P01 Pacman Game

学号	姓名	专业(方向)
18340013，18340014	陈琮昊，陈嘉宁	人工智能与大数据

1. Idea of A* Algorithm (Use a few sentences to describe your understanding of the algorithm)

- A星算法的主要思想是拓展当前边界节点中预期代价最小的节点，而各个节点的预期代价则由`该节点到根节点的最小距离`和`该节点到目标节点的预期距离`决定，其中`该结点到目标结点的预期距离`由一个启发式函数确定。该启发式函数一般与当前节点到目标结点的距离有关，在实际应用中，常采用曼哈顿距离等作为启发式函数，但无论如何，启发式函数返回的`该节点到目标节点的预期距离`必须小于等于该节点到目标节点的真实距离，且启发式函数需要满足一致性（即对任意两节点A，B，不存在节点C使得h(AB) >= g(AC) + h(CB)，其中h()表示启发式函数,g()表示两点的真实距离）。

2. Idea of Min-Max and $\alpha-\beta$ pruning algorithms

- minimax算法用深度优先搜索来探索整个`game tree`。从树的最底层节点看起：最后一层是MIN层，MIN层节点要做的则是最小化“得分”，根据这一原则可以做出选择；然后倒推到上一层，上一层是MAX层，MAX层节点要做的则是最大化“得分”，同样也可以给出自己的选择......重复这个倒推的步骤，我们可以得到第一步的最优选择。
- 在minimax算法中，要搜索整个树，从最底层节点的值倒推计算，效率显然很低。而$\alpha-\beta$ pruning算法则做了优化，可以节省开销。简单来说，α剪枝就是任一MIN层节点的β值不大于任一前驱MAX层节点的α值，则可以终止该MIN层中这个MIN节点以下的搜索过程，β剪枝则是任一MAX层节点的α值不小于任一前驱MIN层节点的β值，则可以终止该MAX层中这个MAX节点以下的搜索过程。这样减少了运算量。实现过程则与文档里给的伪代码相同。

3. Codes

Question 1

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"
    def f(node):
        g = node[2]
        h = heuristic(node[0], problem)
        return g + h

    startState = problem.getStartState()
    startNode = (startState, [], 0)

    frontier = Queue.PriorityQueue()
    frontier.put((f(startNode), startNode))
    explored = set()

    while not frontier.empty():
        (cur_f, cur_node) = frontier.get()         # current_what
        if problem.isGoalState(cur_node[0]):
            return cur_node[1]
        if len(explored):
            tmp = explored
            # print(tmp.pop())
        if cur_node[0] not in explored:
            explored.add(cur_node[0])
            for isuc in problem.getSuccessors(cur_node[0]):
                suc_state, suc_action, suc_cost = isuc          # successor_what
                new_action = cur_node[1] + [suc_action]
                new_cost = cur_node[2] + suc_cost
                new_node = (suc_state, new_action, new_cost)
                frontier.put((f(new_node), new_node))
    return []

    util.raiseNotDefined()
```

Question 2

```
class CornersProblem(search.SearchProblem):
    """
    This search problem finds paths through all four corners of a layout.

    You must select a suitable state space and successor function
    """

    def __init__(self, startingGameState):
        """
        Stores the walls, pacman's starting position and corners.
        """
        self.walls = startingGameState.getWalls()
        self.startingPosition = startingGameState.getPacmanPosition()
        top, right = self.walls.height-2, self.walls.width-2
        self.corners = ((1,1), (1,top), (right, 1), (right, top))
        for corner in self.corners:
            if not startingGameState.hasFood(*corner):
                print 'Warning: no food in corner ' + str(corner)
        self._expanded = 0 # DO NOT CHANGE; Number of search nodes expanded
        # Please add any code here which you would like to use
        # in initializing the problem
        "*** YOUR CODE HERE ***"
        self.right = right
        self.top = top

    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        "*** YOUR CODE HERE ***"
        return (self.startingPosition, (False, False, False, False))
        util.raiseNotDefined()

    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"
        torf = state[1][0] and state[1][1] and state[1][2] and state[1][3]
        return torf
        util.raiseNotDefined()

    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 ***"
            x, y = state[0]
            corner_state = state[1]
            dx, dy = Actions.directionToVector(action)
            next_x = int(x + dx)
            next_y = int(y + dy)
            next_state_pos = (next_x, next_y)
            next_corner_state = list(corner_state)
            if not self.walls[next_x][next_y]:      # 判断[next_x][next_y]是否在墙
内
                if next_state_pos in self.corners:
                    for i in range(4):
                        if next_state_pos == self.corners[i]:
                            next_corner_state[i] = True
                succ = ((next_state_pos, tuple(next_corner_state)), action, 1)
                successors.append(succ)

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

    def getCostOfActions(self, actions):
        """
        Returns the cost of a particular sequence of actions.  If those actions
        include an illegal move, return 999999.  This is implemented for you.
        """
        if actions == None: return 999999
        x,y= self.startingPosition
        for action in actions:
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]: return 999999
        return len(actions)
```

```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).
    """
    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 ***"
    # 直接忽略墙体考虑吃豆人到四个角落的曼哈顿距离，每次到达一个角落后以该角落为起点找最近角落
    pos = state[0]
    re_corners = []
    for i in range(4):
        if(state[1][i] == False):
            re_corners.append(problem.corners[i])
    cost = 0
    cur_pos = pos
    while len(re_corners) > 0:
        min_dis = 999999
        for i in range(len(re_corners)):
            dis = util.manhattanDistance(cur_pos, re_corners[i])
            if dis < min_dis:
                min_dis = dis
                (x, y) = re_corners[i]
        cost += min_dis
        cur_pos = (x, y)
        re_corners.remove(cur_pos)
    return cost
    #return 0 # Default to trivial solution
```

Question 3

```python
class FoodSearchProblem:
    """
    A search problem associated with finding the a path that collects all of the
    food (dots) in a Pacman game.

    A search state in this problem is a tuple ( pacmanPosition, foodGrid ) where
      pacmanPosition: a tuple (x,y) of integers specifying Pacman's position
      foodGrid:       a Grid (see game.py) of either True or False, specifying
remaining food
    """
    def __init__(self, startingGameState):
        self.start = (startingGameState.getPacmanPosition(),
startingGameState.getFood())
        self.walls = startingGameState.getWalls()
        self.startingGameState = startingGameState
        self._expanded = 0 # DO NOT CHANGE
        self.heuristicInfo = {} # A dictionary for the heuristic to store
information

    def getStartState(self):
        return self.start

    def isGoalState(self, state):
        return state[1].count() == 0

    def getSuccessors(self, state):
        "Returns successor states, the actions they require, and a cost of 1."
        successors = []
        self._expanded += 1 # DO NOT CHANGE
        for direction in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
Directions.WEST]:
            x,y = state[0]
            dx, dy = Actions.directionToVector(direction)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:
                nextFood = state[1].copy()
                nextFood[nextx][nexty] = False
                successors.append( ( ((nextx, nexty), nextFood), direction, 1) )
        return successors

    def getCostOfActions(self, actions):
        """Returns the cost of a particular sequence of actions.  If those
actions
        include an illegal move, return 999999"""
        x,y= self.getStartState()[0]
        cost = 0
        for action in actions:
            # figure out the next state and see whether it's legal
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]:
                return 999999
            cost += 1
        return cost

class AStarFoodSearchAgent(SearchAgent):
    "A SearchAgent for FoodSearchProblem using A* and your foodHeuristic"
    def __init__(self):
        self.searchFunction = lambda prob: search.aStarSearch(prob,
foodHeuristic)
        self.searchType = FoodSearchProblem

def foodHeuristic(state, problem):
    """
    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.

    If using A* ever finds a solution that is worse uniform cost search finds,
    your heuristic is *not* consistent, and probably not admissible!  On the
    other hand, inadmissible or inconsistent heuristics may find optimal
    solutions, so be careful.

    The state is a tuple ( pacmanPosition, foodGrid ) where foodGrid is a Grid
    (see game.py) of either True or False. You can call foodGrid.asList() to get
    a list of food coordinates instead.

    If you want access to info like walls, capsules, etc., you can query the
    problem.  For example, problem.walls gives you a Grid of where the walls
    are.

    If you want to *store* information to be reused in other calls to the
    heuristic, there is a dictionary called problem.heuristicInfo that you can
    use. For example, if you only want to count the walls once and store that
    value, try: problem.heuristicInfo['wallCount'] = problem.walls.count()
    Subsequent calls to this heuristic can access
    problem.heuristicInfo['wallCount']
    """

    position, food_grid = state
    "*** YOUR CODE HERE ***"
    x, y = position  # 当前位置
    food_list = food_grid.asList()  # 转换为列表
    from_position = []  # pacman到食物的距离
    to_food = []  # 食物两两之间的距离

    # 外层循环计算pacman当前位置到食物
    for food in food_list:
        from_position.append(abs(x - food[0]) + abs(y - food[1]))

        # 内层循环计算两两食物之间的距离
        for food2 in food_list:
            to_food.append(abs(food[0] - food2[0]) + abs(food[1] - food2[1]))

    value = 0  # 初始值为0，所以目标状态的h值就会为0
    value += min(from_position) if len(from_position) else 0  # pacman到食物最小距
离
    value += max(to_food) if len(to_food) else 0  # 食物之间的最大距离
    return value

class ClosestDotSearchAgent(SearchAgent):
    "Search for all food using a sequence of searches"
    def registerInitialState(self, state):
        self.actions = []
        currentState = state
        while(currentState.getFood().count() > 0):
            nextPathSegment = self.findPathToClosestDot(currentState) # The
missing piece
            self.actions += nextPathSegment
            for action in nextPathSegment:
                legal = currentState.getLegalActions()
                if action not in legal:
                    t = (str(action), str(currentState))
                    raise Exception, 'findPathToClosestDot returned an illegal
move: %s!\n%s' % t
                currentState = currentState.generateSuccessor(0, action)
        self.actionIndex = 0
        print 'Path found with cost %d.' % len(self.actions)
    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)

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

class AnyFoodSearchProblem(PositionSearchProblem):
    """
    A search problem for finding a path to any food.

    This search problem is just like the PositionSearchProblem, but has a
    different goal test, which you need to fill in below.  The state space and
    successor function do not need to be changed.

    The class definition above, AnyFoodSearchProblem(PositionSearchProblem),
    inherits the methods of the PositionSearchProblem.

    You can use this search problem to help you fill in the findPathToClosestDot
    method.
    """

    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
        return self.food[x][y]
        "*** YOUR CODE HERE ***"
        """
        foodgrid = self.food
        if(foodgrid[x][y]==True)or(foodgrid.count()==0):
            return True
        util.raiseNotDefined()
        """
```

```
def mazeDistance(point1, point2, gameState):
    """
    Returns the maze distance between any two points, using the search functions
    you have already built. The gameState can be any game state -- Pacman's
    position in that state is ignored.

    Example usage: mazeDistance( (2,4), (5,6), gameState)

    This might be a useful helper function for your ApproximateSearchAgent.
    """
    x1, y1 = point1
    x2, y2 = point2
    walls = gameState.getWalls()
    assert not walls[x1][y1], 'point1 is a wall: ' + str(point1)
    assert not walls[x2][y2], 'point2 is a wall: ' + str(point2)
    prob = PositionSearchProblem(gameState, start=point1, goal=point2,
warn=False, visualize=False)
    return len(search.bfs(prob))
```

Question 4

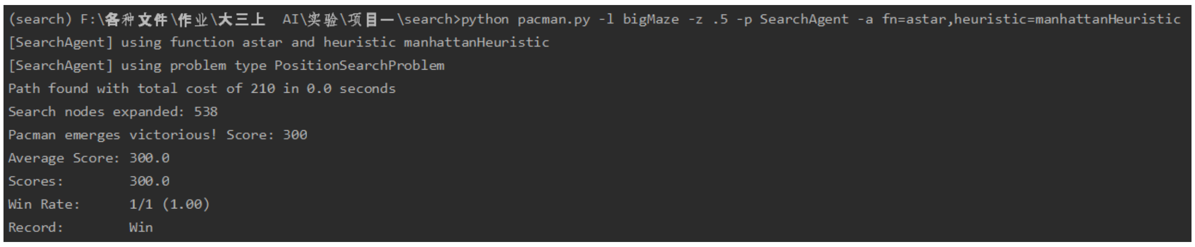
```
class MinimaxAgent(MultiAgentSearchAgent):
    """
      Your minimax agent (question 2)
    """

    def minimax(self, gameState, depth, agent):         # 返回值为（评分，动作）
        action = gameState.getLegalActions(agent)
        if depth > self.depth or len(action) == 0 or gameState.isLose() or
gameState.isWin():
            return self.evaluationFunction(gameState), Directions.STOP
        if agent == 0:                  #max
            max_value = -float('Inf')
            max_action = Directions.STOP
            for a in action:
                state = gameState.generateSuccessor(agent, a)
                new_score = self.minimax(state, depth, 1)[0]
                if new_score > max_value:
                    max_value = new_score
                    max_action = a
            return max_value, max_action
        else:
            min_value = float('Inf')
            for a in action:
                state = gameState.generateSuccessor(agent, a)
                if agent != gameState.getNumAgents()-1:
                    new_score = self.minimax(state, depth, agent+1)[0]
                else:
                    new_score = self.minimax(state, depth+1, 0)[0]
                min_action = Directions.STOP
                if new_score < min_value:
                    min_value = new_score
                    min_action = a
            return min_value, min_action

    def getAction(self, gameState):
        """
          Returns the minimax action from the current gameState using self.depth
          and self.evaluationFunction.

          Here are some method calls that might be useful when implementing
minimax.

          gameState.getLegalActions(agentIndex):
            Returns a list of legal actions for an agent
            agentIndex=0 means Pacman, ghosts are >= 1

          gameState.generateSuccessor(agentIndex, action):
            Returns the successor game state after an agent takes an action

          gameState.getNumAgents():
            Returns the total number of agents in the game
        """

        "*** YOUR CODE HERE ***"
        optimalscore, optimalaction = self.minimax(gameState, 1, 0)
        return optimalaction

        util.raiseNotDefined()
```

Question 5

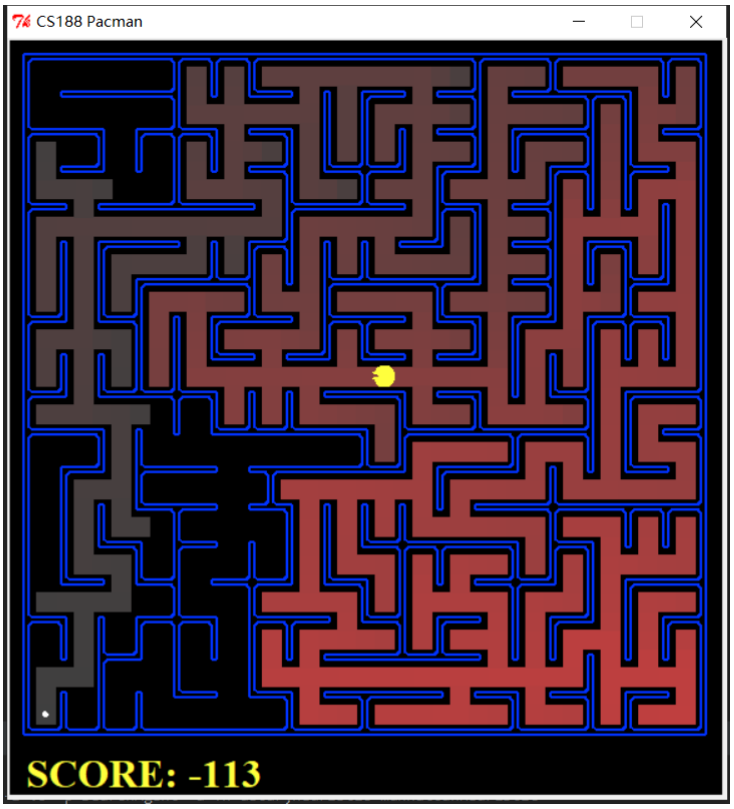
```
class AlphaBetaAgent(MultiAgentSearchAgent):
    """
      Your minimax agent with alpha-beta pruning (question 3)
    """

    def getAction(self, gameState):
        "*** YOUR CODE HERE ***"
        def maxvalue(state, alpha, beta, depth):
            if state.isWin() or state.isLose() or depth >= self.depth:
                return self.evaluationFunction(state), Directions.STOP
            max_score = -float('inf')
            action = Directions.STOP
            for a in state.getLegalActions(0):
                tempbeta = minvalue(state.generateSuccessor(0, a), alpha, beta,
depth, 1)[0]
                if tempbeta > max_score:
                    max_score = tempbeta
                    action = a
                if max_score > beta:
                    return max_score, action        # 剪枝
                alpha = max(alpha, max_score)
            return max_score, action

        def minvalue(state, alpha, beta, depth, num):
            if state.isWin() or state.isLose():       #min need pay attention to
depth
                return self.evaluationFunction(state), Directions.STOP
            min_score = float('inf')
            action = Directions.STOP
            if num != gameState.getNumAgents()-1:
                for a in state.getLegalActions(num):
                    tempalpha = minvalue(state.generateSuccessor(num, a), alpha,
beta, depth, num+1)[0]
                    if tempalpha < min_score:
                        min_score = tempalpha
                        action = a
                    if min_score < alpha:
                        return min_score, action
                    beta = min(beta, min_score)
            else:
                for a in state.getLegalActions(num):
                    tempalpha = maxvalue(state.generateSuccessor(num, a), alpha,
beta, depth+1)[0]
                    if tempalpha < min_score:
                        min_score = tempalpha
                        action = a
                    if min_score < alpha:
                        return min_score, action
                    beta = min(beta, min_score)
            return min_score, action

        optimalscore, optimalaction = maxvalue(gameState, float('-Inf'),
float('Inf'), 0)
        return optimalaction
        util.raiseNotDefined()
```

4.结果展示

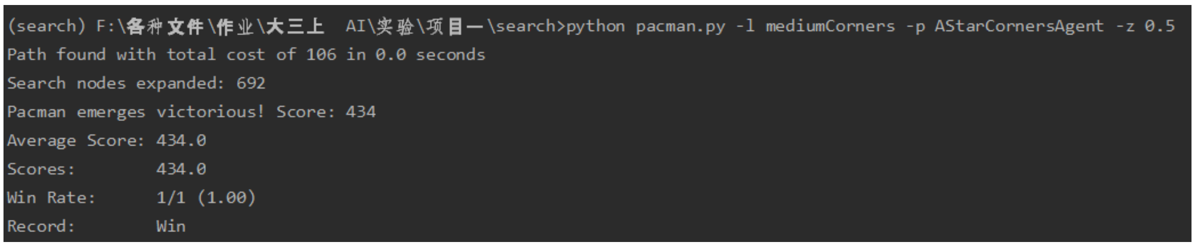
Question 1: A* search



```
(search) F:\各种文件\作业\大三上  AI\实验\项目一\search>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: 538
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

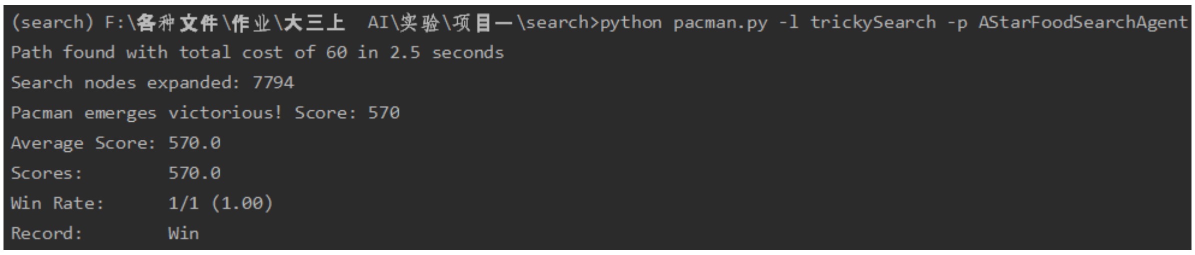


Question 2: Corners Problem: Heuristic



```
(search) F:\各种文件\作业\大三上  AI\实验\项目一\search>python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 106 in 0.0 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
```

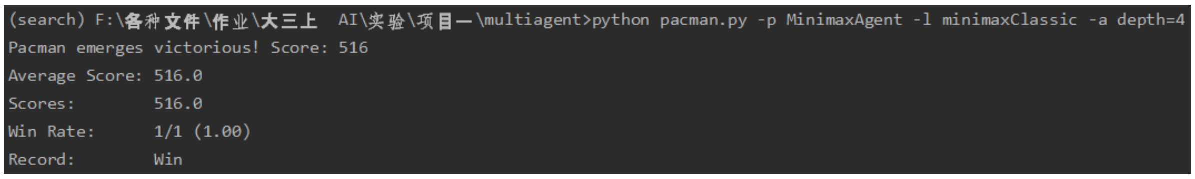
Question 3: Eating All The Dots



```
(search) F:\各种文件\作业\大三上  AI\实验\项目一\search>python pacman.py -l trickySearch -p AStarFoodSearchAgent
Path found with total cost of 60 in 2.5 seconds
Search nodes expanded: 7794
Pacman emerges victorious! Score: 570
Average Score: 570.0
Scores:        570.0
Win Rate:      1/1 (1.00)
Record:        Win
```

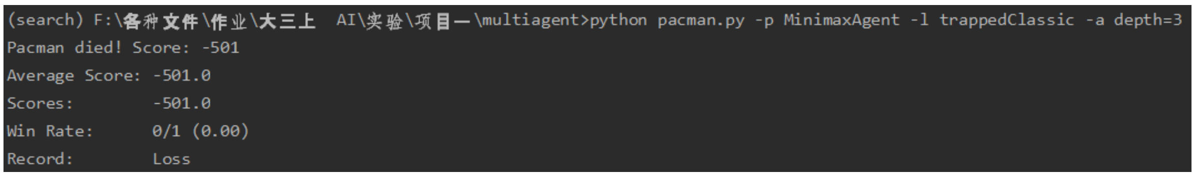
Question 4: Minimax

```
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
```



```
(search) F:\各种文件\作业\大三上  AI\实验\项目一\multiagent>python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
Pacman emerges victorious! Score: 516
Average Score: 516.0
Scores:        516.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
```



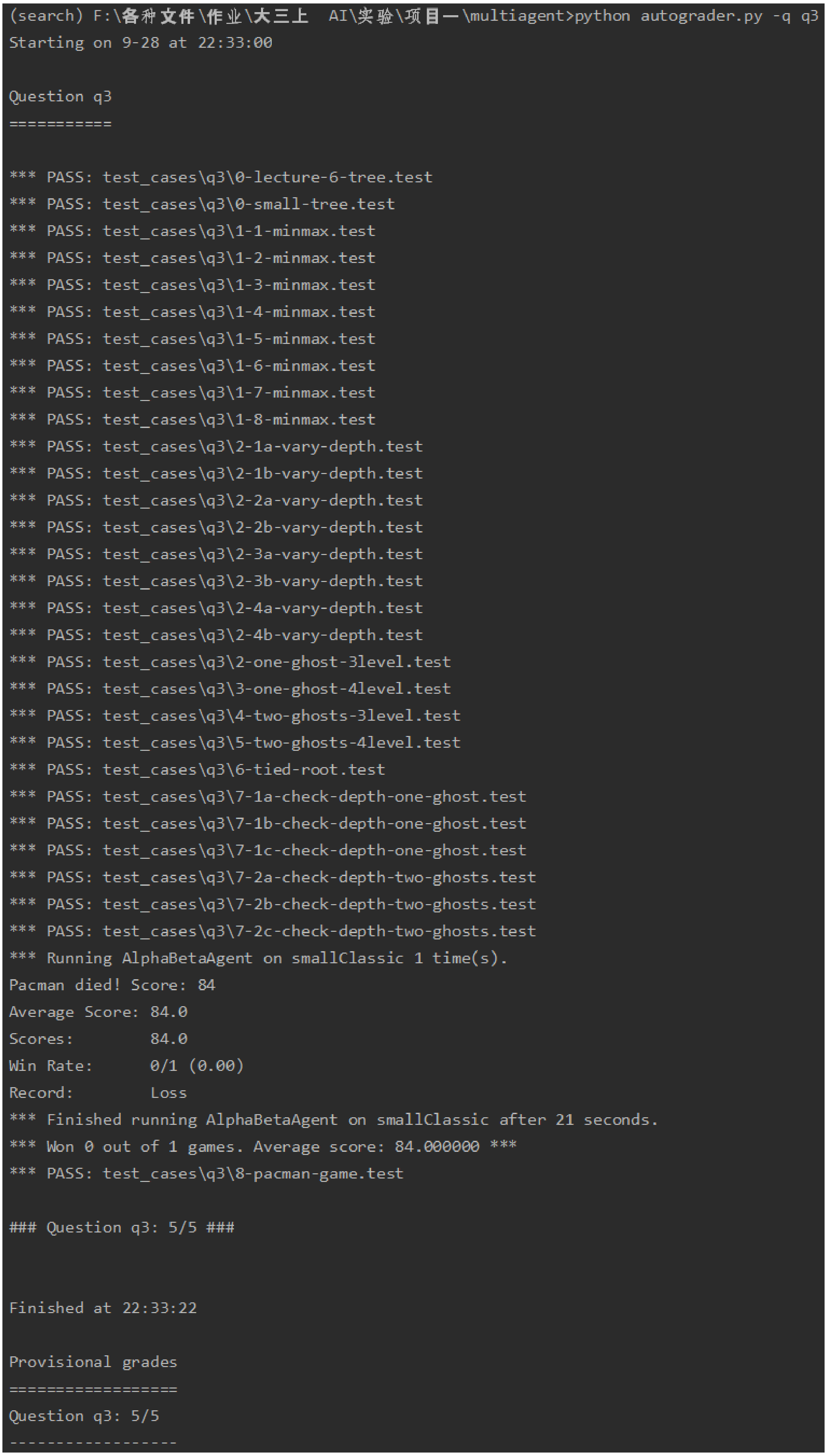
```
(search) F:\各种文件\作业\大三上  AI\实验\项目一\multiagent>python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
Pacman died! Score: -501
Average Score: -501.0
Scores:        -501.0
Win Rate:      0/1 (0.00)
Record:        Loss
```

```
python autograder.py -q q2 --no-graphics
```

```
*** PASS: test_cases\q2\0-lecture-6-tree.test
*** PASS: test_cases\q2\0-small-tree.test
*** PASS: test_cases\q2\1-1-minmax.test
*** PASS: test_cases\q2\1-2-minmax.test
*** PASS: test_cases\q2\1-3-minmax.test
*** PASS: test_cases\q2\1-4-minmax.test
*** PASS: test_cases\q2\1-5-minmax.test
*** PASS: test_cases\q2\1-6-minmax.test
*** PASS: test_cases\q2\1-7-minmax.test
*** PASS: test_cases\q2\1-8-minmax.test
*** PASS: test_cases\q2\2-1a-vary-depth.test
*** PASS: test_cases\q2\2-1b-vary-depth.test
*** PASS: test_cases\q2\2-2a-vary-depth.test
*** PASS: test_cases\q2\2-2b-vary-depth.test
*** PASS: test_cases\q2\2-3a-vary-depth.test
*** PASS: test_cases\q2\2-3b-vary-depth.test
*** PASS: test_cases\q2\2-4a-vary-depth.test
*** PASS: test_cases\q2\2-4b-vary-depth.test
*** PASS: test_cases\q2\2-one-ghost-3level.test
*** PASS: test_cases\q2\3-one-ghost-4level.test
*** PASS: test_cases\q2\4-two-ghosts-3level.test
*** PASS: test_cases\q2\5-two-ghosts-4level.test
*** PASS: test_cases\q2\6-tied-root.test
*** PASS: test_cases\q2\7-1a-check-depth-one-ghost.test
*** PASS: test_cases\q2\7-1b-check-depth-one-ghost.test
*** PASS: test_cases\q2\7-1c-check-depth-one-ghost.test
*** PASS: test_cases\q2\7-2a-check-depth-two-ghosts.test
*** PASS: test_cases\q2\7-2b-check-depth-two-ghosts.test
*** PASS: test_cases\q2\7-2c-check-depth-two-ghosts.test
*** Running MinimaxAgent on smallClassic 1 time(s).
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
*** Finished running MinimaxAgent on smallClassic after 1 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\q2\8-pacman-game.test

### Question q2: 5/5 ###


Finished at 17:25:21

Provisional grades
==================
Question q2: 5/5
------------------
Total: 5/5
```

Question 5: *α-β* Pruning

```
python autograder.py -q q3
```

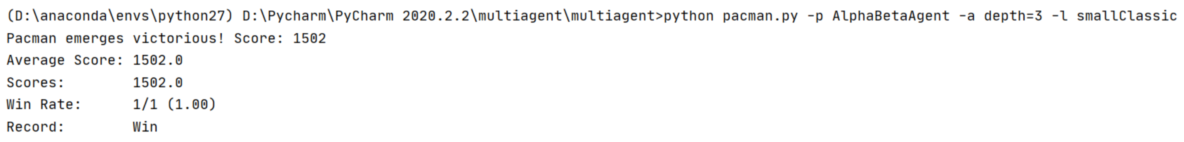

```
(search) F:\各种文件\作业\大三上  AI\实验\项目一\multiagent>python autograder.py -q q3
Starting on 9-28 at 22:33:00

Question q3
===========

*** PASS: test_cases\q3\0-lecture-6-tree.test
*** PASS: test_cases\q3\0-small-tree.test
*** PASS: test_cases\q3\1-1-minmax.test
*** PASS: test_cases\q3\1-2-minmax.test
*** PASS: test_cases\q3\1-3-minmax.test
*** PASS: test_cases\q3\1-4-minmax.test
*** PASS: test_cases\q3\1-5-minmax.test
*** PASS: test_cases\q3\1-6-minmax.test
*** PASS: test_cases\q3\1-7-minmax.test
*** PASS: test_cases\q3\1-8-minmax.test
*** PASS: test_cases\q3\2-1a-vary-depth.test
*** PASS: test_cases\q3\2-1b-vary-depth.test
*** PASS: test_cases\q3\2-2a-vary-depth.test
*** PASS: test_cases\q3\2-2b-vary-depth.test
*** PASS: test_cases\q3\2-3a-vary-depth.test
*** PASS: test_cases\q3\2-3b-vary-depth.test
*** PASS: test_cases\q3\2-4a-vary-depth.test
*** PASS: test_cases\q3\2-4b-vary-depth.test
*** PASS: test_cases\q3\2-one-ghost-3level.test
*** PASS: test_cases\q3\3-one-ghost-4level.test
*** PASS: test_cases\q3\4-two-ghosts-3level.test
*** PASS: test_cases\q3\5-two-ghosts-4level.test
*** PASS: test_cases\q3\6-tied-root.test
*** PASS: test_cases\q3\7-1a-check-depth-one-ghost.test
*** PASS: test_cases\q3\7-1b-check-depth-one-ghost.test
*** PASS: test_cases\q3\7-1c-check-depth-one-ghost.test
*** PASS: test_cases\q3\7-2a-check-depth-two-ghosts.test
*** PASS: test_cases\q3\7-2b-check-depth-two-ghosts.test
*** PASS: test_cases\q3\7-2c-check-depth-two-ghosts.test
*** Running AlphaBetaAgent on smallClassic 1 time(s).
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
*** Finished running AlphaBetaAgent on smallClassic after 21 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\q3\8-pacman-game.test

### Question q3: 5/5 ###


Finished at 22:33:22

Provisional grades
==================
Question q3: 5/5
------------------
```

```
Total: 5/5
```

可以看到，两条指令执行的结果一样，证明了minimax与$\alpha-\beta$ pruning的一致性。

```
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
```

下图为吃豆人胜利的截图，当然并不是每把都能胜利，而且在我们组内的2台电脑上测试胜率差别还是蛮大的。



```
(D:\anaconda\envs\python27) D:\Pycharm\PyCharm 2020.2.2\multiagent\multiagent>python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
Pacman emerges victorious! Score: 1502
Average Score: 1502.0
Scores:        1502.0
Win Rate:      1/1 (1.00)
Record:        Win
```

5.结果分析

1. Search in Pacman

- 在本实验中，我们在解决Q1时使用的启发式函数是`当前节点到目标节点的曼哈顿距离`，即两点横坐标之差的绝对值和纵坐标之差的绝对值的和，其显然满足一致性，且存在将吃豆人导向距离目标更近的节点的趋势，吃豆人将逐渐走向目标。 解决Q2时使用的启发式函数是`当前节点到最近角落的距离与未被探索的相邻角落间的距离之和`。更形象地说明，假设当前吃豆人位置为A_1，未被探索的角落为C_1，C_2，C_3，且C_1与C_2相邻，C_2与C_3相邻，其中C_1为距离A_1最近的角落。则启发式函数的返回值为$d(A_1 C_1) + d(C_1 C_2) + d(C_2 C_3)$，其中d()返回两节点的曼哈顿距离。可以发现，在吃豆人位置一定时，`未被探索的相邻角落间的距离`是一定的，则起到决定性作用的是`当前节点到最近角落的距离`，显然满足一致性（与Q1中性质一样），且存在将吃豆人导向当前状态下最近角落的趋势，则吃豆人将依次遍历最近的角落。 解决Q3时使用的启发式函数是`当前节点到最近食物的距离与最近食物到距离最近食物最远的食物的距离之和`。该启发式函数是吃豆人遍历所有食物走过距离总和的近似，根据三角不等式可知其满足一致性，且具有将吃豆人导向可以更快速遍历所有食物的路径的趋势，故吃豆人将依次遍历所有食物。 综上所述，三个启发式函数的基本思想都是曼哈顿距离，但由于决定性因素不同而有了不同的作用。

2. Multi-Agent Pacman

- 在minimax算法里，我们要探索整个`game tree`，当深度为d，每个点有b个合法有效的动作时，则算法的时间复杂度为$O(b^m)$，而使用$\alpha-\beta$ pruning算法后，搜索时间则减短了许多。本次$\alpha-\beta$ pruning的主要难点是在于有多个MIN Agents，在实现的时候要考虑临界条件：当进行至最后一个MIN Agent时则要更新深度，否则的话还是在同一层进行搜索，即Agents的index更新，而深度不更新。
- 在运行吃豆人的时候，的确可以看到在使用$\alpha-\beta$ pruning算法时，每步的移动所需的时间要比minimax算法短（换句话说，可以看出来使用$\alpha-\beta$ pruning算法的吃豆人的移动速度更快）。

6. Experimental experience

本次Project加深了我们对于这一段时间以来所学习的内容的理解，可以说把我们近来所学的算法都实现了一遍。实现算法的原理其实并不难，但是在实现的过程中遇到了一些问题，而这些问题都是与Python自身相关的。在完成本次任务的过程里，我们了解到许多相关Python的数据结构的知识，比如可哈希的与不可哈希的；还有就是`tuple`的使用，之前对于`tuple`还是蛮陌生的……这些都是在发现bug以后通过上网查找资料才学习到的，这也反映我们对于Python的掌握还不够，未来的时间里还要多多学习Python。

7. Appendix

`search.py`里关于其他搜索算法的实现：

```
def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:

    print "Start:", problem.getStartState()
    print "Is the start a goal?", problem.isGoalState(problem.getStartState())
    print "Start's successors:", problem.getSuccessors(problem.getStartState())
    """
    "*** YOUR CODE HERE ***"
    startState = problem.getStartState()
    startNode = (startState, [], 0)

    frontier = util.Stack()
    frontier.push(startNode)
    explored = set()

    while not frontier.isEmpty():
        cur_node = frontier.pop()        # current_what
        if problem.isGoalState(cur_node[0]):
            return cur_node[1]
        if cur_node[0] not in explored:
            explored.add(cur_node[0])
            for isuc in problem.getSuccessors(cur_node[0]):
                suc_state, suc_action, suc_cost = isuc    # successor_what
                new_action = cur_node[1] + [suc_action]
                new_cost = cur_node[2] + suc_cost
                new_node = (suc_state, new_action, new_cost)
                frontier.push(new_node)
    return []
    util.raiseNotDefined()

def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    startState = problem.getStartState()
    startNode = (startState, [], 0)

    frontier = util.Queue()
    frontier.push(startNode)
    explored = set()

    while not frontier.isEmpty():
        cur_node= frontier.pop()  # current_what
        if problem.isGoalState(cur_node[0]):
            return cur_node[1]
        if cur_node[0] not in explored:
            explored.add(cur_node[0])
            for isuc in problem.getSuccessors(cur_node[0]):
                suc_state, suc_action, suc_cost = isuc  # successor_what
                new_action = cur_node[1] + [suc_action]
                new_cost = cur_node[2] + suc_cost
                new_node = (suc_state, new_action, new_cost)
                frontier.push(new_node)
    return []
    util.raiseNotDefined()

def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    class Node:
        def __init__(self, state, action, g):
            self.state = state
            self.action = action
            self.cost = g

        def get_g(self):
            return self.cost

    startState = problem.getStartState()
    startNode = Node(startState, [], 0)

    frontier = Queue.PriorityQueue()
    frontier.put((startNode.get_g(), startNode))
    explored = set()

    while not frontier.empty():
        (cur_g, cur_node) = frontier.get()        # current_what
        if problem.isGoalState(cur_node.state):
            return cur_node.action
        if cur_node.state not in explored:
            explored.add(tuple(cur_node.state))
            for isuc in problem.getSuccessors(cur_node.state):
                suc_state, suc_action, suc_cost = isuc    # successor_what
                new_action = cur_node.action + [suc_action]
                new_cost = cur_node.cost + suc_cost
                new_node = Node(suc_state, new_action, new_cost)
                frontier.put((new_node.get_g(), new_node))
    return []

    util.raiseNotDefined()
```