人 工 智 能 实 验 报 告

实验名称： 经典搜索

学员姓名: 程景愉

学　号: 202302723005

实验日期: 2025.12.02

国防科技大学教育训练部制

《本科实验报告》填写说明

实验报告内容编排应符合以下要求：

（1）采用 A4（21cm×29.7cm）白色复印纸，单面黑字。上下左右各侧的页边距均为 3cm；缺省文档网格：字号为小 4 号，中文为宋体，英文和阿拉伯数字为 Times New Roman，每页 30 行，每行 36 字；页脚距边界为 2.5cm，页码置于页脚、居中，采用小 5 号阿拉伯数字从 1 开始连续编排，封面不编页码。

（2）报告正文最多可设四级标题，字体均为黑体，第一级标题字号为 4 号，其余各级标题为小 4 号；标题序号第一级用“一、”、“二、”……，第二级用“（一）”、“（二）”……，第三级用“1.”、“2.”……，第四级用“（1）”、“（2）”……，分别按序连续编排。

（3）正文插图、表格中的文字字号均为 5 号。

0 目录

1 实验介绍

本项目是 UC Berkeley CS188 人工智能课程的第一个项目，主要任务是实现多种通用搜索算法，并将其应用于解决经典游戏吃豆人（Pacman）中的路径规划问题。项目涵盖了从基础的无信息搜索算法（如 DFS、BFS）到有信息搜索算法（如 UCS、A*），以及针对特定问题（如访问所有角落、吃掉所有食物）的状态表示和启发式函数设计。通过本项目，我们将深入理解搜索算法的原理，并学会如何利用它们来解决实际问题。

2 实验内容

本次实验内容涵盖了 8 个递进的编程任务，具体如下:

1. **Q1:** 深度优先搜索 **(DFS)**：利用深度优先搜索算法寻找迷宫中的固定食物点。
2. **Q2:** 广度优先搜索 **(BFS)**：利用广度优先搜索算法，找到到达目标所需行动次数最少的路径。
3. **Q3:** 一致代价搜索 **(UCS)**：实现一致代价搜索，以处理具有不同行动代价的路径，找到总代价最小的路径。
4. **Q4: A*** 搜索：实现 A*搜索算法，该算法结合了路径的实际代价和启发式评估，并使用曼哈顿距离作为启发式函数进行测试。
5. **Q5:** 角落问题：设计一个新的搜索问题，要求吃豆人以最短路径访问迷宫的全部四个角落。这需要定义一个合适的状态空间。
6. **Q6:** 角落启发式：为角落问题设计一个一致且可接受的启发式函数，以加速 A*搜索过程。
7. **Q7:** 食物启发式：为“吃完所有食物”这一复杂问题设计一个高效的启发式函数，以在可接受的时间内找到优良路径。
8. **Q8:** 寻找最近点路径：实现一个次优的贪心策略，让吃豆人总是前往最近的食物点，作为一种快速解决复杂问题的近似方法。

3 实验要求

本项目要求在 `search.py` 和 `searchAgents.py` 两个文件中根据指引补全代码，核心要求如下:

1. 通用搜索算法：在 `search.py` 中实现 `depthFirstSearch`, `breadthFirstSearch`, `uniformCostSearch`, 和 `aStarSearch` 四个核心搜索算法。所有搜索函数都需要返回一个动作（action）列表，该列表能够引导吃豆人从起点到达目标。
2. 数据结构：必须使用项目框架中 `util.py` 提供的 `Stack`, `Queue` 和 `PriorityQueue` 数据结构，以确保与自动评分器的兼容性。
3. 问题建模：在 `searchAgents.py` 中，需要为角落问题（`CornersProblem`）选择并实现一个不包含冗余信息的高效状态表示。
4. 启发式函数设计：为角落问题和食物问题设计的启发式函数（`cornersHeuristic` 和 `foodHeuristic`）必须是无不足道的（non-trivial）、非负且一致的（consistent）。启发式函数的性能将根据其在 A*搜索中扩展的节点数量进行评分。

5. 目标测试：为 AnyFoodSearchProblem 补全目标测试函数 isGoalState，使其能够正确判断是否到达任意一个食物所在的位置。

4 实验步骤与实现

4.1 Q1: 深度优先搜索 (DFS)

实现思路：深度优先搜索(DFS)优先探索最深的节点。我们使用图搜索版本，即记录已访问的节点以避免冗余搜索和无限循环。数据结构上，DFS 采用栈（Stack）来实现“后进先出”（LIFO）的节点扩展顺序。

核心代码 (search.py):

```
def depthFirstSearch(problem: SearchProblem):
    fringe = util.Stack()
    visited = set()
    startState = problem.getStartState()
    fringe.push((startState, []))

    while not fringe.isEmpty():
        currentState, actions = fringe.pop()

        if currentState in visited:
            continue

        visited.add(currentState)

        if problem.isGoalState(currentState):
            return actions

        successors = problem.getSuccessors(currentState)

        for successor, action, cost in successors:
            if successor not in visited:
                newActions = actions + [action]
                fringe.push((successor, newActions))

    return []
```

测试指令:

```
python autograder.py -q q1
python pacman.py -l mediumMaze -p SearchAgent -a fn=dfs
```

4.2 Q2: 广度优先搜索 (BFS)

实现思路：广度优先搜索(BFS)逐层扩展节点，确保找到行动次数最少的路径。同样采用图搜索版本。数据结构上，BFS 采用队列（Queue）来实现“先进先出”（FIFO）的节点扩展顺序。

核心代码 (search.py):

```
def breadthFirstSearch(problem: SearchProblem):
    fringe = util.Queue()
    visited = set()
    startState = problem.getStartState()
    fringe.push((startState, []))

    while not fringe.isEmpty():
        currentState, actions = fringe.pop()

        if currentState in visited:
            continue

        visited.add(currentState)

        if problem.isGoalState(currentState):
            return actions

        successors = problem.getSuccessors(currentState)

        for successor, action, cost in successors:
            if successor not in visited:
                newActions = actions + [action]
                fringe.push((successor, newActions))

    return []
```

测试指令:

```
python autograder.py -q q2
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
```

4.3 Q3: 一致代价搜索 (UCS)

实现思路: 一致代价搜索(UCS)扩展总路径代价最小的节点, 从而保证找到最优(总代价最低)的路径。它通过使用优先队列(PriorityQueue)来实现, 节点的优先级由其累积路径代价决定。

核心代码 (search.py):

```
def uniformCostSearch(problem: SearchProblem):
    fringe = util.PriorityQueue()
    visited = {}
    startState = problem.getStartState()
    fringe.push((startState, [], 0), 0)

    while not fringe.isEmpty():
        currentState, actions, currentCost = fringe.pop()

        if currentState in visited and currentCost >= visited[currentState]:
            continue

        visited[currentState] = currentCost

        if problem.isGoalState(currentState):
```

```
            return actions

        successors = problem.getSuccessors(currentState)

        for successor, action, stepCost in successors:
            newCost = currentCost + stepCost
            newActions = actions + [action]
            fringe.push((successor, newActions, newCost), newCost)

    return []
```

测试指令:

```
python autograder.py -q q3
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
```

4.4 Q4: A* 搜索

实现思路: A*搜索是 UCS 的扩展，它在评估节点时不仅考虑已付出的代价$g(n)$，还引入了对未来代价的估计，即启发式函数$h(n)$。节点优先级由$f(n) = g(n) + h(n)$决定。这使得 A* 能够更智能地朝向目标进行搜索。

核心代码 (search.py):

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
    fringe = util.PriorityQueue()
    visited = {}
    startState = problem.getStartState()
    startHeuristic = heuristic(startState, problem)
    fringe.push((startState, [], 0), startHeuristic)

    while not fringe.isEmpty():
        currentState, actions, currentCost = fringe.pop()

        if currentState in visited and currentCost >= visited[currentState]:
            continue

        visited[currentState] = currentCost

        if problem.isGoalState(currentState):
            return actions

        successors = problem.getSuccessors(currentState)

        for successor, action, stepCost in successors:
            newCost = currentCost + stepCost
            newHeuristic = heuristic(successor, problem)
            fValue = newCost + newHeuristic
            newActions = actions + [action]
            fringe.push((successor, newActions, newCost), fValue)

    return []
```

测试指令:

```
python autograder.py -q q4
python pacman.py -l bigMaze -z .5 -p SearchAgent -a
fn=astar,heuristic=manhattanHeuristic
```

4.5 Q5: 角落问题 (Corners Problem)

实现思路：为了解决访问所有四个角落的问题，我们需要设计一个能够追踪 Pacman 位置和哪些角落已被访问的状态表示。一个合适的状态是 (position, visited_corners)，其中 position 是 (x, y) 坐标，visited_corners 是一个布尔元组，记录四个角落各自的访问情况。

核心代码 (searchAgents.py):

```
// 状态表示：(当前位置, 已访问的角落元组)
// getStartState
def getStartState(self):
    cornersVisited = tuple([self.startingPosition == corner for corner in
self.corners])
    return (self.startingPosition, cornersVisited)

// isGoalState: 检查是否所有角落都已访问
def isGoalState(self, state: Any):
    _, cornersVisited = state
    return all(cornersVisited)

// getSuccessors: 生成后继状态，并更新角落访问信息
def getSuccessors(self, state: Any):
    successors = []
    currentPosition, cornersVisited = state

    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
Directions.WEST]:
        x, y = currentPosition
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)

        if not self.walls[nextx][nexty]:
            nextPosition = (nextx, nexty)
            newCornersVisited = list(cornersVisited)

            for i, corner in enumerate(self.corners):
                if nextPosition == corner and not newCornersVisited[i]:
                    newCornersVisited[i] = True

            successorState = (nextPosition, tuple(newCornersVisited))
            successors.append((successorState, action, 1))

    self._expanded += 1
    return successors
```

测试指令:

```
python autograder.py -q q5
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

4.6 Q6: 角落启发式 (Corners Heuristic)

实现思路：为角落问题设计一个一致且可接受的启发式函数，关键是估计从当前状态到达目标状态（所有角落被访问）的最小代价。一个有效的策略是计算当前位置到所有未访问角落的“某种”距离。我们采用的策略是：启发式的值取“当前位置到最远的未访问角落的曼哈顿距离”和“所有未访问角落两两之间最远曼哈顿距离”中的较大者。这确保了启发式是可接受的，因为它低估了必须走过的总路程。

核心代码 (searchAgents.py):

```
def cornersHeuristic(state: Any, problem: CornersProblem):
    currentPosition, cornersVisited = state
    corners = problem.corners

    if all(cornersVisited):
        return 0

    unvisitedCorners = []
    for i, corner in enumerate(corners):
        if not cornersVisited[i]:
            unvisitedCorners.append(corner)

    if not unvisitedCorners:
        return 0

    maxDistance = 0
    for corner in unvisitedCorners:
        distance = util.manhattanDistance(currentPosition, corner)
        maxDistance = max(maxDistance, distance)

    maxCornerDistance = 0
    for i in range(len(unvisitedCorners)):
        for j in range(i + 1, len(unvisitedCorners)):
            distance = util.manhattanDistance(unvisitedCorners[i],
unvisitedCorners[j])
            maxCornerDistance = max(maxCornerDistance, distance)

    return max(maxDistance, maxCornerDistance)
```

测试指令:

```
python autograder.py -q q6
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

4.7 Q7: 食物启发式 (Food Heuristic)

实现思路：为“吃掉所有食物”问题设计启发式函数更具挑战性。状态包含当前位置和食物分布的网格。一个好的启发式需要有效估计吃掉所有剩余食物的最小步数。我们结合了多种策略来构造一个更强的启发式：取“当前位置到最远食物的曼哈顿距离”、“剩余食物中两两之间最远的曼哈顿距离”以及“剩余食物的数量”这三者的最大值。这个值仍然是真实代价的下界，保证了可接受性和一致性。

核心代码 (searchAgents.py):

```
def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem):
    position, foodGrid = state
    foodList = foodGrid.asList()

    if not foodList:
        return 0

    maxDistance = 0
    for food in foodList:
        distance = util.manhattanDistance(position, food)
        maxDistance = max(maxDistance, distance)

    maxFoodDistance = 0
    if len(foodList) > 1:
        for i in range(len(foodList)):
            for j in range(i + 1, len(foodList)):
                distance = util.manhattanDistance(foodList[i], foodList[j])
                maxFoodDistance = max(maxFoodDistance, distance)

    foodCount = len(foodList)

    return max(maxDistance, maxFoodDistance, foodCount)
```

测试指令:

```
python autograder.py -q q7
python pacman.py -l trickySearch -p AStarFoodSearchAgent
```

4.8 Q8: 寻找最近点路径 (Closest Dot Search)

实现思路: 这是一个贪心算法，它不保证找到全局最优解，但通常能快速找到一个较好的解。策略是: 重复寻找并移动到距离当前位置最近的食物点，直到所有食物被吃完。这个任务的核心是实现 findPathToClosestDot 函数。我们可以定义一个 AnyFoodSearchProblem，其目标是到达任意一个食物点，然后使用 BFS 来找到到达最近食物的最短路径。

核心代码 (searchAgents.py):

```
// AnyFoodSearchProblem 的目标测试
def isGoalState(self, state: Tuple[int, int]):
    x,y = state
    return self.food[x][y]

// findPathToClosestDot 函数实现
def findPathToClosestDot(self, gameState: pacman.GameState):
    problem = AnyFoodSearchProblem(gameState)
    // BFS保证找到步数最少的路径，即到达最近的食物
    path = search.bfs(problem)
    return path
```

测试指令:

```
python autograder.py -q q8
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
```

5 实验结果

本项目的所有 8 个任务均已成功实现, 并通过了自动评分器（autograder）的所有测试用例, 最终取得了 **25/25** 的满分成绩。自动评分器给出的最终成绩摘要如下:

```
Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 3/3
Question q6: 3/3
Question q7: 4/4
Question q8: 3/3
------------------
Total: 25/25
```

以下是各部分任务的详细测试结果摘要:

1. **Q1:** 深度优先搜索
   - 所有测试用例通过。
   - 在 `mediumMaze` 迷宫中找到一条长度为 130 的路径，扩展了 146 个节点。
2. **Q2:** 广度优先搜索
   - 所有测试用例通过。
   - 在 `mediumMaze` 迷宫中找到长度为 68 的最短路径，扩展了 269 个节点。
3. **Q3:** 一致代价搜索
   - 所有测试用例通过。
   - 在不同代价的迷宫中（`mediumDottedMaze`, `mediumScaryMaze`）均能找到最优路径。
4. **Q4:** A*搜索
   - 所有测试用例通过。
   - 在 `bigMaze` 中，A* 搜索扩展的节点数（约 549）少于 UCS（约 620），展现了启发式的有效性。
5. **Q5:** 角落问题
   - 所有测试用例通过。
   - 在 `mediumCorners` 上，BFS 扩展了接近 2000 个节点。
6. **Q6:** 角落启发式
   - 所有测试用例通过，启发式满足一致性。
   - 在 `mediumCorners` 上，使用 A*和该启发式将扩展节点数减少到 **961** 个，满足了评分的最高要求。
7. **Q7:** 食物启发式
   - 18 个测试用例全部通过，启发式满足一致性。
   - 在 `trickySearch` 上，扩展节点数为 **8763** 个，达到了满分（4/4）要求。
8. **Q8:** 寻找最近点路径
   - 所有测试用例通过，贪心策略能够成功吃掉所有食物。

6 实验总结

通过本次实验，我成功实现了包括深度优先搜索（DFS）、广度优先搜索（BFS）、一致代价搜索（UCS）和 A*搜索在内的多种经典搜索算法，并将其应用于解决 Pacman 游戏中的路径规划问题。

在实验过程中，我遇到的主要挑战在于为复杂问题设计高效、一致且可接受的启发式函数，特别是 Q6 的角落问题和 Q7 的食物收集问题。通过对问题进行抽象建模，并反复迭代优化启发式策略，最终设计出的启发式函数在满足一致性的前提下，显著减少了搜索节点的扩展数量，达到了评分要求。例如，在 Q7 中，通过比较“到最远食物的距离”、“食物间的最大距离”和“食物数量”等多个策略，并取其最大值作为启发式，最终将扩展节点数控制在了 9000 以内。

通过这个项目，我不仅深入掌握了各种搜索算法的原理、实现细节及其适用场景，还深刻理解了状态空间表示、启发式函数设计（可接受性与一致性）对搜索性能的决定性影响。这些知识和经验对于解决更广泛的人工智能规划问题具有重要的实践意义。未来，可以尝试引入更高级的启发式策略（如基于最小生成树的启发式）或探索其他搜索算法变体，以期进一步提升求解效率。