多智能体搜索实验报告

蒲今（学号：241880622；邮箱：1738831615@qq.com）

摘要

本实验围绕多智能体搜索算法展开，深入实现了基于极小化极大（Minimax）、Alpha-Beta 剪枝（Alpha-Beta Pruning）、期望最大（Expectimax）搜索算法的吃豆人智能体，并设计了一个高效的评价函数（Evaluation Function）。实验通过在不同地图环境下对比分析各算法的性能表现，系统验证了算法在对抗性环境和随机环境下的有效性与适用性。实验结果表明，Alpha-Beta 剪枝能显著提升搜索效率，Expectimax 在随机环境中表现更优，而设计的综合评价函数通过融合多种环境因素，有效提升了智能体的决策能力与游戏表现。

关键词：多智能体搜索；极小化极大；Alpha-Beta 剪枝；期望最大；评价函数；吃豆人游戏；人工智能

1. 引言

在人工智能的经典问题中，**博弈搜索（Game Tree Search）** 是智能体决策的重要手段。通过搜索状态空间，智能体能够在复杂环境中选择最优行动。

本实验基于 **Pacman 游戏**：其中吃豆人（Pacman）作为主角需要尽可能多地吃掉食物，同时避免被幽灵（Ghosts）捕获。Pacman 游戏环境具有典型的对抗性特征，这为多智能体决策提供了优秀的测试场景。

本实验的具体目标包括：

1. **极小化极大搜索**：假设对手总是采取最优策略，通过递归搜索来寻找最大化我方效用的策略。
2. **Alpha-Beta 剪枝**：在 Minimax 的基础上引入剪枝机制，有效减少状态空间的搜索，提高效率。
3. **期望最大搜索**：在对手策略非最优的情况下，将其建模为随机决策，通过期望值进行评估。
4. **更优的启发式评价函数**：设计一个高效的评价函数，综合评估游戏状态的多个维度

该实验加深了我们对 **人工智能对抗搜索算法** 的理解，结合课程文档，并参考 Russell 和 Norvig 的《Artificial Intelligence: A Modern Approach》[1]以及 UC Berkeley CS188 项目文档[2]，我们系统地实现并验证了这些算法。

2. 实验内容与实现

2.1 极小化极大搜索（Minimax）[3]

极小化极大搜索是处理对抗性决策问题的经典算法，其基本思想是假设对手总是做出对自己最不利的决策，从而在有限的搜索深度内选择最优的行动策略。在本实现中，算法递归地遍历所有可能的动作序列，交替计算最大值层（吃豆人）和最小值层（幽灵）的评分。

2.1.1 算法设计与实现

- **实现细节**：在 `multiAgents.py` 的 `MinimaxAgent` 中实现了 `getAction`，递归函数 `minimax` 负责展开博弈树。
- **时间复杂度**：状态分支因子设为 `b`，最大深度为 `d`，总 agent 数为 `n`。时间复杂度为 `O(b^(n·d))`，空间复杂度取决于递归深度。
- **算法核心结构（简化）如下**：

```
class MinimaxAgent(MultiAgentSearchAgent):
    def getAction(self, gameState: GameState):
        def minimax(agentIndex, depth, gameState):
            if gameState.isWin() or gameState.isLose() or depth == self.depth:
                return self.evaluationFunction(gameState)

            if agentIndex == 0:
                return max(minimax(1, depth,
gameState.generateSuccessor(agentIndex, a))
                           for a in gameState.getLegalActions(agentIndex))
            else:
                nextAgent = (agentIndex + 1) % gameState.getNumAgents()
                nextDepth = depth + 1 if nextAgent == 0 else depth
                return min(minimax(nextAgent, nextDepth,
gameState.generateSuccessor(agentIndex, a))
                           for a in gameState.getLegalActions(agentIndex))

        bestAction = max(gameState.getLegalActions(0),
                         key=lambda a: minimax(1, 0,
gameState.generateSuccessor(0, a)))
        return bestAction
```

2.1.2 实验结果与分析

运行 `python autograder.py -q q2` 样例全部通过，实验结果表明该算法能够有效规避幽灵陷阱，在简单场景下表现合理，但随着搜索深度增加计算时间呈指数级上升。尽管受限于搜索深度，算法在复杂地图中难以预见长序列对抗，但其能够正确处理多个幽灵的决策过程，验证了实现的正确性。

2.2 Alpha-Beta 剪枝（Alpha-Beta Pruning）[3]

Alpha-Beta剪枝是极小化极大算法的重要优化技术，通过在搜索过程中剪除不可能影响最终决策的分支，显著减少搜索空间，提高算法效率，而不影响最终决策结果。

2.2.1 算法设计与实现

- **算法原理**
  - 在 Minimax 基础上增加 α (当前最佳下界) 与 β (当前最佳上界)：
  - 当 MAX 节点的值 ≥ β 时，剪去剩余分支；
  - 当 MIN 节点的值 ≤ α 时，剪去剩余分支。

    - 这样避免了不必要的状态扩展。

- **实现细节**：在 `AlphaBetaAgent` 中实现 `alphaBeta` 递归函数，传入参数 `(alpha, beta)`，在每次比较时更新。
- **时间复杂度**：最优情况下时间复杂度降为 `O(b^(n·d/2))`，比 `Minimax` 快一倍以上。
- **算法核心结构（简化）如下**：

```
class AlphaBetaAgent(MultiAgentSearchAgent):
    def getAction(self, gameState: GameState):
        def alphaBeta(agentIndex, depth, state, alpha, beta):
            if state.isWin() or state.isLose() or depth == self.depth:
                return self.evaluationFunction(state)

            if agentIndex == 0:
                value = float('-inf')
                for a in state.getLegalActions(agentIndex):
                    value = max(value, alphaBeta(1, depth,
state.generateSuccessor(agentIndex, a), alpha, beta))
                    if value > beta: return value
                    alpha = max(alpha, value)
                return value
            else:
                nextAgent = (agentIndex + 1) % state.getNumAgents()
                nextDepth = depth + 1 if nextAgent == 0 else depth
                value = float('inf')
                for a in state.getLegalActions(agentIndex):
                    value = min(value, alphaBeta(nextAgent, nextDepth,
state.generateSuccessor(agentIndex, a), alpha, beta))
                    if value < alpha: return value
                    beta = min(beta, value)
                return value

        bestAction = max(gameState.getLegalActions(0),
                         key=lambda a: alphaBeta(1, 0,
gameState.generateSuccessor(0, a),
                                                 float('-inf'), float('inf')))
        return bestAction
```

2.2.2 实验结果与分析

运行在 `smallClassic` 地图上测试表明，深度为 `3` 的 `Alpha-Beta` 搜索速度与深度为 `2` 的 `Minimax` 相当，显著提升了效率，同时保持与完整 `Minimax` 一致的决策结果，验证了剪枝在不影响决策质量的前提下能够支持更深层次搜索，并在复杂地图中优势更为明显。代码通过了 `python autograder.py -q q3` 的全部测试，并在 `python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic` 的可视化实验中表现符合预期。

2.3 期望最大搜索（Expectimax）

期望最大搜索（Expectimax）是Minimax算法的变体，适用于对手行为不完全理性的环境。它不再假设对手总是做出最优决策，而是考虑对手可能采取的各种行动及其概率分布，计算期望效用值。

2.3.1 算法设计与实现

- **算法原理**：区别于 `Minimax` 假设对手总是最优，这里假设鬼魂随机选择动作。鬼魂节点返回所有子节点效用值的期望，而非最小值。
- **实现细节**：在 `ExpectimaxAgent` 中实现 `expectimax`：
  - Pacman 节点取最大值；
  - 鬼魂节点取平均值，概率为 $\dfrac{1}{|legalActions|}$。
- **算法核心结构（简化）如下**：

```
class ExpectimaxAgent(MultiAgentSearchAgent):
    def getAction(self, gameState: GameState):
        def expectimax(agentIndex, depth, state):
            if state.isWin() or state.isLose() or depth == self.depth:
                return self.evaluationFunction(state)

            if agentIndex == 0:
                return max(expectimax(1, depth,
state.generateSuccessor(agentIndex, a))
                           for a in state.getLegalActions(agentIndex))
            else:
                nextAgent = (agentIndex + 1) % state.getNumAgents()
                nextDepth = depth + 1 if nextAgent == 0 else depth
                actions = state.getLegalActions(agentIndex)
                p = 1 / len(actions)
                return sum(expectimax(nextAgent, nextDepth,
state.generateSuccessor(agentIndex, a)) * p
                           for a in actions)

        return max(gameState.getLegalActions(0),
                   key=lambda a: expectimax(1, 0, gameState.generateSuccessor(0,
a)))
```

2.3.2 实验结果与分析

在 `trappedClassic` 地图的对比实验中，`Expectimax` 的胜率约 50%，而 `AlphaBetaAgent` 始终失败，显示出 `Expectimax` 在随机环境中的适应性优势。尽管其计算复杂度与 Minimax 相当，但通过期望建模能更好应对非理性或不完全信息的对手。 实验通过 `python autograder.py -q q4` 验证了算法的正确性与优势。

2.4 评价函数设计（Evaluation Function）

评价函数是搜索算法的重要组成部分，用于在非终局状态评估局面优劣。我们设计了一个综合多维度信息的评价函数，考虑以下因素：食物距离、幽灵状态、胶囊效果、行动选项等。

2.4.1 评价函数设计与实现

- **设计思路**
  - **基础得分**：以 `currentGameState.getScore()` 为基准。
  - **食物启发**：鼓励靠近最近食物，惩罚剩余食物数。
  - **胶囊启发**：鼓励靠近胶囊，惩罚剩余数量。
  - **鬼魂启发**：若鬼魂受惊，则鼓励靠近并追击；若鬼魂正常，则保持距离。
  - **行动自由度**：奖励可选动作数，避免陷入死路。
- **公式结构（简化）**： $$ Eval(s) = 1.2 \cdot Score(s) + f_{food}(s) + f_{capsule}(s) + f_{ghost}(s) + f_{mobility}(s) $$
- **具体代码实现**：

```
def betterEvaluationFunction(currentGameState):
    pacmanPos = currentGameState.getPacmanPosition()
    foodGrid = currentGameState.getFood()
    ghostStates = currentGameState.getGhostStates()
    capsules = currentGameState.getCapsules()
    legalActions = currentGameState.getLegalActions(0)

    score = 1.2 * currentGameState.getScore()

    foodList = foodGrid.asList()
    if foodList:
        closestFood = min([manhattanDistance(pacmanPos, food) for food in
foodList])
        score += 5.0 / (closestFood + 1)
        score -= 0.5 * closestFood
    else:
        score += 2.0

    score -= 8.0 * len(foodList)
    score -= 20.0 / (len(foodList) + 1)

    if capsules:
        closestCapsule = min([manhattanDistance(pacmanPos, capsule) for capsule in
capsules])
        score += 6.0 / (closestCapsule + 1)
    else:
        score += 10.0

    score -= 15.0 * len(capsules)

    for ghost in ghostStates:
        ghostPos = ghost.getPosition()
        distance = manhattanDistance(pacmanPos, ghostPos)

        if ghost.scaredTimer > 0:
            score += 80.0 / (distance + 1)
            score += max(0, 25 - distance)
```

```
            score += 1.5 * len(legalActions)
        else:
            if distance < 2:
                score -= 5.0
            elif distance < 6:
                score -= 1.0 / (distance + 1)

    if len(legalActions) == 1:
        score -= 5

    score += 0.5 * len(legalActions)

    return score
```

2.4.2 实验结果与分析

运行 `python autograder.py -q q5` 结果如下：

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

Pacman emerges victorious! Score: 1372
Pacman emerges victorious! Score: 1359
Pacman emerges victorious! Score: 1367
Pacman emerges victorious! Score: 1358
Pacman emerges victorious! Score: 1369
Pacman emerges victorious! Score: 1368
Pacman emerges victorious! Score: 1374
Pacman emerges victorious! Score: 1380
Pacman emerges victorious! Score: 1374
Pacman emerges victorious! Score: 1360
Average Score: 1368.1
Scores:        1372.0, 1359.0, 1367.0, 1358.0, 1369.0, 1368.0, 1374.0, 1380.0,
1374.0, 1360.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win

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

Finished at 20:54:24

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

- **综合表现**：在 `smallClassic` 地图上，使用该评价函数的 `Expectimax` 智能体在 **100%** 的情况下能吃掉所有豆子，平均得分约 **1368** 分。

- **因素权重**：通过调整不同因素的权重系数，优化了评价函数的平衡性，使其在不同场景下都能做出合理决策。
- **计算效率**：评价函数的计算复杂度控制在合理范围内，不会显著影响搜索算法的整体性能。
- **泛化能力**：在不同地图和幽灵行为模式下，评价函数都表现出良好的适应性和稳定性。

---

3. 结束语

本实验通过系统实现三种多智能体搜索算法和一个综合评价函数，深入探究了对抗性搜索与随机环境下的决策机制。实验结果表明，算法选择与评价函数设计对智能体性能具有显著影响，不同算法适用于不同特性的环境。

- Minimax 保证了对抗最优对手时的正确性；
- Alpha-Beta 剪枝显著提升效率；
- Expectimax 在处理随机性对手时更具优势；
- 改进的评价函数提升了吃豆人智能体的整体表现。

本实验不仅提高了我们的算法实现能力，更重要的是培养了分析和解决复杂决策问题的系统性思维，为后续人工智能领域的研究和实践奠定了坚实基础。

致谢

首先，衷心感谢课程指导教师对本实验的悉心指导。同时感谢助教团队在实验过程中给予的技术支持与宝贵建议。也感谢项目开发者提供的吃豆人游戏框架和测试平台。在此向所有为本实验提供直接或间接帮助的人们表示最诚挚的感谢！

参考文献 · References

[1] UC Berkeley CS188: Artificial Intelligence Project.
[2] Russell, S., & Norvig, P. (2010). Artificial Intelligence: A Modern Approach. Prentice Hall.
[3] Minimax Algorithm