同濟大學
TONGJI UNIVERSITY

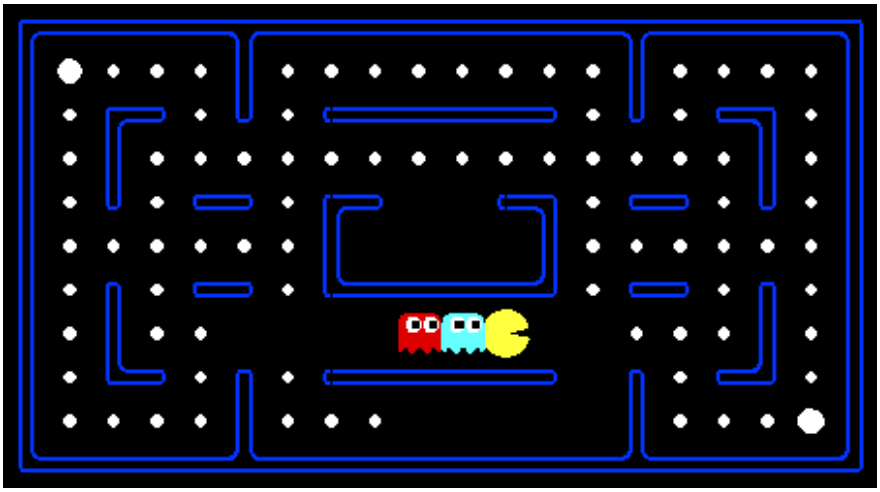
人工智能原理与技术
Project2实验报告

学　　号：2154312
姓　　名：郑博远
教　　师：王俊丽
完成日期：2023 年 4 月 13 日

1. 问题概述

1.1. 问题的直观描述

在如下图所示的迷宫世界中，白色圆点代表着食物（大圆点表示胶囊，在吃掉胶囊后Pacman 吃掉幽灵可以得到额外的加分），蓝色部分代表着墙壁，黄色的吃豆人智能体Pacman 将会在躲避幽灵的情况下，尽可能高效地收集食物。Project2 将实现 Minimax 和Expectimax 搜索，并进行评估函数的设计。本项目共分为 5 个子问题，具体描述如下：



问题 1 中的 Pacman 是反射型智能体（Reflex Agent），它会根据当前感知到的环境状态来选择最佳的动作。在本问题中，需要设计合适的评估函数，考虑到食物距离、幽灵位置、得分情况等以帮助 Pacman 决策出最佳的下一步动作，从而实现食物的高效收集。

问题 2、问题 3 中，我们将实现 Pacman 的对抗搜索，即通过 Minimax 搜索树来指导智能体 Pacman 在尽可能避开幽灵的前提下作出收集食物的决策。

- 问题 2 中，我们要实现 Minimax 搜索树的建立。特别地，Minimax 树将会有多个 min 层（对应每个幽灵）和一个 max 层。需要注意的是，实现的代码要能够将搜索树扩展到任意深度，这可以通过提供的 self.depth 来实现；此外，还将使用提供的 self.evaluationFunction 来作为评估函数；
- 问题 3 中，需要在问题 2 构建的搜索树基础上创建新的 Agent 进行 $\alpha-\beta$剪枝，以实现更高效的 Minimax 搜索。具体来说，需要实现一个 AlphaBetaAgent 对 Minimax 搜索树进行$\alpha-\beta$剪枝，在搜索树中更加高效地寻找最优策略。

问题 4 和问题 5 中，我们将探究并实现 ExpectimaxAgent。与问题 2 和问题 3 中的Minimax 算法不同，ExpectimaxAgent 不再假定幽灵具有最优决策，而是能够针对作出次优

决策的 Agent 做出决策。在 Pacman 游戏中，幽灵并非总是能做出最优决策，因此使用 ExpectimaxAgent 算法进行决策更为合适。在此算法中，我们将幽灵的决策简化为在所有合法的动作中进行随机选择。

- 问题 4 中，我们将进行与问题 2 中类似的搜索树建立。不同的是，在 Pacman 的决策层仍然采用取 max 的策略，而在每个幽灵的决策层（对应问题 2 中的 min），则返回各个不同动作对应估值的期望（即算术平均值）；
- 问题 5 中，我们需要设计对状态的评估函数，从而使 Pacman 在躲避幽灵的前提下尽可能高效地收集食物。为了使决策更佳，评估函数需要考虑到得分情况、食物位置和幽灵距离等信息以进行估值。需要注意的是，此问题的评估函数不同于问题 1 中的反射型智能体，后者对某个状态下的动作转移进行评估，而前者则是对 Pacman 对应的具体游戏状态进行评估。

1.2. 项目已有代码的阅读与理解

1.2.1. `util.py`

与 Project1 中类似，该部分提供了搜索算法的有用数据结构，包括栈（LIFO，后进先出）、队列（FIFO，先进先出）与优先队列（保持队首元素最小）。其中 PriorityQueue 在向队列加入元素时额外传入一个值作为排序的依据；此外，还提供了继承 PriorityQueue 的 PriorityQueueWithFunction 类，可以通过通过传入 priority function 作为依据进行排序。这些数据结构主要用于其他项目中，在 Project2 中并未涉及。

此外，utill.py 中还提供了函数 manhattanDistance，能够返回 Pacman 迷宫中两点的曼哈顿距离。该函数返回的曼哈顿距离可以作为估值函数中的重要评估依据。

1.2.2. `pacman.py`

该文件运行 Pacman 游戏的主要文件。其中主要与外部交互的是 pacman.py 中定义的 GameState 类，其涉及到与智能体 Pacman 相关的信息，具体包含食物、智能体配置、分数变化等信息。在 searchAgent.py 文件中对具体问题 searchProblem 类的定义与初始化中，就包含了 GameState 类的信息。

在本次的 Project2 中，GameState 类中常用的方法有：getPacmanPosition，能够获取到当下吃豆人智能体所在的位置；getGhostPositions，能够获取到各个幽灵的当前位置；getCapsules，能够获取到胶囊的位置；getFood 与 getNumFood，能够返回食物所在的位置与食物的数量；generateSuccessor（或 generatePacmanSuccessor），可以生成某个 action 后的新游戏状态，在构建搜索树的过程中十分重要。以下方法没有在代码编写中直接调用，

但在其他文件或其他项目中较为常用：hasFood 与 hasWall，能够判断指定位置是否有食物或墙壁；isWin 和 isLose 能判断游戏进行情况。

```
class GameState:
    """
    A GameState specifies the full game state, including the food, capsules,
    agent configurations and score changes.

    Note that in classic Pacman, Pacman is always agent 0.
    """
```

1.2.3. game.py

该文件描述了 Pacman 游戏世界运作的逻辑。它包含了几种类的定义，包括 AgentState、Agent、Direction 和 Grid。

```
class AgentState:
    """
    AgentStates hold the state of an agent (configuration, speed, scared,
etc).
    """
```

game.py 中给出了 Agent 类的抽象类定义，规定了 Agent 类必须要具备 getAction 方法；具体的各类型 Agent 继承类在 searchAgent.py 中定义。AgentState 类用于存储智能体的当前状态。在本次 Project 中，AgentState 信息包括其配置、速度等。

```
class Configuration:
    """
    A Configuration holds the (x,y) coordinate of a character, along with its
    traveling direction.
    """
```

Directions 类主要用于东、西、南、北、左、右等方向常量的定义，使代码具有更高的可读性。Configuration 类用于存储角色的位置与运动方向。位置坐标系和常见的坐标系较不同，它以游戏窗口的左下角为原点，x 轴向右，y 轴向上，因此向上运动是沿着 y 轴的正方向，即(0,1)方向。其中的 generateSuccessors 方法在 Project1 中较常使用，用于获得当前位置 Agent 可行的后继位置。

Grid 类是封装好的二维列表。与 Configuration 类中规定相同，grid[x][y]存储的某位置

信息以左下角为 (0,0) 坐标，向右上角递增。Grid 类可以用于以二维数组的形式传递墙壁位置信息、食物位置信息等。其中的__str__方法可以以字符串形式输出 Pacman 地图信息，asList 方法可以输出所有值为真的位置坐标列表。

```
class Grid:
    """
    A 2-dimensional array of objects backed by a list of lists.  Data is
accessed via grid[x][y] where (x,y) are positions on a Pacman map with x
horizontal,
    """
```

1.2.4. multiAgents.py

该文件对问题 1 到 5 中所需要的智能体 Agent 进行了定义。主要分为问题 1 中的反射型 ReflexAgent 和问题 2 到问题 5 中的多智能体搜索型 MultiAgentSearchAgent。这两个类都是对 game.py 中 Agent 类的继承。更进一步地，问题 2 中的未剪枝 Minimax 搜索对应的 MinimaxAgent、问题 3 中剪枝后的 AlphaBetaAgent、问题 4 中的 ExpectimaxAgent 均是 MultiAgentSearchAgent 的继承。

Agent 类中最必不可少的方法是初始化的__init__方法以及获取下一步动作的 getAction 方法。在 MultiAgentSearchAgent 中，__init__函数初始化了搜索最大深度的 self.depth 以及用于评估的 self.evaluationFunction 这两个重要变量。在该文件中，主要需要为定义不同的 Agent 添加 getAction 方法；此外，还需要设计估值函数用以搜索树的评估。

```
class ReflexAgent(Agent):
    """
    A reflex agent chooses an action at each choice point by examining
    its alternatives via a state evaluation function.
    """
```

```
class MultiAgentSearchAgent(Agent):
    """
    This class provides some common elements to all of your
    multi-agent searchers.  Any methods defined here will be available
    to the MinimaxPacmanAgent, AlphaBetaPacmanAgent & ExpectimaxPacmanAgent.
    """
```

1.3. 解决问题的思路和想法

1.3.1. 问题 1 的解决思路

问题 1 中，我们需要设计一个对于所有合法动作的估值函数，从而指引反射性智能体在躲避幽灵的前提下尽可能高效地收集食物。容易注意到，食物的距离、幽灵的距离都是设计估值函数的有效信息。转移后最近食物的距离可以很好地启发 Pacman 收集食物，因此作为考量依据之一。幽灵在较远处的移动不应对 Pacman 产生影响使其畏手畏脚；因此，若幽灵太远，则 Pacman 应该自由活动。考虑到让各个变量尽可能的归一化以方便调整各个部分的权值，都采用倒数的形式；即前者使用最近食物的倒数，后者累加某个距离范围内的各个幽灵的距离倒数和（由于负相关要添加负号）。此外，某一步动作若正好吃掉食物可能反而造成了最近食物更远，因此可以额外添加一项前后食物数量差来引导 Pacman 收集食物（若吃掉食物该值为 1，产生的影响大于最近食物距离带来的影响）。

1.3.2. 问题 2 的解决思路

问题 2 中，我们需要实现一个最小化最大值（Minimax）搜索算法，使得 Pacman 智能体能够在多个幽灵的干扰下找到最优的动作方案。我们需要使用提供的 MinimaxAgent 类，实现一个递归的算法。具体实现时，可以通过自行设计 getMin 和 getMax 两个函数在递归的互相调用中实现 Minimax 搜索。该算法需要在任意数量的鬼魂下运行，并且需要扩展游戏树以任意深度。算法需要引用 self.depth 和 self.evaluationFunction 这两个变量来实现深度的要求和状态的评估；对于每个叶子节点的评分（到达深度或者游戏结束），可以使用提供的评估函数 self.evaluationFunction。问题的重点是确保在游戏中只使用必要的次数，避免过多或过少的调用 GameState.generateSuccessor 方法。

1.3.3. 问题 3 的解决思路

问题 3 中，我们需要使用 $\alpha-\beta$ 剪枝算法更高效地搜索 Minimax 树。$\alpha-\beta$ 剪枝通过排除不必要的搜索来达到加速的效果，即当发现某个节点的值已经不会影响当前搜索的结果时，就不再搜索这个节点的子节点。

在代码实现上，$\alpha-\beta$ 剪枝是通过设置 α 和 β 值来实现的。α 表示当前搜索中的最大值，β 表示当前搜索中的最小值。在 max 层中，每个子节点的值都要大于 α，否则就可以剪枝；在 min 层中，每个子节点的值都要小于 β，否则也可以剪枝。通过 α 和 β 的不断更新，可以逐渐缩小搜索范围，从而减少搜索节点的数量。

对于本问题，我们可以先考虑使用递归的方式实现 $\alpha-\beta$ 剪枝算法。具体来说，我们需要实现两个递归函数 getMax 和 getMin，分别代表 max 层和 min 层。在 getMax 函数中，我们需要找到当前节点的最大值，并更新 α 的值。在 getMin 函数中，我们需要找到当前节

点的最小值，并更新 β 的值。通过递归地调用这两个函数，我们可以逐步遍历整个 Minimax 树进行剪枝，从而达到加速的效果。

1.3.4. 问题 4 的解决思路

问题 4 中，我们需要实现一个 Expectimax 智能体，用于应对可能会做出次优选择的智能体的行为。与 Minimax 搜索不同的是，ExpectimaxAgent 不再假设正在与做出最佳决策的对手进行对战。在本问题中，对于 Pacman 世界中的幽灵，我们将其选择动作的概率分布视为均匀分布。与问题 2、问题 3 中类似，我们需要使用提供的 ExpectimaxAgent 类，实现一个递归的算法；算法需要引用 self.depth 和 self.evaluationFunction 这两个变量来实现深度的要求和状态的评估；对于每个叶子节点的评分（到达深度或者游戏结束），可以使用提供的评估函数 self.evaluationFunction。具体实现时，当 Pacman 进行选择时仍然要设计类似的 getMax 函数，但幽灵的决策层则给出各个动作的平均期望值即可（因为假设其在合法动作中进行随机选择），通过两个函数的互相调用便可实现 Expectimax 搜索。

1.3.5. 问题 5 的解决思路

问题 5 中，我们需要设计一个更好的估值函数来指导 Pacman 智能体在多个幽灵的干扰下收集食物并尽可能地获得高分；不同于问题 1 中的是，评估函数的对象是 Pacman 的状态而非 Pacman 动作。在游戏分值的基础上，考虑如下因素作为估值函数的一部分：首先考虑到的是食物的距离，与问题 1 中的思路类似，我们可以使用最近食物的倒数来表达距离的影响；此外，Pacman 到食物的平均距离也能够引导其靠近大片食物连接的区域从而高效地收集食物，因此也可以纳入估值函数的累计中；其次，也要考虑幽灵的距离，这一点的设计与问题 1 中基本相同，即当幽灵过远时可以不纳入考虑以避免 Pacman 畏手畏脚。最后，我们可以根据实际情况调整各个因素的权重，使得估值函数更加准确。

2. 算法设计

2.1. 问题 1 反射型智能体的估值函数

算法功能：

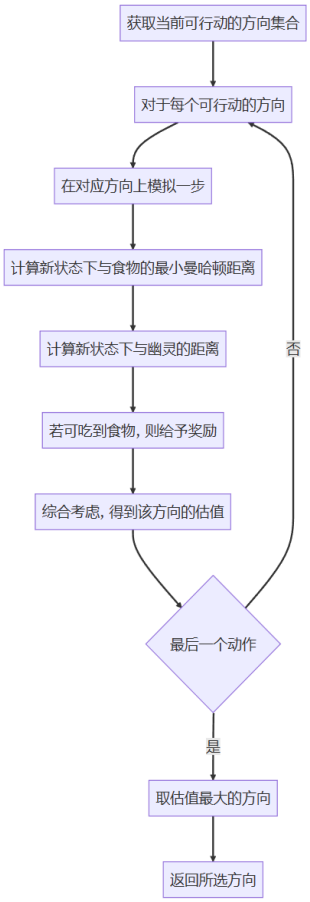
本算法实现了一个反射型智能体，即根据当前状态下各项指标的值，通过对各项指标权重的赋值，对可选的动作进行评估并选择最优动作。通过对各个合法动作的估值函数的设计（考虑食物数量、最近的食物距离、一定范围内的幽灵距离等因素），智能体 Pacman 能够在高效收集食物的同时，尽可能地避开幽灵。

设计思路：

1. 选取食物中的最小曼哈顿距离作为估值之一，即 Pacman 希望离最近的食物越近越好，从而引导其收集食物；
2. 考虑幽灵的影响，通过计算 Pacman 与每个幽灵的曼哈顿距离，判断 Pacman 是否处于危险中，以此来决定幽灵对于估值的贡献。若幽灵距离过大，则此项不计入估值函数考量中（避免 Pacman 畏手畏脚）；
3. 若 Pacman 下一步能吃食物，则给予一定激励。

具体而言，在第 1 步中，算法遍历当前状态下所有食物的位置，并计算 Pacman 到该食物的曼哈顿距离；选取曼哈顿距离最小值的倒数作为估值函数的一部分。在第 2 步中，算法遍历了当前状态下所有幽灵的位置，若 Pacman 与某个幽灵的曼哈顿距离小于 3，则认为 Pacman 处于危险中，并将该幽灵对估值的贡献计入总值中，即取其负倒数乘以该部分权重进行累加。在第 3 步中，算法考虑了 Pacman 该步吃掉食物对于估值的影响，若下一步能够吃掉食物，则估值加上一个激励。

算法流程图：如上下图所示。

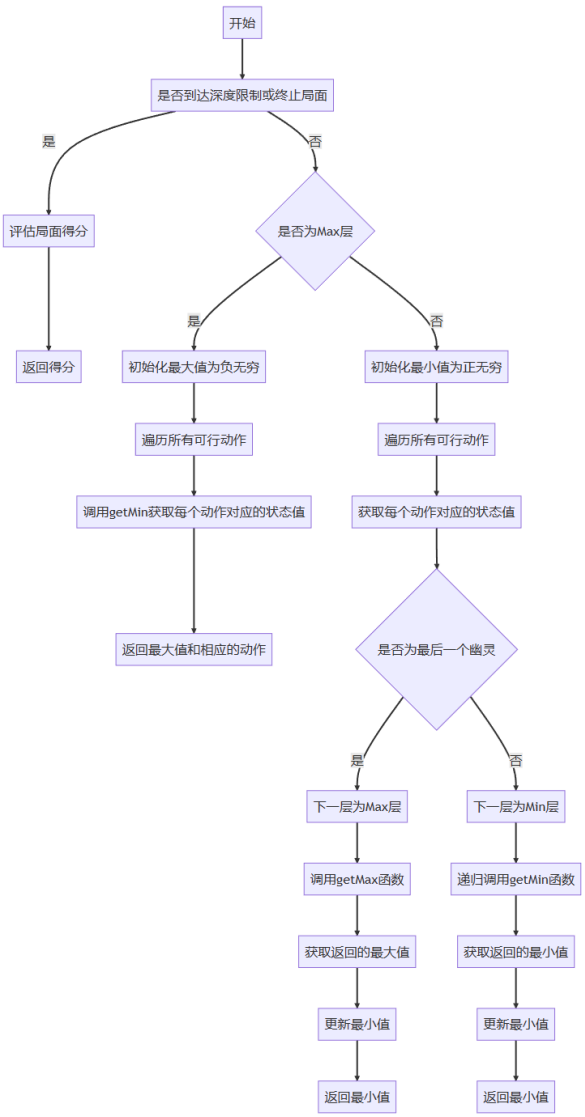


反射型智能体的算法流程图

2.2. 问题 2 Minimax 搜索树

算法功能：

本次实验的目的是实现 Minimax 算法，用于计算游戏状态下 Pacman 与幽灵之间的最佳决策。Minimax 算法是一种博弈树搜索算法，用于求解零和博弈的最优策略。其功能是在博弈树上搜索所有可能的游戏状态，并为每个游戏状态计算出一定深度内的最优解（即能够获得最大收益的策略），从而找到最优的决策。在游戏状态中，Pacman 和幽灵之间交替进行动作，因此 Minimax 算法将游戏状态看作是一个多层的树形结构，其中每一层代表一个 Agent 的动作轮次。在每个 Max 层，Pacman 会选择可以获得最大收益的动作，而在每个 Min 层，幽灵则会选择可以获得最小收益的动作。通过这种方式，算法能够预测出对手的最优策略，并尽可能地避免对手的优势。



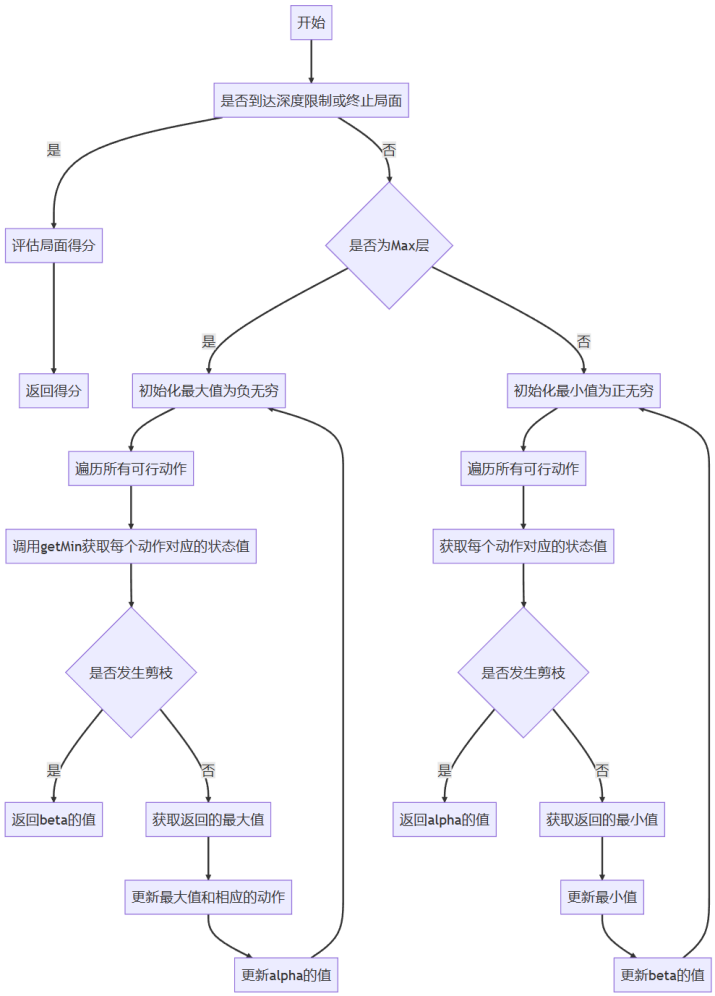
Minimax 搜索的算法流程

设计思路：

1. 从当前游戏状态开始，通过递归搜索博弈树；
2. 在搜索过程中，交替进行 Max（对应 Pacman 决策）和 Min（对应幽灵决策）操作，找到最优解并返回该节点的估价值以及对应的动作；
3. 在 Max 层中，对于每个合法的动作，通过递归调用 Min 层，找到所有子节点中的最大值和对应的动作；
4. 在 Min 层中，对于每个合法的动作，若当前 Agent 不是最后一个 Agent，则递归调用下一个 Agent 的 Min 层；否则下一层 Pacman 决策，递归调用 Max 层；
5. 如果当前的搜索树达到了设定的搜索深度，或者搜索到了终止状态，则通过调用 self.evaluationFunction 返回当前状态的估价值。

算法流程图：如上页图所示。

2.3. 问题 3 $\alpha-\beta$ 剪枝



$\alpha-\beta$ 剪枝的算法流程图

算法功能：本算法是一个与问题 2 中的 Minimax 算法相同，是寻找最优决策的基于博弈树的搜索算法。在问题 2 中 Minimax 算法的基础上，$\alpha-\beta$ 剪枝算法也是在博弈树上搜索最优解。该算法通过减少搜索的分支，从而减少搜索所需的时间。其核心思想是在搜索过程中对已经搜索的结点设定一 α 和 β 的边界值，只搜索那些没有超出这个范围的结点，从而剪去大量的冗余搜索，很大程度上提升了搜索的效率。

设计思路：

$\alpha-\beta$ 剪枝算法是 Minimax 算法的一个优化版本。其核心思想是对于某个结点，如果已经找到了某个最优解，那么在搜索到后续结点的过程中，如果发现后续结点并不能提供更优的解，那么就可以直接剪掉这个后续结点的搜索。因此，该算法可以减少搜索的结点数量，从而提高搜索效率。

在 $\alpha-\beta$ 剪枝算法中，维护两个值 α 和 β，分别表示当前结点的最优解的下界和上界。在搜索过程中，当一个 Max 层节点搜索到一个比当前 β 值大的 Min 层节点时，就可以直接剪枝，因为 Min 层节点的选择肯定不会优于当前的 β 值，因此这个 Min 层节点的选择不会被选择；反之，当一个 Min 层节点搜索到一个比当前 α 值小的 Max 层节点时，也可以直接剪枝，因为 Max 层节点的选择肯定不会优于当前的 α 值，因此这个 Max 层节点的选择不会被选择。

在实现上，我们可以使用递归函数（即 getMin 与 getMax 两个层函数的互相调用）实现 $\alpha-\beta$ 剪枝算法，该函数可以接收当前的游戏状态、搜索深度、当前是哪个幽灵在进行搜索、当前的 α 和 β 值。在每次递归调用时，函数会根据当前搜索的是 Max 层还是 Min 层来更新 α 或 β 的值，并根据当前的 α 和 β 值来判断是否需要剪枝。同时，递归调用下一层的函数来进行搜索，并根据搜索到的值来更新当前结点的值。

算法流程图：如上页图所示。

2.4. 问题 4 对手随机决策的 Expectimax 算法

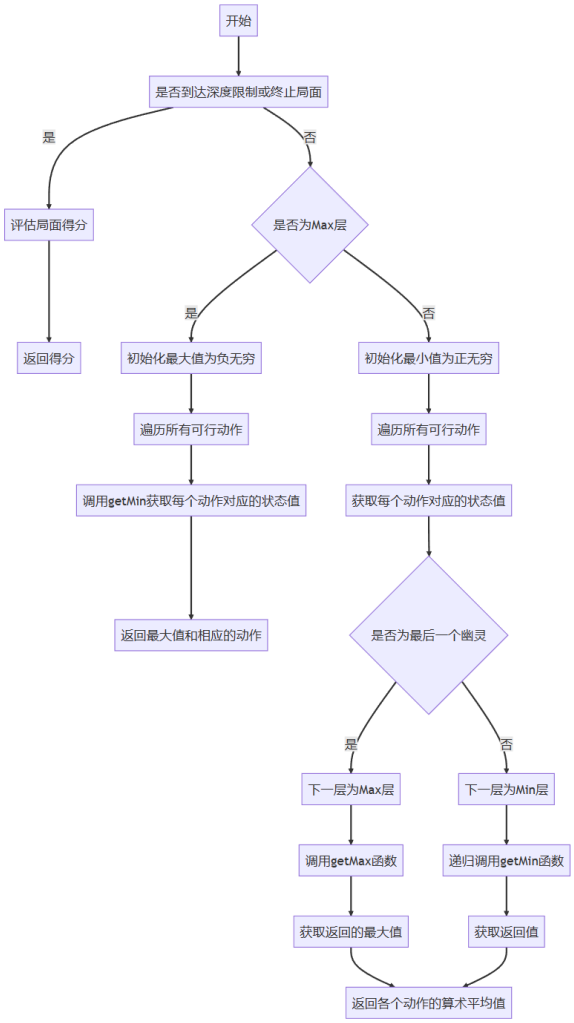
算法功能：

本算法是基于 Expectimax 的多智能体搜索算法，是在 MiniMax 算法的基础上，针对多智能体环境的搜索算法。该算法可以用于求解博弈中的最优策略，也可用于其他多智能体环境中的最优动作。与 MiniMax 算法不同的是，Expectimax 算法会对随机因素进行期望处理，而不是像问题 2 或问题 3 中 MiniMax 算法的最大或最小值。Expectimax 算法将随机因素纳入考虑范围内，即假设对手随机选择动作。因此，Expectimax 算法将在每个 Min 节点上将所有可行的动作加权平均，而不是将它们视为最坏情况。这种随机因素的引入，使得 Expectimax 算法能够在不完全信息游戏中更好地应对复杂情况。

设计思路：

1. 从当前游戏状态开始，通过递归搜索博弈树；
2. 在搜索过程中，交替进行 Max（对应 Pacman 决策）和 Random（对应幽灵随机决策）操作，找到最优解并返回该节点的估价值以及对应的动作；
3. 在 Max 层中，对于每个合法的动作，通过递归调用 Random 层，找到所有子节点中的最大值和对应的动作；
4. 在 Random 层中，对于每个合法的动作，若当前 Agent 不是最后一个 Agent，则递归调用下一个 Agent 的 Random 层；否则下一层 Pacman 决策，递归调用 Max 层。特别地，为了考虑幽灵的不确定性，需要在 Random 层中对所有子节点的估价值取算术平均值而不是最小值；
5. 如果当前的搜索树达到了设定的搜索深度，或者搜索到了终止状态，则通过调用 self.evaluationFunction 返回当前状态的估价值；

算法流程图：如下图所示。



Exceptimax 的算法流程图

2.5. 问题 5 状态估值函数设计

算法功能：

本小题中定义了函数 betterEvaluationFunction。该函数是一个评估函数，用于评估当前游戏状态的好坏程度，可以用于智能体搜索树中到达搜索深度或陷入死状态时的局面评估。在该函数中，使用了以下策略来计算估价值：1. 选取食物中的最小曼哈顿距离作为估值之一；2. 如果靠近胶囊，则考虑吃胶囊的收益；3. 考虑幽灵的影响，对于惊吓状态的幽灵可以主动追逐，如果幽灵距离太远则不应该干扰 Pacman，而极力避免被幽灵吃。

设计思路：

1. 首先获取当前游戏状态的所有信息，包括食物、胶囊、幽灵和 Pacman 的位置；
2. 计算 Pacman 到食物的最小曼哈顿距离，将其倒数作为估值的一部分。同时，计算所有食物的平均曼哈顿距离，将其倒数作为另一个估值的一部分；
3. 如果胶囊距离 Pacman 较近，则将其距离作为估值的一部分。胶囊的权重更大，因为吃掉胶囊可以让幽灵变成惊恐模式，Pacman 可以主动追逐幽灵；
4. 考虑考虑幽灵的影响：对于每个幽灵，计算它和 Pacman 之间的曼哈顿距离。如果距离大于 0，则考虑幽灵的状态，进行以下操作：对于惊吓状态的幽灵，权重更大地加入到估值中，因为此时幽灵应该会逃离 Pacman，可以主动追逐幽灵；对于非惊吓状态的幽灵，如果距离太远，则不应该影响 Pacman 的决策，因此将其权重降低；如果距离较近，则应该避免被幽灵吃掉，因此将其权重提高。如果幽灵距离 Pacman 的曼哈顿距离为 0，则 Pacman 会被幽灵吃掉，此时将估值设置为负无穷；
5. 返回估值，估值包括当前游戏状态的得分以及以上 4 个因素的加权和。

3. 算法实现

3.1. 问题 1 反射型智能体的估值函数

该问题中，我实现了反射型智能体 ReflexAgent。其中 getAction 方法已经给出，是整个智能体算法的核心。它首先获取当前状态下所有合法的动作，然后使用我自行设计的估值函数 evaluationFunction 对这些动作进行评估并选择最优的动作。若多个最优动作有相同估值，则在其中进行随机选择。

evaluationFunction 函数的作用是评估当前状态到下一个可能的状态的转移动作，返回一个值表示这个 Action 的好坏。具体来说，它考虑了下一个状态下 Pacman 的位置、食物的位置、幽灵的位置等因素，并计算出一个估值。这个估值可以作为 getAction 方法的参考，帮助它选择最优的动作。

在我的代码中，我选择了以下三个因素作为动作估值的依据：Pacman 在新位置距离最近的食物的曼哈顿距离、幽灵对 Pacman 的影响以及该动作执行后的下一个状态是否可以吃到食物。对于第一个因素，我计算了所有食物的曼哈顿距离，并选取其中最小的距离的倒数作为当前状态的估值之一，以促进 Pacman 对事物的收集。对于第二个因素，我考虑了幽灵对 Pacman 的影响。如果幽灵距离 Pacman 太近，那么 Pacman 可能会因为被幽灵追赶而失败。因此，我为幽灵距离 Pacman 的倒数作为 Pacman 的负面估值，这样 Pacman 就会尽可能地避开幽灵。但若幽灵距离太远，则不应将幽灵的影响纳入考量。对于第三个因素，如果下一个状态可以吃到食物，则让估值加一。这可以避免吃了食物之后最近的食物曼哈顿距离反而更远的问题，让 Pacman 更积极主动地收集食物。具体代码如下：

```
def evaluationFunction(self, currentGameState: GameState, action):
    """
    Design a better evaluation function here.
    """
    # Useful information you can extract from a GameState (pacman.py)
    successorGameState = currentGameState.generatePacmanSuccessor(action)
    curFood = currentGameState.getFood()
    newPos = successorGameState.getPacmanPosition()
    newFood = successorGameState.getFood()
    newGhostPos = successorGameState.getGhostPositions()

    "*** YOUR CODE HERE ***"

    # 选取食物中的最小曼哈顿距离作为估值之一
    minFoodDis = float("inf")
    ghostValue = 0
    for foodPos in newFood.asList():
        minFoodDis = min(minFoodDis, manhattanDistance(newPos, foodPos))

    # 考虑幽灵的影响
    for GhostPos in newGhostPos:
        dis = manhattanDistance(newPos, GhostPos)
        # 如果幽灵距离太远 则不应该干扰 Pacman
        ghostValue += 5 / (dis + 1) if dis < 3 else 0
```

3.2. 问题 2 Minimax 搜索树

本问题中，我主要实现了一个 Minimax 搜索算法，用于寻找当前游戏状态下 Pacman 的最佳动作（即最大化 pacman 的得分）。该算法通过递归地在 max 和 min 层之间交替运

行，计算 GameState 中每个 Agent 可以采取的动作，并选择最优动作。

在 getMax 函数中，它首先获取当前 GameState 中所有合法的动作。然后，它检查是否达到了搜索深度或者游戏结束。如果达到了深度上限或游戏结束，那么它会直接返回当前 GameState 的评估值。否则，它遍历所有合法的动作，针对每个动作，它将生成一个新的 GameState，并将其作为参数传递给 getMin 函数，以计算当前 Agent 的值。最终，它将返回最大的值以及相应的动作（因为最顶层要返回 Pacman 的动作）。

在 getMin 函数中，它首先获取当前 GameState 中所有合法的动作。然后，如果没有合法的动作，那么它会直接返回当前 GameState 的评估值。否则，它遍历所有合法的动作，对于每个动作，如果当前 Agent 是最后一个 Agent，那么它将生成一个新的 GameState，并将其作为参数传递给 getMax 函数，以计算下一层的值。否则，它将递归调用 getMin 函数，以计算下一个 Agent 的值。最终，它将返回最小的值。

整个算法的核心是递归遍历所有的 Agent 和动作，寻找最大值和最小值。在搜索树的底部，算法通过评估函数来估计每个 GameState 的得分，然后逐级向上递归，直到回到根节点。这个算法的时间复杂度是指数级的，因为它要遍历整个搜索树。为了加速搜索，可以使用 $\alpha-\beta$ 剪枝等技术，这将在第三小题中做进一步讨论。

```
def getAction(self, gameState: GameState):
        """
        Returns the minimax action from the current gameState using
self.depth and self.evaluationFunction.
        """
        "*** YOUR CODE HERE ***"
        return self.getMax(gameState, 0, 0)[1]

    # Max 层 返回最大值和相应动作的元组
    def getMax(self, gameState: GameState, depth, agentIndex = 0):
        actions = gameState.getLegalActions(agentIndex)
        # 深度到达上限或死局 终止
        if depth == self.depth or len(actions) == 0:
            return self.evaluationFunction(gameState), None
        maxValue = float("-inf")
        nextAction = None
        for action in actions:
            # 从第一个幽灵开始
            value = self.getMin(gameState.generateSuccessor(agentIndex,
action), depth, 1)
            if value > maxValue:
```

```
                maxValue = value
                nextAction = action
        return maxValue, nextAction

    # Min 层，返回最小值
    def getMin(self, gameState: GameState, depth, agentIndex = 1):
        actions = gameState.getLegalActions(agentIndex)
        # 陷入死局 终止
        if len(actions) == 0:
            return self.evaluationFunction(gameState)
        minValue = float("inf")
        for action in actions:
            # 若当前是最后一个幽灵，则下一层 Max
            if agentIndex == gameState.getNumAgents() - 1:
                value = self.getMax(gameState.generateSuccessor(agentIndex,
action), depth + 1, 0)[0]
            # 逐个 Agent 取 min 值
            else:
                value = self.getMin(gameState.generateSuccessor(agentIndex,
action), depth, agentIndex + 1)
            minValue = min(minValue, value)
        return minValue
```

3.3. 问题 3 $\alpha-\beta$ 剪枝

$\alpha-\beta$ 剪枝是基于 MinMax 算法的一种优化，它通过维护两个值 α 和 β 来剪枝，其中 α 表示 Max 层当前已知的最大值，而 β 表示 Min 层当前已知的最小值。当某个节点的值不再对答案产生影响时，就可以进行剪枝。

具体实现上，在 getMax 和 getMin 函数调用时均要传递与维护 α 与 β 两值。以 getMax 为例，若当前调用 getMin 得到的值大于 β，则 getMax 取得的值一定对结果不产生影响，应当剪枝；此外，若当前 getMin 得到的值大于 α，则应该更新维护 α 的值。同理可得，在 getMin 层中，若当前调用 getMax 得到的值小于 α，则 getMin 取得的值一定对结果不产生影响，应当剪枝；此外，若当前 getMax 得到的值小于 β，则应该更新维护 β 的值。具体解释参见代码中的注释：

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

```
    """
    "*** YOUR CODE HERE ***"
    return self.getMax(gameState, 0, 0)[1]

# Max 层 返回最大值和相应动作的元组
def getMax(self, gameState: GameState, depth, agentIndex = 0, alpha =
float("-inf"), beta = float("inf")):
    actions = gameState.getLegalActions(agentIndex)
    # 深度到达上限或死局 终止
    if depth == self.depth or len(actions) == 0:
        return self.evaluationFunction(gameState), None
    maxValue = float("-inf")
    nextAction = None
    for action in actions:
        # 从第一个幽灵开始
        value = self.getMin(gameState.generateSuccessor(agentIndex, action),
depth, 1, alpha, beta)
        if value > beta:
            return value, action
        if value > maxValue:
            maxValue = value
            nextAction = action
        # alpha 的值也要更新 保持 alpha 是当前 max 层能获得的最大值
        alpha = max(alpha, value)
    return maxValue, nextAction

# Min 层，返回最小值
def getMin(self, gameState: GameState, depth, agentIndex = 1, alpha =
float("-inf"), beta = float("inf")):
    actions = gameState.getLegalActions(agentIndex)
    # 陷入死局 终止
    if len(actions) == 0:
        return self.evaluationFunction(gameState)
    minValue = float("inf")
    for action in actions:
        # 若当前是最后一个幽灵，则下一层 Max
        if agentIndex == gameState.getNumAgents() - 1:
            value = self.getMax(gameState.generateSuccessor(agentIndex,
action), depth + 1, 0, alpha, beta)[0]
        # 逐个 Agent 取 min 值
        else:
```

```
            value = self.getMin(gameState.generateSuccessor(agentIndex,
action), depth, agentIndex + 1, alpha, beta)
        if value < alpha:
            return value
        minValue = min(minValue, value)
        # beta 的值也要更新 保持 beta 是当前 min 层能获得的最小值
        beta = min(beta, minValue)
    return minValue
```

3.4. 问题 4 对手随机决策的 Expectimax 算法

本问题实现的 Expectimax 智能体与问题 2 中极为类似，也是通过 getMax 与 getRandom 层的相互递归调用实现搜索树的构建。不同点在于，由于假定幽灵在所有动作中进行随机选择，getRandom 函数应该返回所有合法状态值的算术平均值。具体实现上，需要累计所有合法值进行求和，并除以合法动作总数进行算术平均计算。

```
def getAction(self, gameState: GameState):
    """
    Returns the minimax action using self.depth and self.evaluationFunction
    """
    "*** YOUR CODE HERE ***"
    return self.getMax(gameState, 0, 0)[1]

# Max 层 返回最大值和相应动作的元组
def getMax(self, gameState: GameState, depth, agentIndex = 0):
    actions = gameState.getLegalActions(agentIndex)
    # 深度到达上限或死局 终止
    if depth == self.depth or len(actions) == 0:
        return self.evaluationFunction(gameState), None
    maxValue = float("-inf")
    nextAction = None
    for action in actions:
        # 从第一个幽灵开始
        value = self.getRandom(gameState.generateSuccessor(agentIndex,
action), depth, 1)
        if value > maxValue:
            maxValue = value
            nextAction = action
    return maxValue, nextAction
```

```
# Agent 随机选择
def getRandom(self, gameState: GameState, depth, agentIndex = 1):
    actions = gameState.getLegalActions(agentIndex)
    # 陷入死局 终止
    if len(actions) == 0:
        return self.evaluationFunction(gameState)
    valueSum = 0
    for action in actions:
        if agentIndex == gameState.getNumAgents() - 1:
            value = self.getMax(gameState.generateSuccessor(agentIndex,
action), depth + 1, 0)[0]
        else:
            value = self.getRandom(gameState.generateSuccessor(agentIndex,
action), depth, agentIndex + 1)
        # 对所有的 value 求和
        valueSum += value

    # 因为是完全随机求和 所以应该对各种情况的值求算术平均
    return valueSum / len(actions)
```

3.5. 问题 5 状态估值函数设计

本问题中需要自定义的评估函数，不同于问题 1 中对于动作的评估函数，该函数用于衡量当前游戏状态的好坏程度，返回一个实数作为评估值。

该函数的实现思路如下：首先，计算当前位置到所有食物的最小曼哈顿距离，选取其中最小值作为估值之一。同时，将所有食物的平均距离作为另一个估值之一，表示吃掉所有食物的难度。如果当前位置附近有胶囊，则考虑将其视为食物并加以考虑，且将其权重设置为更大的值以引导 Pacman 收集。其次，考虑幽灵的影响。如果幽灵距离太远，则不应该对 Pacman 产生干扰；如果幽灵距离太近，则应该尽力避免被吃掉，将幽灵的距离加入到评估值中（特别地，应该极力避免 Pacman 已经被幽灵吃掉的情况，因此在该情况下下该项估值是负无穷）；如果幽灵处于惊吓状态，则应该主动追逐它们以获取更高的得分。最后，将当前游戏状态的得分加入到评估值中，并返回总评估值。

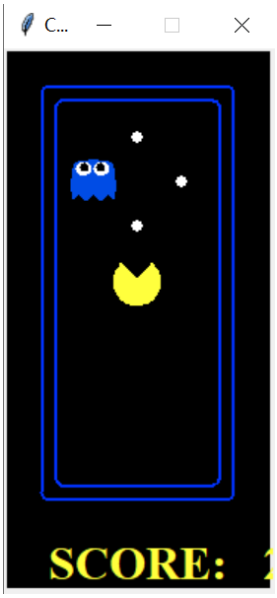
代码实现中，首先获取当前地图上所有食物、胶囊、幽灵和 Pacman 的位置和状态信息。然后，计算最小曼哈顿距离、食物距离和幽灵距离，并根据一定的规则将它们相加得到评估值。最后，将当前游戏状态的得分加入到评估值中，并返回总评估值。

```
def betterEvaluationFunction(currentGameState: GameState):
    """
    Your extreme ghost-hunting, pellet-nabbing, food-gobbling, unstoppable
    evaluation function (question 5).
    """
    "*** YOUR CODE HERE ***"
    foodList = currentGameState.getFood().asList()
    foodNum = currentGameState.getFood().count()
    capsuleList = currentGameState.getCapsules()
    pacmanPos = currentGameState.getPacmanPosition()
    ghosts = currentGameState.getGhostStates()

    # 选取食物中的最小曼哈顿距离作为估值之一
    minFoodDis = float("inf")
    foodDisSum = 1
    for foodPos in foodList:
        minFoodDis = min(minFoodDis, manhattanDistance(pacmanPos, foodPos))
        foodDisSum += manhattanDistance(pacmanPos, foodPos)

    # 如果 capsule 很近，就考虑吃
    for capsule in capsuleList:
        minFoodDis = min(minFoodDis, manhattanDistance(pacmanPos, capsule) /
2)  # capsule 的权重更大一些

    # 考虑幽灵的影响
    ghostValue = 0
    for ghost in ghosts:
        dis = manhattanDistance(pacmanPos, ghost.getPosition())
        if dis > 0:
            # 对于惊吓状态的幽灵 可以主动追逐
            if ghost.scaredTimer > 0:
                ghostValue += 10 / dis
            # 如果幽灵距离太远 则不应该干扰 Pacman
            else:
                ghostValue -= 10 / dis if dis < 3 else 0
        # 极力避免被幽灵吃
        else:
            ghostValue = float("-inf")

    return currentGameState.getScore() + 5 / minFoodDis + foodNum /
foodDisSum + ghostValue
```

4. 实验结果

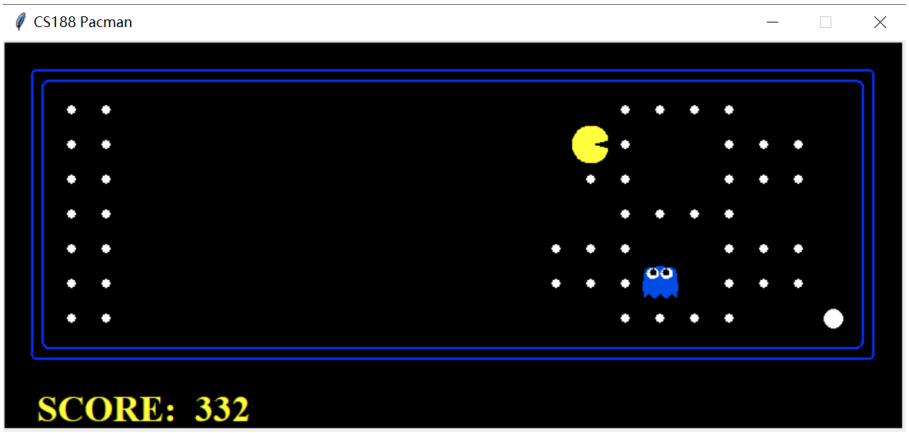
4.1. 问题 1 反射型智能体的估值函数

首先测试反射型智能体的估值函数效果。如下方图片所示，我测试了 Pacman 在不同大小的地图中对于分别在一个和两个幽灵情况下的躲避以及收集食物的能力。在调试过程中，我尝试了最近食物距离、幽灵距离等因素。当只考虑二者时，Pacman 可能由于收集了食物反而导致最近食物的曼哈顿距离变远估值反而降低，因此需要将食物数量也纳入考虑范围；此外，若幽灵距离不加限制的被加入 Pacman 的估值考量中，则幽灵很远时 Pacman 也会无意义地和幽灵进行近乎同步的运动，这也应该被避免。修改后可以看到，Pacman 均有较高的成功率能够躲避幽灵并成功收集食物，其取得的游戏得分也较高。




```
Pacman emerges victorious! Score: 553
Average Score: 553.0
Scores:        553.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
Pacman emerges victorious! Score: 1310
Average Score: 1310.0
Scores:        1310.0
Win Rate:      1/1 (1.00)
Record:        Win
```



下面测试问题 1 测试样例的运行情况，地图如上图所示。该地图中，Pacman 需要在没有障碍物的迷宫中躲避单个幽灵进行食物收集。

对于所有的测试样例均高分通过，截图如下：

```
Question q1
===========

Pacman emerges victorious! Score: 1220
Pacman emerges victorious! Score: 1239
Pacman emerges victorious! Score: 1212
Pacman emerges victorious! Score: 1226
Pacman emerges victorious! Score: 1245
Pacman emerges victorious! Score: 1218
Pacman emerges victorious! Score: 1230
Pacman emerges victorious! Score: 1232
Pacman emerges victorious! Score: 1237
Pacman emerges victorious! Score: 1230
Average Score: 1228.9
Scores:        1220.0, 1239.0, 1212.0, 1226.0, 1245.0, 1218.0, 1230.0, 1232.0, 1237.0, 1230.0
Win Rate:      10/10 (1.00)

### Question q1: 4/4 ###


Finished at 21:59:08

Provisional grades
==================
Question q1: 4/4
------------------
Total: 4/4

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

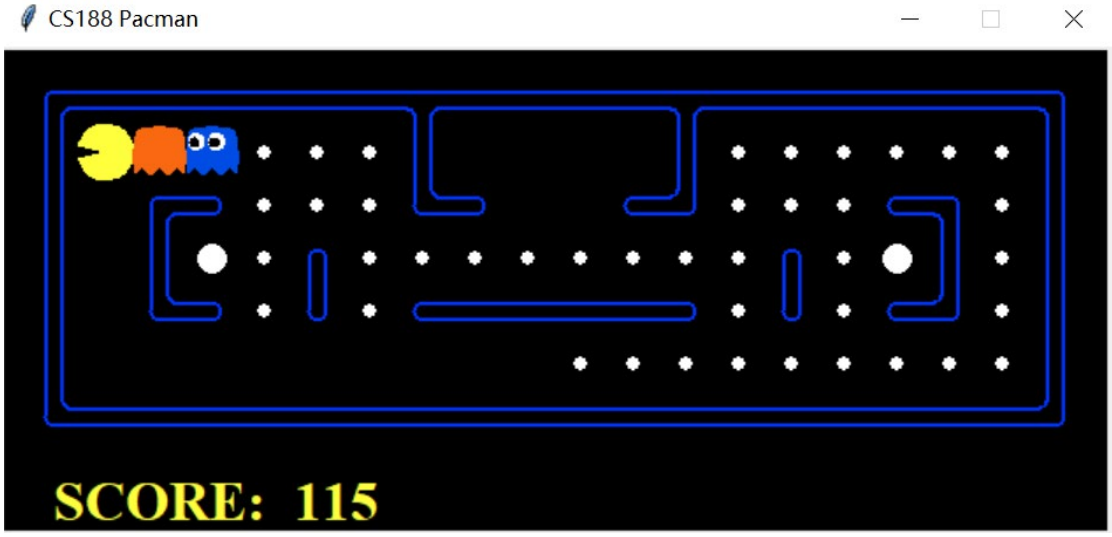
4.2. 问题 2 Minimax 搜索树


```
Pacman emerges victorious! Score: 516
Average Score: 516.0
Scores:        516.0
Win Rate:      1/1 (1.00)
Record:        Win
```


```
Pacman died! Score: -501
Average Score: -501.0
Scores:        -501.0
Win Rate:      0/1 (0.00)
Record:        Loss
```

上图中分别测试了在搜索深度为 4 时的小迷宫中 Pacman 对三个幽灵，以及在搜索深度为 3 的小迷宫中 Pacman 对两个幽灵的对局情况。在前者中，Pacman 有胜利的概率，但也有时会被捕获；在后者中，Pacman 会直接撞向其中一个幽灵结束对局。如题目描述中所说，这是由于 Pacman 为了避免时间流逝所带来的分数惩罚，因此倾向于选择直接结束比赛。后续的通过测试样例能说明 Minimax 算法已经被正确实现，我认为此处 Minimax 算法效果不好有两个原因：一是幽灵不一定每次都做出对 Pacman 最不利的决策，二是采用了默认的效果不佳的估值函数。

下面测试本题的测试样例。以问题 2 的第一个测试场景为例，其地图场景是如下图所示的迷宫，Pacman 将在两个幽灵的追捕下进行食物的高效收集。



对于所有的测试样例均通过，截图如下：

```
Question q2
===========

*** PASS: test_cases\q2\0-eval-function-lose-states-1.test
*** PASS: test_cases\q2\0-eval-function-lose-states-2.test
*** PASS: test_cases\q2\0-eval-function-win-states-1.test
*** PASS: test_cases\q2\0-eval-function-win-states-2.test
*** 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
```

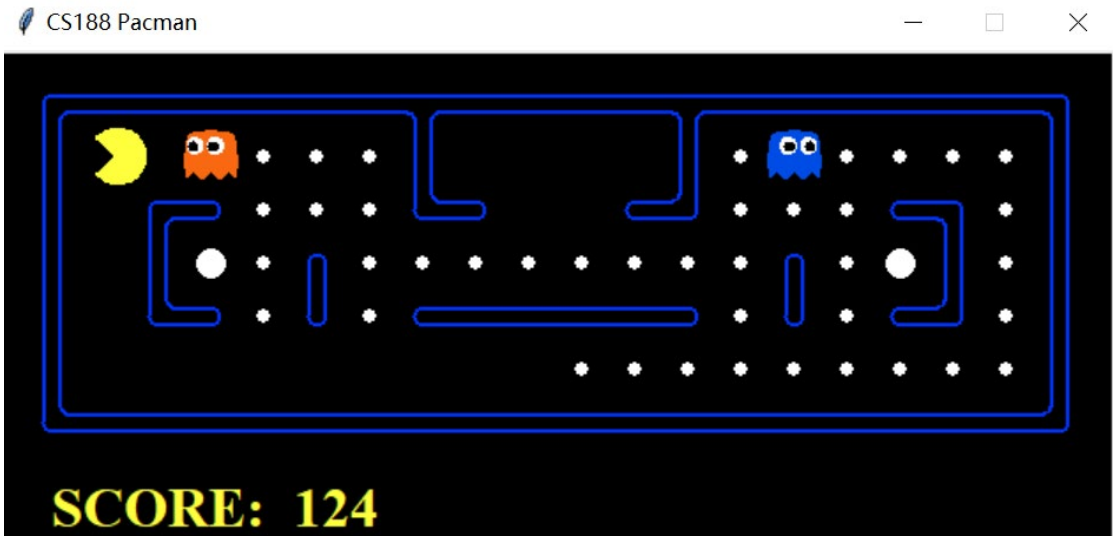
```
*** PASS: test_cases\q2\3-one-ghost-4level.test
*** PASS: test_cases\q2\4-two-ghosts-3level.test
*** PASS: test_cases\q2\5-two-ghosts-4level.test
*** PASS: test_cases\q2\6-tied-root.test
*** PASS: test_cases\q2\7-1a-check-depth-one-ghost.test
*** PASS: test_cases\q2\7-1b-check-depth-one-ghost.test
*** 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
*** PASS: test_cases\q2\8-pacman-game.test

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


Finished at 23:21:18

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

4.3. 问题 3 $\alpha-\beta$ 剪枝

首先测试 $\alpha-\beta$ 剪枝对于 Minimax 算法的效率提升。可以观察到，Pacman 在如下的 smallClassic 地图中每一步的决策时间较未剪枝的 Minimax 大大减少了。通过命令行执行：

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

可以观察到在搜索深度为 3 的情况下，经过 $\alpha-\beta$ 剪枝的算法下 Pacman 的每一步决策速度都大大提升了，其速度大概等同于搜索深度为 2 情况下未经剪枝的 Minimax 算法。当深度为 5 的情况下，未经剪枝的算法每一步决策时间都极长，但 $\alpha-\beta$ 剪枝能将每一步决策时间控制在一秒左右。



所有的测试样例均通过，截图如下：

```
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
*** PASS: test_cases\q3\8-pacman-game.test

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


Finished at 23:48:47

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

4.4. 问题 4 对手随机决策的 Expectimax 算法

再次测试问题 2 中的小地图 trappedClassic。在问题 2 中，Pacman 会由于找不到合理的规避路径而直接向幽灵冲撞，因此会观察到连续十次的游戏中，Pacman 均以失败告终。Expectimax 算法下，Pacman 不会出现上述情况，但是由于幽灵的决策是随机的，不能保证每次都不会冲撞到幽灵导致 Pacman 的死亡，因此大概有一半的游戏能够成功收集食物，一半的游戏会被幽灵捕获而死亡。具体情况如下：

Minimax 搜索的智能体表现：

```
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Average Score: -501.0
Scores:        -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0
Win Rate:      0/10 (0.00)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss
```

Expectimax 搜索的智能体表现：

```
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Pacman died! Score: -502
Pacman died! Score: -502
Pacman died! Score: -502
Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Pacman emerges victorious! Score: 532
Average Score: 15.0
Scores:        532.0, 532.0, 532.0, -502.0, -502.0, -502.0, -502.0, 532.0, -502.0, 532.0
Win Rate:      5/10 (0.50)
Record:        Win, Win, Win, Loss, Loss, Loss, Loss, Win, Loss, Win
```

对于所有的测试样例均通过，截图如下：

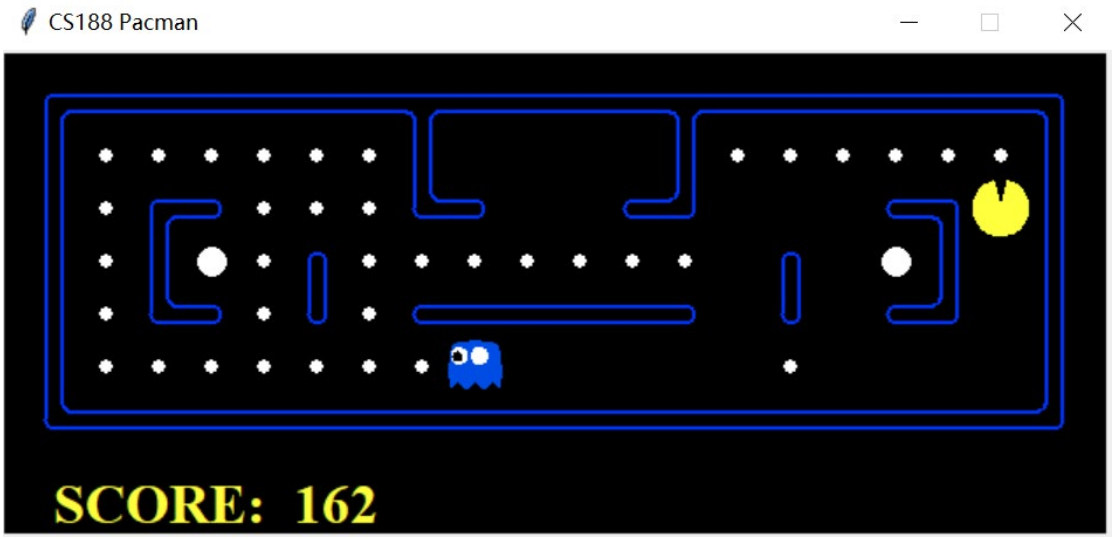
```
Question q4
===========

*** PASS: test_cases\q4\0-eval-function-lose-states-1.test
*** PASS: test_cases\q4\0-eval-function-lose-states-2.test
*** PASS: test_cases\q4\0-eval-function-win-states-1.test
*** PASS: test_cases\q4\0-eval-function-win-states-2.test
*** PASS: test_cases\q4\0-expectimax1.test
*** PASS: test_cases\q4\1-expectimax2.test
*** PASS: test_cases\q4\2-one-ghost-3level.test
*** PASS: test_cases\q4\3-one-ghost-4level.test
*** PASS: test_cases\q4\4-two-ghosts-3level.test
*** PASS: test_cases\q4\5-two-ghosts-4level.test
```

```
*** PASS: test_cases\q4\6-1a-check-depth-one-ghost.test
*** PASS: test_cases\q4\6-1b-check-depth-one-ghost.test
*** PASS: test_cases\q4\6-1c-check-depth-one-ghost.test
*** PASS: test_cases\q4\6-2a-check-depth-two-ghosts.test
*** PASS: test_cases\q4\6-2b-check-depth-two-ghosts.test
*** PASS: test_cases\q4\6-2c-check-depth-two-ghosts.test
*** PASS: test_cases\q4\7-pacman-game.test

### Question q4: 5/5 ###


Finished at 0:13:12

Provisional grades
==================
Question q4: 5/5
------------------
Total: 5/5
```

4.5. 问题 5 状态估值函数设计

在如下的地图中，我测试了问题 5 中的评估函数效果。在这个地图中，Pacman 需要在一个幽灵的追逐下尽可能高效地收集食物。此外，地图中还有胶囊，吃掉胶囊后，Pacman 可以主动追逐并吃掉幽灵，从而获得更高的分数。使用上述实现的评估函数对 Pacman 进行指导，结果表明在多次的对局中 Pacman 均能够比较高效地收集食物；此外，Pacman 在靠近胶囊附近时也能够主动收集并对幽灵展开追逐。

在 10 次对局中，Pacman 均能够成功躲避幽灵的追赶，并成功地收集到地图中的所有物品并取得胜利。这表明评估函数在 Pacman 地图中能够发挥出非常好的效果。



对于问题 5 的所有测试样例,Pacman 均能以较高的分数取得胜利（平均分数为 1082.3），具体展示如下图：

```
Question q5
===========

Pacman emerges victorious! Score: 1149
Pacman emerges victorious! Score: 1082
Pacman emerges victorious! Score: 1274
Pacman emerges victorious! Score: 1037
Pacman emerges victorious! Score: 921
Pacman emerges victorious! Score: 1215
Pacman emerges victorious! Score: 1121
Pacman emerges victorious! Score: 925
Pacman emerges victorious! Score: 976
Pacman emerges victorious! Score: 1123
Average Score: 1082.3
Scores:        1149.0, 1082.0, 1274.0, 1037.0, 921.0, 1215.0, 1121.0, 925.0, 976.0, 1123.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
*** PASS: test_cases\q5\grade-agent.test (6 of 6 points)
***     1082.3 average score (2 of 2 points)
***         Grading scheme:
***          < 500:  0 points
***         >= 500:  1 points
***         >= 1000:  2 points
***     10 games not timed out (1 of 1 points)
***         Grading scheme:
***          < 0:  fail
***         >= 0:  0 points
***         >= 10:  1 points
***     10 wins (3 of 3 points)
***         Grading scheme:
***          < 1:  fail
***         >= 1:  1 points
***         >= 5:  2 points
***         >= 10:  3 points

### Question q5: 6/6 ###


Finished at 8:53:17

Provisional grades
==================
Question q5: 6/6
------------------
Total: 6/6
```

5. 总结与分析

在本次 Project2 中，我主要探究了多智能体博弈中的几种算法，包括反射型智能体、Minimax 算法、$\alpha-\beta$ 剪枝算法、Expectimax 算法和评估函数的优化。通过这次 Project 的学习，我对多智能体博弈算法的使用与理解都有了更深入的了解。

针对问题 1 中的反射型智能体，我通过改进 multiAgents.py 中的 ReflexAgent，让其考虑食物位置、食物数量以及幽灵位置，以让 Pacman 表现出更好的性能。针对问题 2，我实现了 Minimax 算法，使得智能体可以在游戏中思考并选择最好的行动。在实现该算法时，我考虑了搜索深度并使用 self.evaluationFunction 进行局面评估，以保证智能体在游戏中的决策能够在较短时间内得出。针对问题 3，我同样在 multiAgents.py 中实现了 $\alpha-\beta$ 算法，

使得智能体可以更加高效地在游戏中进行决策。$\alpha-\beta$ 剪枝能够减少搜索空间，从而提高算法效率，保证智能体在游戏中的决策能够在更短的时间内得出。针对问题 4 中的 Expectimax 算法，我也通过在原先的 Min 层对所有状态值求算术平均值，使得智能体可以在游戏中进行随机决策。通过实现该算法，我使得智能体能够在游戏中更好地适应幽灵随机移动的情况。最后，针对问题 5 中的评估函数优化，我实现了一种评估函数，考虑到当前游戏状态、食物位置、胶囊位置和幽灵位置等因素，以提供更准确的预测。通过优化评估函数，我使得智能体在游戏中的表现更好，从而在游戏中获得更高的得分。下面我将分别对本次实验中应用的算法进行总结分析。

5.1. 对反射型智能体的总结分析

在本次实验中，我对反射型智能体进行了探究和改进，使其在游戏中表现更加出色。在此过程中，我也遇到了诸如在收集食物时由于曼哈顿距离的评估效果不佳导致的 Pacman 卡顿、将幽灵纳入考虑时容易过于敏感等问题。为了解决这些问题，我经过不断的调试和改进最终实现了一个能够轻松、可靠地通过许多不同测试的反射型智能体。

总的来说，反射型智能体是一种简单、实用的算法，其实现相对简单，只需要考虑当前和动作转移后状态的幽灵位置和食物位置等因素即可。此外，反射型智能体可以实时地进行决策和行动，可以适用于一些要求及时响应的场景。由于算法简单，因此反射型智能体的决策过程也较为清晰，易于解释和理解。反射型智能体也存在如下缺点：由于反射型智能体只考虑当前状态和局部信息，对于复杂的问题和大规模的搜索空间难以有效地解决。反射型智能体只进行局部搜索，无法对整个搜索空间进行全局搜索，因此在许多情况下可能无法找到最优解。

综上所述，反射型智能体适用于一些简单的问题和小规模的搜索空间，但是对于复杂的问题和大规模的搜索空间，需要使用更为复杂的搜索算法进行解决。因此在后续的实验中，将会进一步探究 Minimax、Expectimax 等复杂的搜索算法，以解决 Pacman 在应对更为复杂的问题时的决策。

5.2. 对 Minimax 搜索和 $\alpha-\beta$ 剪枝的总结分析

Minimax 搜索和 $\alpha-\beta$ 剪枝是博弈树搜索中经典的算法。在本次实验中，我对这两种算法进行了深入的学习和实践，并取得了一定的收获。

Minimax 搜索算法是一种在博弈树中搜索最佳决策的算法。它的基本思想是假设对手会采取最优策略，然后在此前提下，选择使得自己最优的策略。Minimax 算法是一种基于深度优先搜索的算法，其时间复杂度为指数级，因此在实际应用中需要进行一定的优化。$\alpha-\beta$ 剪枝算法是对 Minimax 搜索算法的优化，能够有效减少搜索空间，提高算法效率。该算法在搜索博弈树的过程中，记录一个最优解的上下界 α 与 β，通过剪枝来减少搜索空

间，从而提高搜索效率。此外，我也探究了 Minimax 算法与 $\alpha-\beta$ 剪枝的优缺点：对于 Minimax 算法，其优点在不限制搜索深度的前提下能够找到最优解，并且在完全信息的博弈中，该算法的表现是最好的。然而，Minimax 算法也有其缺点，例如时间复杂度高、无法应用于部分信息博弈中等，此外博弈的队手也并非一定有最优的决策。 $\alpha-\beta$ 剪枝算法能够很好的辅助 Minimax 算法，其能够减少博弈树的搜索空间，提高算法效率。

在本次实验的调试过程中，我也遇到了一些问题。在终止状态的判断上，我起初只考虑了搜索达到限制深度的情况，而忽略了在对局中 Pacman 收集食物从而取得胜利，或 Pacman 被幽灵捕获而对局失败的情况。在 $\alpha-\beta$ 剪枝时，由于起初对算法了解不够深入透彻，我对剪枝的条件以及 α、β 值的更新维护都犯了一些错误，导致剪枝错误或没有剪掉本能够剪的分支等情况。我最终都在不断调试下解决了这些问题。

综上所述，Minimax 搜索和 $\alpha-\beta$ 剪枝是博弈树搜索中常用的算法，在实践中具有广泛的应用。在选择算法时，需要根据实际情况选择适合的算法，并在实践中不断调试和优化算法，从而使得算法更加高效和准确。

5.3. 对 Expectimax 的总结分析

Expectimax 搜索算法是一种常用于博弈树搜索的算法，与问题 2 中的 Minimax 搜索不同，Expectimax 搜索考虑了对手可能采取的行动，而非总预测队手有最佳的选择。在 Expectimax 算法中，我们假设对手的行为是随机的，并且我们能够对其采取的行为进行概率分布的预测。我们对每个状态的期望值进行计算，然后采取使期望值最大化的行动。在该问题中，题目假设幽灵每一步的决策都是完全随机且概率相等的。

在 Expectimax 搜索中，我们需要计算每个可能的行动的期望值。这需要我们能够预测对手采取的行动，然后根据概率分布计算出每个行动的期望值。由于本题目中将幽灵决策简化为在所有的可能动作上做等概率的随机选择，因此只需要在幽灵决策时对所有可能的情况累计求和，再除以合法动作数，即对各个情况值做加权平均即可。

总的来说，Expectimax 搜索算法适用于博弈树搜索等场景。相对于 Minimax 搜索算法，Expectimax 搜索算法可以更好地应对对手的随机行为。但是，Expectimax 搜索算法的搜索空间较大，搜索效率较低；因此，也可以采用剪枝来进行优化。在实验中，我通过使用 Expectimax 搜索算法来解决 Pacman 迷宫问题，掌握了该算法的基本原理和实现方式，并通过优化技术提高了搜索效率，同时也提高了自己的编程和算法分析能力。

在完成本次 Project2 实验的过程中，我学习和实践了反射型智能体、Minimax、$\alpha-\beta$ 剪枝、Expectimax 以及评估函数等搜索算法的基本原理和实现方式。在改进 ReflexAgent 的过程中，我探究了食物位置和幽灵位置对于智能体表现的影响，并通过调试和改进，提

高了其在不同地图中的对局表现。在实现 Minimax 算法中，实现了相应的搜索策略，体会了智能体轮流决策的博弈方式。在问题 3 中，$\alpha-\beta$ 剪枝算法的实现使得智能体可以更加高效地进行决策，并减少了搜索空间。而 Expectimax 算法的实现则使得智能体可以进行随机决策，并应对一些如幽灵随机移动的难以确定的情况。在实现问题 5 的评估函数时，我考虑了当前游戏状态、食物位置、胶囊位置和幽灵位置等因素，以提供更准确的预测。通过调试和改进，我不断提高了评估函数的准确性，并改善了智能体在游戏中的表现。

在本次实验的过程中，我不仅深入了解了对抗搜索算法的优点、局限性以及适用场景，还培养了解决问题的能力。我需要根据问题要求选择合适的算法，编写代码并测试其正确性和效率。不管是对算法本身的理解、撰写，还是对 Python 语法的学习与熟悉，不断撰写、纠错、调试的过程都让我逐渐提高了自己的分析问题、寻找解决方案和调试代码的能力，以及在解决问题时需要的耐心和毅力。

总的来说，本次实验使我深入了解了人工智能领域的搜索算法，并为我未来在该领域的学习和研究奠定了坚实的基础。通过本次实验的学习和实践，我对未来的学习更充满信心，也希望自己在未来能够通过更多的学习与实践不断进步！