实验二

实验描述

本次实验难度较低，需要写的代码总计约100行以内。

一. 环境配置

实验推荐使用 Python 3.6 版本（其他 Python 3.x 版本大概率也可以运行），推荐使用 anaconda 来管理 Python 环境，推荐使用 Linux，因为 Linux 系统下测试只需要在命令行中运行 `./test.sh`。Linux 可以使用虚拟机安装，也可以装双系统等。在 Windows 下可以在命令行中手动逐行输入 `./test.sh` 中的代码进行测试。正确代码应该 PASS 所有的测试。如果你实现的代码有误，请善用报错信息和 `print()` 函数。

测试效果部分如图：

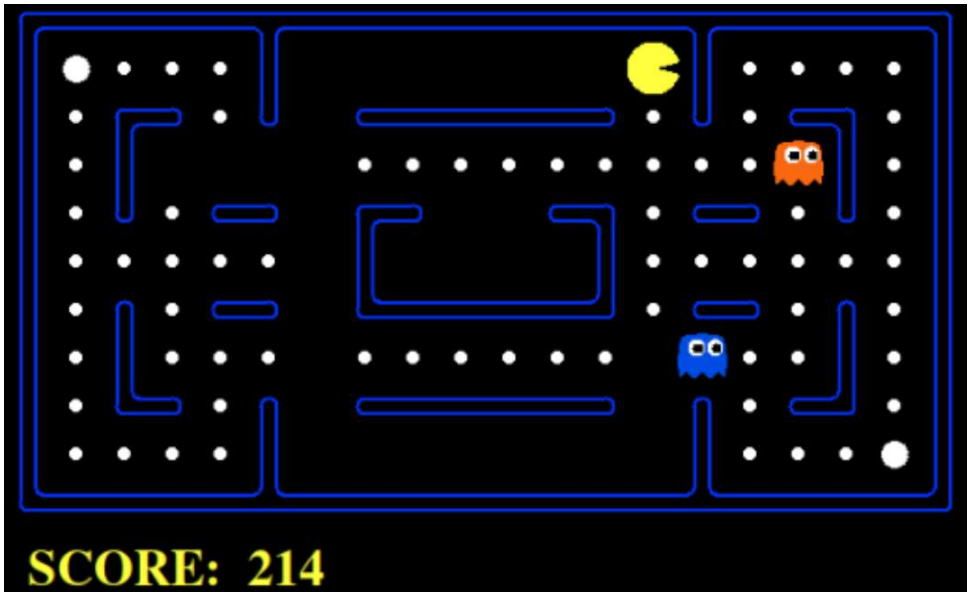
```
Question q3
===========

*** PASS: test_cases/q3/0-eval-function-lose-states-1.test
*** PASS: test_cases/q3/0-eval-function-lose-states-2.test
*** PASS: test_cases/q3/0-eval-function-win-states-1.test
*** PASS: test_cases/q3/0-eval-function-win-states-2.test
*** 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
```

二. 整体框架

本实验利用吃豆人游戏框架，玩家操控一个角色（吃豆人），在避开 NPC （鬼）的条件下吃掉（触碰）尽可能多的小球（豆）。如果你想更好的了解游戏规则，体验一下实验的乐趣，可以先玩一局吃豆人。在

命令行中输入以下命令即可。

```
cd search
python pacman.py
```



本次实验**只需改动并提交 mylmpl.py 文件**，完成实验无需阅读其他代码文件。**请勿在 mylmpl.py 中增加 `import` 其他模块**，否则会造成测试失败。
本次实验需要你按照给定框架补完4个算法，分为 Search 和 Multi-Agent 两类。具体而言，Search 的目标是吃豆人
在没有鬼的干扰下寻找食物；Multi-Agent 的目标是在鬼的干扰下选择最优行动去吃食物。在 Search 中需要你实现 BFS 算法和 A* 算法，Multi-Agent 类需要实现 minimax 算法和 alpha-beta 剪枝。

三. Search

你需要实现 BFS 算法和 A* 算法。你只需要填写 `myBreadthFirstSearch` 和 `myAStarSearch` 两个函数。函数的返回值为从初始状态到目标状态所经过的所有状态的列表。此处给出完整的 DFS 算法 `myDepthFirstSearch` 作为参考，请仔细阅读。**实现时请删去 `util.raiseNotDefined()`**。

```
def myBreadthFirstSearch(problem):
    # YOUR CODE HERE
    util.raiseNotDefined()
    return []

def myAStarSearch(problem, heuristic):
    # YOUR CODE HERE
    util.raiseNotDefined()
    return []
```

函数的参数 `problem` 可以调用3个函数：

- 函数 `getStartState` 可以获得该 `problem` 的初始状态:

```
start_state = problem.getStartState()
```

- 函数 `isGoalState` 可以判断当前状态 `state` 是否为目标状态:

```
problem.isGoalState(state) == True
```

- 函数 `getChildren` 可以获得 `state` 后可以到达的一系列状态。返回值是由二元组 `(next_state, step_cost)` 组成的列表。 `next_state` 是下一状态， `step_cost` 是从 `state` 到 `next_state` 需要的代价。

```
children = problem.getChildren(state)
```

参数 heuristic 本身就是一个函数，可以获得当前状态到目标状态的启发式估计值:

```
h_n = heuristic(state)
```

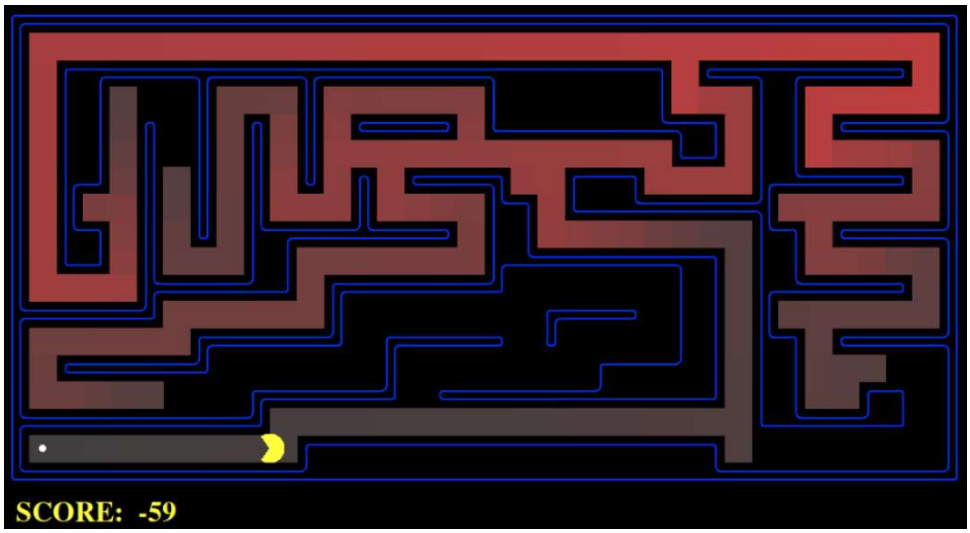
你可能还需要使用我们提供的**栈**、**队列**和**优先队列**这些数据结构。
大家在学习数据结构时可能都已经熟悉了栈和队列。它们的特点可以分别简单概括为先进后出和先进先出。

```
# stack  栈
stack = util.Stack()
stack.push('eat')
stack.push('study')
stack.push('sleep')
stack.pop() == 'sleep'

# queue 队列
queue = util.Queue()
queue.push('eat')
queue.push('study')
queue.push('sleep')
queue.pop() == 'eat'
```

优先队列的使用则需要赋予一个表示优先性的值，值越小就会越先出队。

```
pq = util.PriorityQueue()
pq.update('eat', 2)
pq.update('study', 1)
pq.update('sleep', 3)
pq.pop() == 'study'
```

3 个数据结构都有 `isEmpty()` 函数来判断数据结构内部是否有数据。
最终测试时，会动画显示 3 种搜索方法选择的路径以及搜索过的状态（红色表示），请比较一下三者的区别。

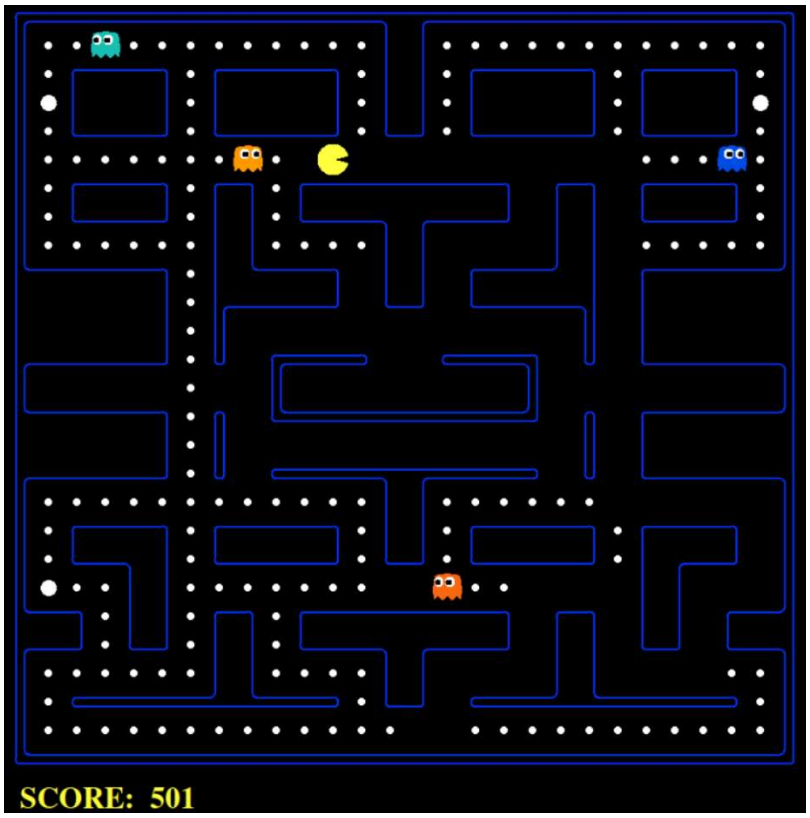


四. Multi-Agent

Multi-Agent 的问题是在有对手的情况下做出下一步决策使自己的利益最大化。游戏中自己的 agent 为吃豆人，对手 agent 为鬼。

我们已经实现了一个只基于当前状态做出反应的吃豆人，你可以输入以下命令查看它的表现。目录 multiagent/layouts 中有不同的游戏场景，你可以更改 `-l` 后的选项为对应文件名（不含 .py）进行测试。你可以更改 `-p` 后的选项为 `MinimaxAgent` 或 `AlphaBetaAgent` 来测试你实现的算法。

```
cd multiagent
python pacman.py -p ReflexAgent -l originalClassic --frameTime 0
```



你需要实现 minimax 算法和 alpha-beta 剪枝则会提前预估几步，在最坏的打算下最优化自己的效用。你**只需要填写** `MyMinimaxAgent` **和** `MyAlphaBetaAgent` **两个类**。其中函数getNextState 会被外部程序调用，获得当前状态下最优的下一个状态。你可能需要添加一些辅助函数来进行递归调用。
参数 `state` 可以调用 4 个函数：

- 函数 `isTerminate` 将返回当前状态 `state` 是否已经停止。停止状态意味着不会有下一个状态，游戏中指已经赢了或输了。

```
state.isTerminated() == True    # 取值只有 True 和 False
```

- 函数 `isMe` 将返回是否为己方 `agent` 在进行操作。在游戏中， `True` 表示轮到吃豆人采取移动操作， `False` 表示轮到某个鬼在采取移动操作。你可以用来判断当前应该最大化还是最小化效用。

```
state.isMe() == True    # 取值只有 True 和 False
```

- 函数 `getChildren` 将返回当前状态 `state` 接下来所有可能的状态。请使用 `for` 来遍历。注意：alpha-beta 剪枝的目的是缩小搜索空间，如果在遍历 `getChildren()` 中，`MyAlphaBetaAgent` 发现可以剪枝，请停止遍历。

```
for child in state.getChildren():
    ...
```

- 函数 evaluateScore 将返回当前状态对于目标 agent 的效用。

```
score = state.evaluateScore()
```

值得注意的是算法搜索的深度 depth，它指的是每个 agent 所走的步数。例如 depth=2，有 1 个 pacman 和 2 个 ghost，则从搜索树的最顶层到最底层应该经过 pacman->ghost1->ghost2->pacman->ghost1->ghost2，操作应该为 max->min->min->max->min->min。

实验提交

截止时间：2022年7月3日23:59。
提交方式：BB 系统作业区。
提交格式：按下目录组织文件夹，并打包为同名（.zip 等）压缩文件

学号+姓名+Lab2
|- report.pdf
|- myImpl.py

注意事项：代码请提交且只能提交 myImpl.py，不得提交其余任何代码文件（包括但不限于 sh、py 等）。【所以你只能对 myImpl.py 进行修改，不得修改其他文件】

实验评分

百分制，每个算法各占 20 分，实验报告 20 分。
要求：

1. 各个算法在语义上基本正确，如 BFS 算法不能写成 DFS 或代价一致搜索等；
2. 通过所有测试；
3. 实验报告包含
    1. 简要说明每个算法的实现思路（每个算法几十字~200字即可）；

2. 附上每个算法的测试截图，分为命令行输出结果和运行画面截图，两种都要放在报告里。