P01 Pacman Game

学 号	姓 名	专业 (方向)
18340052	何 泽	计算机科学与技术（超级计算方向）
18340032	邓俊锋	计算机科学与技术（大数据与人工智能方向）

目录

1.Idea of A* Algorithm

- $A*$ 是一种启发式搜索，与传统的搜索相比，它对每一个搜索的位置进行评估，得到最好的位置，再从这个位置进行搜索直到目标。这种方式可以省略大量无畏的搜索路径，提

高了效率。

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

- $Min - Max$

  极小极大算法用于博弈树搜索，即游戏一方想要极大化最终结果，另外一方想要极小化最终结果，且两者交替执行策略。极小极大算法对博弈树执行完整的深度优先探索，递归算法自上而下一直前进到树的叶结点，然后随着递归回溯通过搜索树把极小极大值回传。若当前节点为极大方，则取子节点中最大的回传给父节点；反之回传最小的。

- $\alpha - \beta\ pruning$

  极小极大值搜索的问题是必须检查的游戏状态的数目是随着博弈的进行呈指数级增长，虽然无法让它变成多项式级，但是仍可以删掉很多不需要进行搜索的节点。$alpha - beta$ 剪枝可以应用于任何深度的树，很多情况下可以剪裁整个子树，而不仅仅是剪裁叶结点。一般原则是：考虑在树中某处的结点n, 选手选择移动到该结点。如果选手在n的父结点或者更上层的任何选择点有更好的选择m，那么在实际的博弈中就永远不会到达n。所以一旦发现关于n的足够信息（通过检查它的某些后代）， 能够得到上述结论，我们就可以剪裁它，即

  - 只要当前Max结点的值 $\geq$ 祖先某一Min结点的值，就可以在该Max结点上做α剪枝
  - 只要当前Min结点的值 $\leq$ 祖先某一Max结点的值，就可以在该Min结点上做β剪枝

3. Codes

Question 1

```python
#search.py

def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    q = util.Queue()
    beg = problem.getStartState()
```

```
    q.push((beg, []))
    visited = []
    while not q.isEmpty():
        cur, actions = q.pop()
        if cur in visited:
            continue
        visited.append(cur)
        if problem.isGoalState(cur):
            break
        for suc in problem.getSuccessors(cur):
            q.push((suc[0], actions + [suc[1]]))
    return actions

def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    q = util.PriorityQueue()
    beg = problem.getStartState()
    q.push((0, beg, []), 0)
    visited = []
    while not q.isEmpty():
        cost, cur, actions = q.pop()
        if cur in visited:
            continue
        visited.append(cur)
        if problem.isGoalState(cur):
            break
        for suc in problem.getSuccessors(cur):
            priority = cost + suc[2] + heuristic(suc[0], problem)
            q.push((cost + suc[2], suc[0], actions + [suc[1]]), priority)
    return actions
```

Question 2

```
#searchAgents.py

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
        self.startingGameState = startingGameState
        visited = [0,0,0,0]
        for (i,corner) in enumerate(self.corners):
            if self.startingPosition == corner:
                visited[i] = 1
        self.startState = (self.startingPosition,visited)

    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        return self.startState

    def isGoalState(self, state):
        """
```

```
38          Returns whether this search state is a goal state of the
    problem.
39          """
40          if state[1] == [1,1,1,1]:
41              return True
42          return False
43
44      def getSuccessors(self, state):
45          """
46          Returns successor states, the actions they require, and a
    cost of 1.
47           As noted in search.py:
48              For a given state, this should return a list of
    triples, (successor,
49              action, stepCost), where 'successor' is a successor
    to the current
50              state, 'action' is the action required to get there,
    and 'stepCost'
51              is the incremental cost of expanding to that
    successor
52          """
53
54          successors = []
55          for action in [Directions.NORTH, Directions.SOUTH,
    Directions.EAST, Directions.WEST]:
56              # Add a successor state to the successor list if the
    action is legal
57              # Here's a code snippet for figuring out whether a
    new position hits a wall:
58              #   x,y = currentPosition
59              #   dx, dy = Actions.directionToVector(action)
60              #   nextx, nexty = int(x + dx), int(y + dy)
61              #   hitsWall = self.walls[nextx][nexty]
62              x, y = state[0]
63              dx, dy = Actions.directionToVector(action)
64              nextx, nexty = int(x + dx), int(y + dy)
65              if not self.walls[nextx][nexty]:
66                  next = [item for item in state[1]]
67                  if (nextx,nexty) in self.corners:
```

```
68                      i = self.corners.index((nextx,nexty))
69                      next[i] = 1
70                  nextState = ((nextx,nexty),next)
71                  successors.append((nextState,action,1))
72          self._expanded += 1 # DO NOT CHANGE
73          return successors
74
75      def getCostOfActions(self, actions):
76          """
77          Returns the cost of a particular sequence of actions.  If
    those actions
78          include an illegal move, return 999999.  This is
    implemented for you.
79          """
80          if actions == None: return 999999
81          x,y= self.startingPosition
82          for action in actions:
83              dx, dy = Actions.directionToVector(action)
84              x, y = int(x + dx), int(y + dy)
85              if self.walls[x][y]: return 999999
86          return len(actions)
87
88
89  def cornersHeuristic(state, problem):
90      """
91      A heuristic for the CornersProblem that you defined.
92
93        state:   The current search state
94                 (a data structure you chose in your search
    problem)
95
96        problem: The CornersProblem instance for this layout.
97
98      This function should always return a number that is a lower
    bound on the
99      shortest path from the state to a goal of the problem; i.e.
    it should be
100     admissible (as well as consistent).
101     """
```

```
    corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid (game.py)
    top, right = walls.height - 2, walls.width - 2
    pos = state[0]
    cornerFlag = state[1]
    remain = []
    for i, corner in enumerate(corners):
        if not cornerFlag[i]:
            remain.append(corner)
    if len(remain) == 0:
        return 0
    return max(map(lambda x: mazeDistance(pos, x, problem.startingGameState), remain))
```

Question 3

```
#searchAgents.py

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, foodGrid = state
    foodLst = foodGrid.asList()

    if len(foodLst) == 0:
        return 0
    ans = max(map(lambda x:mazeDistance(position,x,problem.startingGameState), foodLst))
    return ans
```

Question 4

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

```
5
6       def getAction(self, gameState):
7
8           def max_agent(state, depth):
9               if state.isWin() or state.isLose() or depth ==
    self.depth:
10                  return self.evaluationFunction(state), None
11              score = float('-Inf')
12              max_action = ''
13              for action in state.getLegalActions(0):
14                  tmp_score = min_agent(state.generateSuccessor(0,
    action), depth, 1)[0]
15                  if score < tmp_score:
16                      score = tmp_score
17                      max_action = action
18              return score, max_action
19
20          def min_agent(state, depth, ghostNum):
21              if state.isWin() or state.isLose():
22                  return self.evaluationFunction(state), None
23              score = float('Inf')
24              min_action = ''
25              if ghostNum == gameState.getNumAgents() - 1:
26                  for action in state.getLegalActions(ghostNum):
27                      tmp_score =
    max_agent(state.generateSuccessor(ghostNum, action), depth + 1)[0]
28                      if score > tmp_score:
29                          score = tmp_score
30                          min_action = action
31              else:
32                  for action in state.getLegalActions(ghostNum):
33                      tmp_score =
    min_agent(state.generateSuccessor(ghostNum, action), depth,
    ghostNum + 1)[0]
34                      if score > tmp_score:
35                          score = tmp_score
36                          min_action = action
37              return score, min_action
38
```

```
        maxiscore, bestAction = max_agent(gameState, 0)
        return bestAction
        util.raiseNotDefined()
```

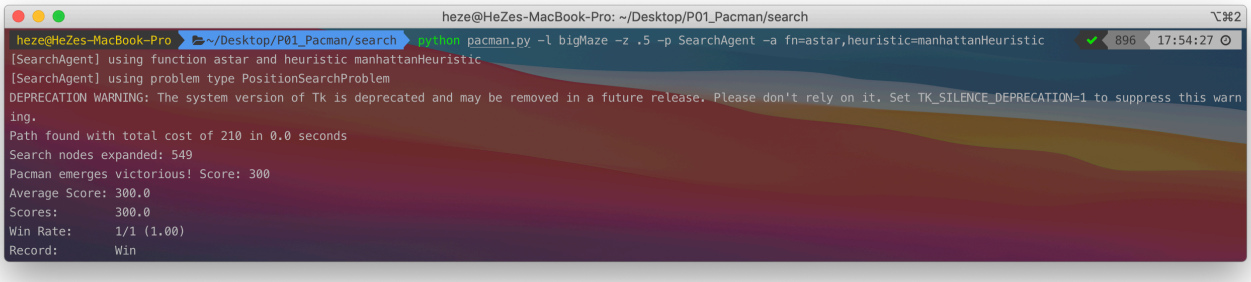
Question 5

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

    def getAction(self, gameState):
        """
            Returns the minimax action using self.depth and self.evaluationFunction
        """
        def max_agent(state, depth, alpha, beta):
            if state.isWin() or state.isLose() or depth == self.depth:
                return self.evaluationFunction(state), None
            score = float('-Inf')
            max_action = ''
            for action in state.getLegalActions(0):
                tmp_score = min_agent(state.generateSuccessor(0, action), depth, 1, alpha, beta)[0]
                if score < tmp_score:
                    score = tmp_score
                    max_action = action
                if score > beta:
                    return score, max_action
                alpha = max(alpha, score)
            return score, max_action

        def min_agent(state, depth, ghostNum, alpha, beta):
            if state.isWin() or state.isLose():
                return self.evaluationFunction(state), None
            score = float('Inf')
            min_action = ''
```

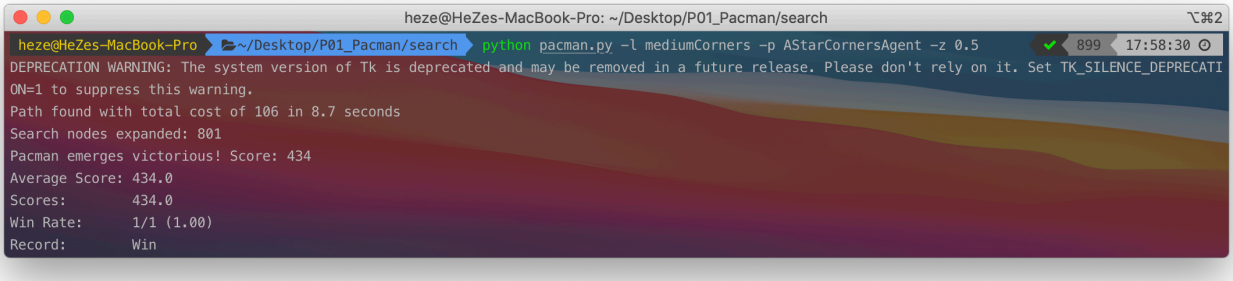
```
            # Last ghost, go to next max ply
            if ghostNum == gameState.getNumAgents() - 1:
                for action in state.getLegalActions(ghostNum):
                    tmp_score = max_agent(state.generateSuccessor(ghostNum, action), depth + 1, alpha, beta)[0]
                    if score > tmp_score:
                        score = tmp_score
                        min_action = action
                    if score < alpha:
                        return score, min_action
                    beta = min(beta, score)
            else:
                for action in state.getLegalActions(ghostNum):
                    tmp_score = min_agent(state.generateSuccessor(ghostNum, action), depth, ghostNum + 1, alpha, beta)[0]
                    if score > tmp_score:
                        score = tmp_score
                        min_action = action
                    if score < alpha:
                        return score, min_action
                    beta = min(beta, score)
            return score, min_action

        maxiscore, bestAction = max_agent(gameState, 0, float('-Inf'), float('Inf'))
        return bestAction
        util.raiseNotDefined()
```

4.结果展示

Question 1

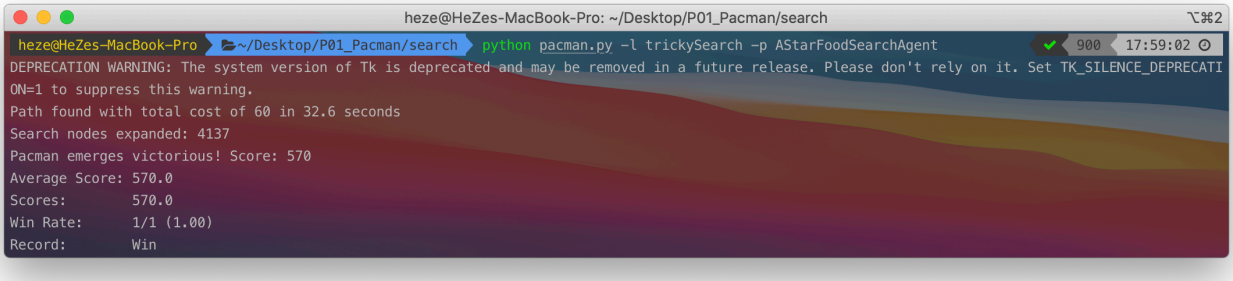


Question 2



- 共搜索 434 个节点

Question 3



- 共搜索 4137 个节点

Question 4

- 通过所有测试

Question 5

- 通过所有测试

5.结果分析

(1) Search in Pacman

- 在 `Question 1` 里，启发式函数是离目标点的曼哈顿距离。启发函数十分简单，离目标越近，函数值越小；到达终点后启发式函数为0而；且比实际需要的cost小，所以能引导吃豆人走到目标点。
- 在 `Question 2` 里，我的启发式函数参考了 `Question 1` 的启发式函数。我的启发式函数是距离未访问目标点（Corners）迷宫距离（在 `search.py` 中用 `bfs` 实现）的最大值。每到达一个角落，启发式函数都会减少；到达终点后启发式函数为0；而且比实际需要的cost小，所以能引导吃豆人走完四个角落。虽然计算启发函数很慢，但是能大大减小搜索空间。
- 在 `Question 3` 里，启发式函数与 `Question 2` 类似。我的启发式函数是距离未吃的点的迷宫距离（在 `search.py` 中用 `bfs` 实现）的最大值。每吃掉了一个点，启发式函数都会减少；到达终点后启发式函数为0；而且比实际需要的cost小。所以能引导吃豆人吃完全部点。

(2) Multi-Agent Pacman

可以看出使用$\alpha-\beta$剪枝之后在运行相同深度的时候更快，比如分别运行如下2条命令：

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

在运行的时候 `MinimaxAgent` 差不多是2～3秒移动一次，而 `AlphaBetaAgent` 差不多1秒就移动一次，可以明显感受到使用 `AlphaBetaAgent` 之后每一次移动需要计算的时间更短，移动过程表现地更为“流畅”，这便是剪枝的作用。

6.Experimental experience

这是人工智能的第一次项目作业，完成起来和之前的感觉完全不同，因为之前的都是一个文件即可完成，但是这次的项目有好多文件，这就要考虑文件之间的逻辑关系，在完成之前需要提前了解整个项目的代码框架以及涉及到的文件之间的关系 。

除此之外，在 `Multi-Agent Pacman` 中，虽然完成了实验任务，但是在实际运行过程中还是可以看出很多问题的，比如经常停在某一个位置不动，直到怪物靠近的时候才会继续走，即使在旁边就有食物还没有怪物这种情况下有时也不会动；此外基本上每次都会输，都会被怪物吃掉。这些都代表现在的算法还是有一些缺陷的，可目前的算法已经是费了很大的力才写出来的，可见想要实现一个完美的算法是多么的难。

总之，通过自己的实现，这一项目加深了我对搜索、$Min - Max$ 以及剪枝算法的理解。