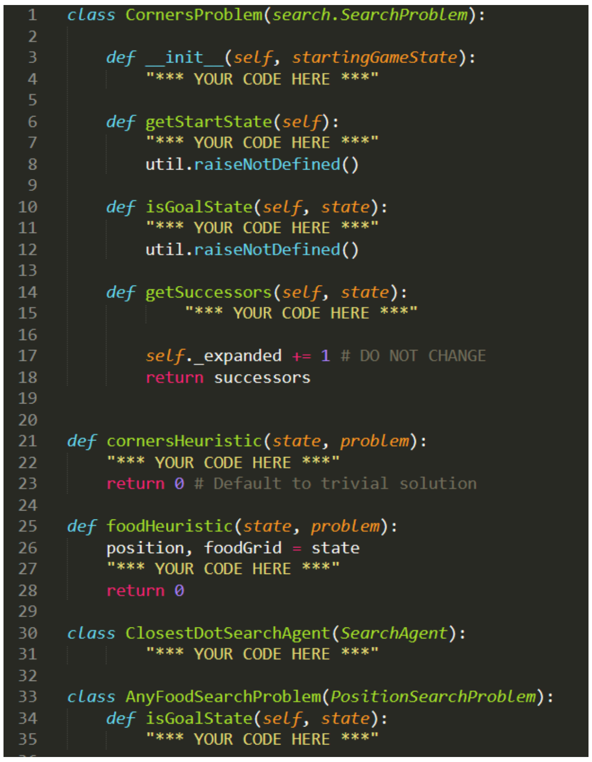
```
def uniformCostSearch(problem):
    "*** YOUR CODE HERE ***"
    open = util.PriorityQueue()
    open.push((problem.getStartState(), 0), 0)#初始节点入优先队列
   close = {}
    pre = {}
    ans = []
    close[problem.getStartState()] = 0
    while not open.isEmpty():
        u, w = open.pop()
        if w > close[u]:
            continue
        if problem.isGoalState(u):#如果节点是目标节点，根据路径记录返回到初始节点并
记录路径
          p = u
            while p in pre:
                fa, Dir = pre[p]
                ans.append(Dir)
                p = fa
            ans.reverse()
            break
        for v, dir, cost in problem.getSuccessors(u):#得到后继节点
          if v in close and w + cost >= close[v]:#如果当前节点已在 close 表中并且代
价比目标节点大，则直接退出
             continue
            close[v] = w + cost
            pre[v] = (u, dir)#记录扩展节点的父节点
          open.push((v, w + cost), w + cost)
    return ans
```

（4）关于 A*算法的代码如下：

```
def aStarSearch(problem, heuristic=nullHeuristic):
    "*** YOUR CODE HERE ***"
    open = util.PriorityQueue()
    open.push((problem.getStartState(), 0), heuristic(problem.getStartState(),
problem))#初始节点入优先队列
   close = {}
    pre = {}
    ans = []
    close[problem.getStartState()] = 0
    while not open.isEmpty():
        u, w = open.pop()
        if w > close[u]:
            continue
        if problem.isGoalState(u):#如果节点是目标节点，根据路径记录返回到初始节点并
记录路径
          p = u
            while p in pre:
                fa, Dir = pre[p]
                ans.append(Dir)
                p = fa
            ans.reverse()
            break
        for v, dir, cost in problem.getSuccessors(u):#得到后继节点
          if v in close and w + cost >= close[v]:#如果当前节点已在 close 表中并且代
价比目标节点大，则直接退出
             continue
            close[v] = w + cost
            pre[v] = (u, dir)#记录扩展节点的父节点
          open.push((v, w + cost), w + cost + heuristic(v, problem))#在优先队列
中比较的参数需要加上代价函数
   return ans
```

（二）searchAgents.py【只保留了 YOU CODE HERE 的部分，课程已给代码未贴出】

下面是我们要填充的代码的结构总览：



```python
class CornersProblem(search.SearchProblem):

    def __init__(self, startingGameState):
        "*** YOUR CODE HERE ***"

    def getStartState(self):
        "*** YOUR CODE HERE ***"
        util.raiseNotDefined()

    def isGoalState(self, state):
        "*** YOUR CODE HERE ***"
        util.raiseNotDefined()

    def getSuccessors(self, state):
            "*** YOUR CODE HERE ***"

        self._expanded += 1 # DO NOT CHANGE
        return successors


def cornersHeuristic(state, problem):
    "*** YOUR CODE HERE ***"
    return 0 # Default to trivial solution

def foodHeuristic(state, problem):
    position, foodGrid = state
    "*** YOUR CODE HERE ***"
    return 0

class ClosestDotSearchAgent(SearchAgent):
        "*** YOUR CODE HERE ***"

class AnyFoodSearchProblem(PositionSearchProblem):
    def isGoalState(self, state):
        "*** YOUR CODE HERE ***"
```

（1）class CornersProblem(search.SearchProblem):

①def __init__(self, startingGameState):

```python
def __init__(self, startingGameState):
    "*** YOUR CODE HERE ***"
    self.startState = (self.startingPosition, 0)  # 初始状态共有两个状态，第一个状
态是初始位置，第二个状态为状态压缩后表示四个角落已经吃到的情况
```

②def getStartState(self):

```python
def getStartState(self):
    "*** YOUR CODE HERE ***"
    return self.startState  # 返回初始位置
```

③def isGoalState(self, state):

```python
def isGoalState(self, state):
    "*** YOUR CODE HERE ***"
    return state[1] == 15  # 如果四个角落都吃掉了则是目标状态
```

④def getSuccessors(self, state):

```python
def getSuccessors(self, state):
        "*** YOUR CODE HERE ***"
        x, y = state[0]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        if not self.walls[nextx][nexty]:  # 如果该方向不是墙
          nextState = (nextx, nexty)
            if nextState in self.corners:  # 如果该节点为角落，则更新状态情况
             successors.append(((nextState, state[1] | (1 <<
self.corners.index(nextState))), action, 1))
```

```
            else:
                successors.append(((nextState, state[1]), action, 1))
```

（2）def cornersHeuristic(state, problem):

```
def cornersHeuristic(state, problem):
    "*** YOUR CODE HERE ***"
    unVisitedCorners = []
    for i in range(len(corners)):  # 得到没吃到的角落
      if ((state[1] >> i) & 1) == 0:
            unVisitedCorners.append(corners[i])
    finalState = (1 << len(unVisitedCorners)) - 1

    def dfs(u, State, cost):  # dfs 得到从当前位置遍历剩余没去过的顶点的最短距离，距
离使用的是曼哈顿距离
      if State == finalState:
            return cost
        ans = []
        for i in range(len(unVisitedCorners)):
            if ((State >> i) & 1) == 0:
                ans.append(
                    dfs(unVisitedCorners[i], State | (1 << i), cost +
util.manhattanDistance(u, unVisitedCorners[i])))
        return min(ans)

    return dfs(state[0], 0, 0)  # Default to trivial solution
```

（3）def foodHeuristic(state, problem):

```
def foodHeuristic(state, problem):
    "*** YOUR CODE HERE ***"

    corners = [(position, 0), (position, 0), (position, 0), (position, 0)]
    x, y = position
    for u in foodGrid.asList():  # 挑选当前位置四个象限最远距离的点，采用的是曼哈顿
距离作为比较参数
      w = util.manhattanDistance(position, u)
        if u[0] <= x and u[1] <= y:
            if w > corners[0][1]:
                corners[0] = (u, w)
        elif u[0] > x and u[1] <= y:
            if w > corners[1][1]:
                corners[1] = (u, w)
        elif u[0] <= x and u[1] > y:
            if w > corners[2][1]:
                corners[2] = (u, w)
        else:
            if w > corners[3][1]:
                corners[3] = (u, w)

    # 方法一，采用的是实际最短距离作为参数

    def bfs(s):  # bfs 得到当前得到当前点到其余点的距离
        q = util.Queue()
        vis = {}
        q.push((s, 0))
        vis[s] = True
        while not q.isEmpty():
            u, w = q.pop()
            problem.heuristicInfo[(s, u)] = w  # BFS 记录单源最短路路径
            for direction in [Directions.NORTH, Directions.SOUTH,
Directions.EAST, Directions.WEST]:
                x, y = u
                dx, dy = Actions.directionToVector(direction)
                nextx, nexty = int(x + dx), int(y + dy)
                v = (nextx, nexty)
                if problem.walls[nextx][nexty] or v in vis:
                    continue
                vis[v] = True
                q.push((v, w + 1))
            # problem._expanded += 1 #如果内部 BFS 算扩展节点则取消注释

    if (position, position) not in problem.heuristicInfo:  # 如果当前搜索位置没有
经历过 BFS，则进行 BFS
        bfs(position)

    for i in range(len(corners)):
        if (corners[i][0], corners[i][0]) not in problem.heuristicInfo:  # 如果
选择的四个点没有经历过 BFS，则进行 BFS
            bfs(corners[i][0])

    # 方法二，考虑横向墙的阻挡和竖向强的阻挡，采用记忆化搜索的策略，搭建出只考虑横向阻
挡和竖向阻挡的最远距离

    # def disrow(u, v): # 只考虑横向阻挡的搜索
    #     if (u, v, 'row') in problem.heuristicInfo: # 记忆化
    #         return problem.heuristicInfo[(u, v, 'row')]
    #     ret = util.manhattanDistance(u, v)
    #     if u[0] + 1 < v[0]:
    #         res = float('inf')
    #         for i in range(problem.walls.height):
    #             if not problem.walls[u[0] + 1][i]:
    #                 res = min(res, disrow(u, (u[0] + 1, i)) + disrow((u[0] +
1, i), v))
    #         ret = max(ret, res)
    #     if v[0] + 1 < u[0]:
    #         res = float('inf')
    #         for i in range(problem.walls.height):
    #             if not problem.walls[v[0] + 1][i]:
    #                 res = min(res, disrow(u, (v[0] + 1, i)) + disrow((v[0] +
1, i), v))
    #         ret = max(ret, res)
    #     problem.heuristicInfo[(u, v, 'row')] = ret
    #     problem.heuristicInfo[(v, u, 'row')] = ret
    #     return ret
    #
    # def discol(u, v): # 只考虑竖向阻挡的搜索
   #     if (u, v, 'col') in problem.heuristicInfo: # 记忆化
   #         return problem.heuristicInfo[(u, v, 'col')]
    #     ret = util.manhattanDistance(u, v)
    #     if u[1] + 1 < v[1]:
    #         res = float('inf')
    #         for i in range(problem.walls.width):
    #             if not problem.walls[i][u[1] + 1]:
    #                 res = min(res, discol(u, (i, u[1] + 1)) + discol((i, u[1]
+ 1), v))
    #         ret = max(ret, res)
    #     if v[1] + 1 < u[1]:
    #         res = float('inf')
    #         for i in range(problem.walls.width):
    #             if not problem.walls[i][v[1] + 1]:
    #                 res = min(res, discol(u, (i, v[1] + 1)) + discol((i, v[1]
+ 1), v))
    #         ret = max(ret, res)
    #     problem.heuristicInfo[(u, v, 'col')] = ret
    #     problem.heuristicInfo[(v, u, 'col')] = ret
    #     return ret
    #
    # def dis(u, v): # 只考虑竖向阻挡的 dfs 搜索
   #     if (u, v) in problem.heuristicInfo: # 记忆化
   #         return problem.heuristicInfo[(u, v)]
    #     ret = max(discol(u, v), disrow(u, v))
    #     problem.heuristicInfo[(u, v)] = ret
    #     problem.heuristicInfo[(v, u)] = ret
    #     return ret

    # 方法三：考虑每行以及每列对于两点之间距离的阻挡，选择每行或者每列中可以同行的点作
为中间点，每行或者每列当作取每次的最小值，所有的行和所有的列中取最大值

   # def dis(u, v):
    #     if (u, v) in problem.heuristicInfo: # 记忆化
   #         return problem.heuristicInfo[(u, v)]
    #     ret = util.manhattanDistance(u, v)
    #     up = u[0]
    #     down = v[0]
    #     if up > down:
    #         up, down = down, up
    #     for i in range(up + 1, down): # 枚举每行的最小值并取其中的最大值
   #         if i - 1 < v[0]:
    #             if ((i - 1, u[1]), v) in problem.heuristicInfo: # 记忆化如果当
前行到目标行搜索过则直接退出
   #                 ret = max(ret, problem.heuristicInfo[((i - 1, u[1]), v)] +
abs(u[0] - i + 1))
    #                 break
    #         else:
    #             if (u, (i - 1, v[1])) in problem.heuristicInfo: # 记忆化如果当
前行到目标行搜索过则直接退出
   #                 ret = max(ret, problem.heuristicInfo[(u, (i - 1, v[1]))] +
abs(v[0] - i + 1))
    #                 break
    #         res = float('inf')
    #         for j in range(problem.walls.height):
    #             if not problem.walls[i][j]:
    #                 res = min(res, util.manhattanDistance(u, (i, j)) +
util.manhattanDistance((i, j), v))
    #         ret = max(ret, res)
    #     up = u[1]
    #     down = v[1]
    #     if up > down:
    #         up, down = down, up
    #     for i in range(up + 1, down): # 枚举每列的最小值并取其中的最大值
   #         if i - 1 < v[1]:
    #             if ((u[0], i - 1), v) in problem.heuristicInfo: # 记忆化如果当
前列到目标列搜索过则直接退出
   #                 ret = max(ret, problem.heuristicInfo[((u[0], i - 1), v)] +
abs(u[1] - i + 1))
    #                 break
    #         else:
```

```
    #               if (u, (v[0], i - 1)) in problem.heuristicInfo: # 记忆化如果当
前列到目标列搜索过则直接退出
    #                   ret = max(ret, problem.heuristicInfo[(u, (v[0], i - 1))] +
abs(v[1] - i + 1))
    #                   break
    #           res = float('inf')
    #           for j in range(problem.walls.width):
    #               if not problem.walls[j][i]:
    #                   res = min(res, util.manhattanDistance(u, (j, i)) +
util.manhattanDistance((j, i), v))
    #           ret = max(ret, res)
    #       problem.heuristicInfo[(u, v)] = ret
    #       problem.heuristicInfo[(v, u)] = ret
    #       return ret

    def dfs(u, State, cost):  # dfs 得到从当前位置遍历剩余没去过的顶点的最短距离，距
离使用的是曼哈顿距离
        if State == 15:
            return cost
        ans = []
        for i in range(len(corners)):
            if ((State >> i) & 1) == 0:
                ans.append(
                    dfs(corners[i][0], State | (1 << i), cost +
problem.heuristicInfo[(u, corners[i][0])])) # 采用方法一
                # ans.append(
                #     dfs(corners[i][0], State | (1 << i), cost + dis(u,
corners[i][0]))) # 采用方法二、方法三
                # ans.append(
                #     dfs(corners[i][0], State | (1 << i), cost +
util.manhattanDistance(u, corners[i][0]))) # 直接采用曼哈顿距离
        return min(ans)

    return dfs(position, 0, 0)
```

（4）class ClosestDotSearchAgent(SearchAgent):

```
class ClosestDotSearchAgent(SearchAgent):
        "*** YOUR CODE HERE ***"
        return search.bfs(problem)  # 调用系统的 bfs 得到最近的节点的路径
```

（5）class AnyFoodSearchProblem(PositionSearchProblem):

①def isGoalState(self,state):

```python
def isGoalState(self, state):
    "*** YOUR CODE HERE ***"
    return self.food[x][y]  # 如果当前位置是 food 则为目标节点
```