人工智能 LAB1 实验报告

一、实验目标

实现 BFS，A*算法，完成吃豆人寻找食物的静态搜索。实现 minimax 算法和 alpha-beta 剪枝，完成吃豆人与对手的动态博弈并减少搜索空间。

二、实验环境和工具

实验环境：Ubuntu 16.04 Python3.6.5
实验工具：PyCharm

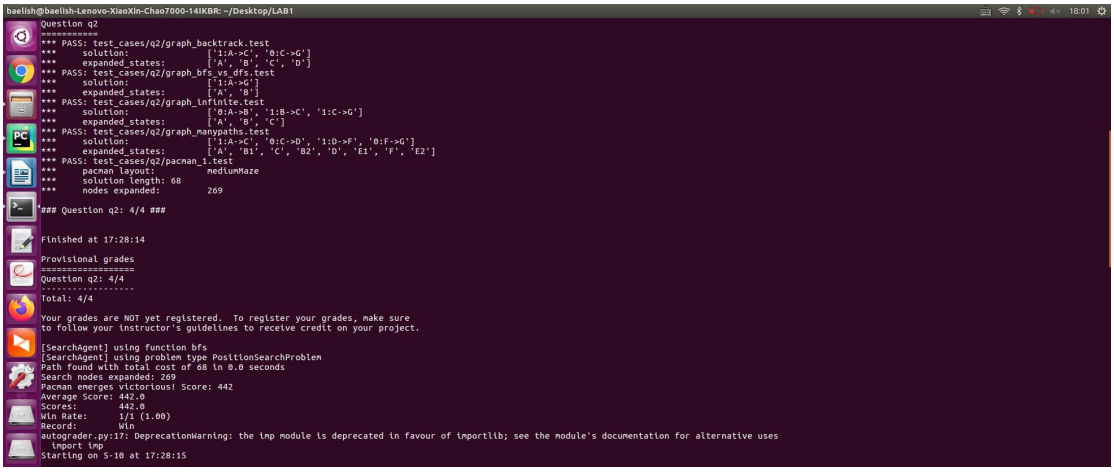
三、实验内容和过程

（1）BFS

考虑到 BFS 即广度优先搜索，区别于 DFS 深度优先搜索，BFS 需要维护一个存储形式，使得状态先于子状态被遍历，容易想到是一个队列，故只需要把 DFS 的栈 Stack 改为队列 Queue 即可。代码如下：

```
def myBreadthFirstSearch(problem):
    visited = {}
    frontier = util.Queue()
    frontier.push((problem.getStartState(), None))

    while not frontier.isEmpty():
        state, prev_state = frontier.pop()

        if problem.isGoalState(state):
            solution = [state]
            while prev_state != None:
                solution.append(prev_state)
                prev_state = visited[prev_state]
            return solution[::-1]

        if state not in visited:
            visited[state] = prev_state

            for next_state, step_cost in problem.getChildren(state):
                frontier.push((next_state, state))

    return []
```

测试结果如下：

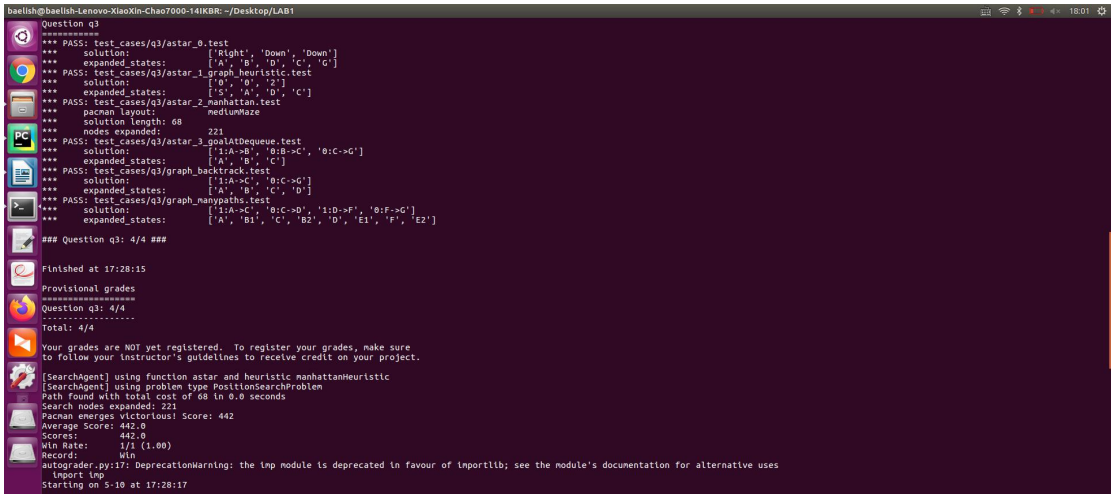


（2）A*

A*算法需要依照最小的启发值选择搜索子状态，是一种高级的广度优先搜索，容易想到用优先队列存储状态空间，按照启发值选择搜索状态。代码如下：（其中 f_state 是开始状态到当前状态的实际值）

```
def myAStarSearch(problem, heuristic):
    visited = {}
    frontier = util.PriorityQueue()
    frontier.push((problem.getStartState(), None, 0),
heuristic(problem.getStartState()))

    while not frontier.isEmpty():
        state, prev_state, f_state = frontier.pop()

        if problem.isGoalState(state):
            solution = [state]
            while prev_state != None:
                solution.append(prev_state)
                prev_state = visited[prev_state]
            return solution[::-1]

        if state not in visited:
            visited[state] = prev_state

            for next_state, step_cost in problem.getChildren(state):
                frontier.push((next_state, state, f_state+step_cost),
f_state+step_cost+heuristic(next_state))

    return []
```

测试结果如下：

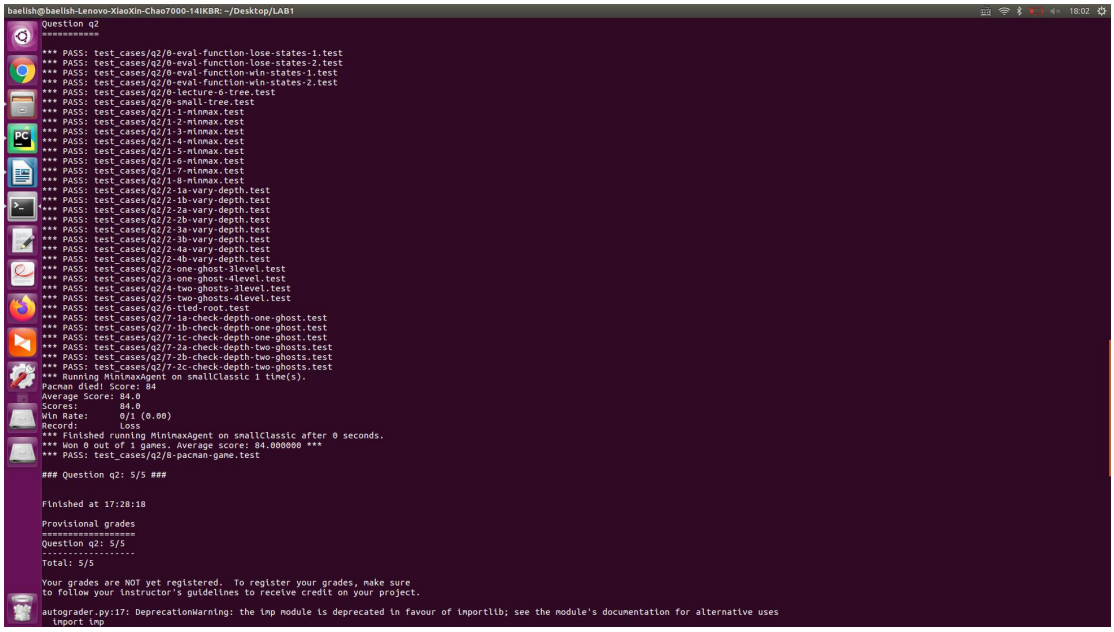


（3）Minimax

考虑预估步数为 depth，写一个递归函数 minimax(state,depth)，每次从对手转移执行权到吃豆人时，把 depth-1 送入下次递归，如果是吃豆人执行操作，就求相对于吃豆人的状态最大值，反之求相对于吃豆人的状态最小值，把对应状态作为本次操作的子状态。代码如下：

```
def minimax(self, state, depth):
    if state.isTerminated() or depth == 0:
        return None, state.evaluateScore()

    best_state, best_score = None, -float('inf') if state.isMe() else float('inf')

    for child in state.getChildren():

        if child.isMe():
            _, score = self.minimax(child, depth - 1)
        else:
            _, score = self.minimax(child, depth)

        if state.isMe() and score > best_score:
            best_score = score
            best_state = child
        elif not state.isMe() and score < best_score:
            best_score = score
            best_state = child

    return best_state, best_score
```

测试结果如下：



（4）minimax 的 alpha-beta 剪枝

在 minimax 算法上做剪枝，记录对手执行不同操作时的最大状态值 alpha，如果在某次轮到吃豆人执行造作时，不管执行何种操作，获得状态值永远小于 alpha，那么这个状态下的搜索子空间的状态值不可能比 alpha 大，对最终结果没有影响，故可以剪枝。beta 同理，代码如下：

```
def alphabeta(self, state, depth, alpha, beta):
    if state.isTerminated() or depth == 0:
        return None, state.evaluateScore()

    best_state, best_score = None, -float('inf') if state.isMe() else float('inf')

    for child in state.getChildren():

        if child.isMe():
            _, score = self.alphabeta(child, depth - 1, alpha, beta)
        else:
            _, score = self.alphabeta(child, depth, alpha, beta)

        if state.isMe():
            if score > beta:
                return child, score
            if score > alpha:
                alpha = score
            if score > best_score:
```

```
                best_score = score
                best_state = child
        else:
            if score < alpha:
                return child, score
            if score < beta:
                beta = score
            if score < best_score:
                best_score = score
                best_state = child

    return best_state, best_score

def getNextState(self, state):
    best_state, _ = self.alphabeta(state, self.depth, -float('inf'), float('inf'))
    return best_state
```

测试结果如下：