哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验报告

开课学期：2025 春季

课程名称：人工智能（实验）

实验名称：搜索策略 pacman

实验性质：综合设计型

实验学时：4 地点：T2612

学生班级：6

学生学号：220110630

学生姓名：王佳佶

评阅教师：

报告成绩：

实验与创新实践教育中心制

2025 年 3 月

1、PositionSearchProblem：问题 1-4

1.1 代码实现

```python
def graphSearch(problem:SearchProblem, frontier_type, use_priority=False, heuristic=None):
    reached = set()
    solution=[]
    if use_priority:
        if heuristic is None:
            priority_function = lambda item: problem.getCostOfActions(item[1])
            frontier = frontier_type(priority_function)
        else:
            priority_function = lambda item: problem.getCostOfActions(item[1]) + heuristic(item[0], problem)
            frontier = frontier_type(priority_function)
    else:
        frontier = frontier_type()

    frontier.push((problem.getStartState(),solution))

    while not frontier.isEmpty():
        curState,curSolution = frontier.pop()
        if problem.isGoalState(curState):
            return curSolution
        if curState not in reached:
            reached.add(curState)
            for [nextState,nextDirection,_] in problem.getSuccessors(curState):
                frontier.push((nextState,curSolution+[nextDirection]))

    return []
```

图搜索框架

```python
def depthFirstSearch(problem: SearchProblem):
    return graphSearch(problem,util.Stack)

def breadthFirstSearch(problem: SearchProblem):
    return graphSearch(problem,util.Queue)

def uniformCostSearch(problem: SearchProblem):
    return graphSearch(problem,util.PriorityQueueWithFunction,use_priority=True)

def nullHeuristic(state, problem=None):
    return 0

def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
    return graphSearch(problem,util.PriorityQueueWithFunction,use_priority=True,heuristic=heuristic)
    #util.raiseNotDefined()
```

四种搜索算法对图搜索框架的调用

1.2 实验结果分析

以下列表格的形式，列出四种算法在 bigMaze 迷宫上的实验数据，包括扩展节点数、路径 cost、耗时，以及在 pacman 项目中的完备性、代价最优性，并就完备性和代价最优性给出总结说明。

	扩展节点数	路径 cost	耗时/s	完备性	代价最优性
dfs	390	210	0.00902	否	否
bfs	620	210	0.01485	是	是
ucs	620	210	0.07481	是	是
A*	549	219	0.06292	不一定	不一定

完备性：
DFS：如果搜索树存在无限深的分支，DFS 可能会陷入其中而无法返回，不完备
BFS：因为它是逐层扩展节点，只要解存在，它会在有限的步数内找到解，完备
UCS：只要每个边的代价都是非负的，UCS 就能保证找到解，完备
A*：完备性取决于启发函数的，本次实验采用曼哈顿距离可实现完备性

代价最优性：
DFS：找到的解不一定是最短路径，不具有最优性
BFS：逐层扩展的，最先找到的解一定是最短路径，具有最优性
UCS：总是扩展路径代价最小的节点，一定是代价最小的路径，具有最优性
A*：当启发式函数是可采纳的和一致的时，具有最优性

2、CornersProblem：问题 5-6

2.1 问题 5 的代码

```
def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
    "*** YOUR CODE HERE ***"
    return (self.startingPosition,tuple(self.corners))
    #util.raiseNotDefined()

def isGoalState(self, state: Any):
    """
    Returns whether this search state is a goal state of the problem.
    """
    "*** YOUR CODE HERE ***"
    return not state[1]
    #util.raiseNotDefined()
```

```
successors = []
for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
    # Add a successor state to the successor list if the action is legal
    # Here's a code snippet for figuring out whether a new position hits a wall:
    #   x,y = currentPosition
    #   dx, dy = Actions.directionToVector(action)
    #   nextx, nexty = int(x + dx), int(y + dy)
    #   hitsWall = self.walls[nextx][nexty]

    "*** YOUR CODE HERE ***"
    currentPosition,visited = state
    x,y=currentPosition
    dx,dy=Actions.directionToVector(action)
    nextx, nexty = int(x + dx), int(y + dy)
    if not self.walls[nextx][nexty]:
        nextState=(nextx,nexty)
        cornerState = list(state[1])
        if (nextx, nexty) in cornerState:
            cornerState.remove((nextx, nexty))
        cornerState = tuple(cornerState)

        successors.append(((nextState, cornerState), action, 1))
self._expanded += 1 # DO NOT CHANGE
return successors
```

初始状态时包含四个角落的坐标，在遍历的时候每遇到一个角落坐标，则该状态的角落坐标列表就去掉这一个角落，直到所有角落都被去掉之后则表示遍历完四个角落。

2.2 问题 6 的启发函数设计和代码实现

尝试两种不同的启发函数实现

（1）

启发函数 1：曼哈顿距离，具有可纳性和一致性

$$d_{12} = |x_1 - x_2| + |y_1 - y_2|$$

```
def manhattanDistance( xy1, xy2 ):
    "Returns the Manhattan distance between points xy1 and xy2"
    return abs( xy1[0] - xy2[0] ) + abs( xy1[1] - xy2[1] )
```

启发函数 2：欧几里得距离，具有可纳性和一致性

$$d_{12} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

```
def euclideanDistance( xy1, xy2 ):
    return ( (xy1[0] - xy2[0]) ** 2 + (xy1[1] - xy2[1]) ** 2 ) ** 0.5
```

（2）

在迷宫问题种，曼哈顿距离优于欧几里得距离，因为曼哈顿距离只算左右上下四个方向的直线距离，和迷宫的实际路径更接近，对于问题的启发信息更多，而欧几里得距离更适合不规定角度方向的问题。

2.3 实验结果分析

	扩展节点数	路径 cost	耗时/s
曼哈顿距离	692	106	0.03098
欧几里得距离	806	106	0.04449

由数据分析可知，曼哈顿距离作为启发函数时的 A*算法扩展节点数更小，耗时更少，所以曼哈顿距离作为启发函数在迷宫问题里更好。

3、FoodSearchProblem：问题 7-8

3.1 问题 7 的启发函数设计和代码

设 P 为吃豆人的当前位置，$F=\{f_1,f_2,f_3\ldots\}$是剩余食物的位置的集合，$d(p_1,p_2)$表示点p_1,p_2之间的迷宫距离，$f_{near}=arg\ min_{f\in F}\ d(P,f)$表示距离吃豆人最近的食物的位置，则启发式函数$h(P,F)$的公式如下：

$$h(P,F)=\begin{cases}0,F=\emptyset\\ d(P,f_{near})+max_{f^{'}\in F-\{f_{near}\}}d(f_{near},f^{'}),F\neq\emptyset\end{cases}$$

```
def heuristicInfoDistance(point1,point2):
    try:
        return problem.heuristicInfo[(point1,point2)]
    except:
        problem.heuristicInfo[(point1,point2)] = mazeDistance(point1,point2,problem.startingGameState)
        return problem.heuristicInfo[(point1,point2)]
position, foodGrid = state
foodLeft = foodGrid.asList()
foodDistance = []
foodFromFood = []
for food in foodLeft:
    foodDistance.append((heuristicInfoDistance(position,food),food))

if not foodDistance:
    return 0

minFoodDistance = min(foodDistance)
minDistance = minFoodDistance[0]
nextFood = minFoodDistance[1]

for food in foodLeft:
        foodFromFood.append(heuristicInfoDistance(food,nextFood))
return minDistance+max(foodFromFood)
```

启发函数的值为吃豆人先吃到最近的食物的迷宫距离再加上这个食物到距离这个食物最远的食物的迷宫距离，显然小于等于吃完所有食物所需要的代价之和，而且对于每个不同的位置，启发函数的值都会发生变化，不恒等于 0，故具有一致性和可纳性。

事实证明，该启发函数能在 autograder 中获得附加分。

```
***      expanded nodes: 1818
***      thresholds: [15000, 12000, 9000, 7000]

### Question q7: 5/4 ###


Finished at 10:49:36

Provisional grades
==================
Question q4: 3/3
Question q7: 5/4
------------------
Total: 8/7
```

3.2 问题 8 的代码实现

```
def findPathToClosestDot(self, gameState: pacman.GameState):
    """
    Returns a path (a list of actions) to the closest dot, starting from
    gameState.
    """
    # Here are some useful elements of the startState
    startPosition = gameState.getPacmanPosition()
    food = gameState.getFood()
    walls = gameState.getWalls()
    problem = AnyFoodSearchProblem(gameState)
    "*** YOUR CODE HERE ***"
    return search.astar(problem)
    #util.raiseNotDefined()
```

```
def isGoalState(self, state: Tuple[int, int]):
    """
    The state is Pacman's position. Fill this in with a goal test that will
    complete the problem definition.
    """
    x,y = state

    "*** YOUR CODE HERE ***"
    return self.food[x][y]
    #util.raiseNotDefined()
```

贪心算法只需要找到距离最近的一个食物就行，判断目标状态也就是看第一个找到的位置上是否具有食物，如果有的话则可保证是最近的一个食物。本体选用的搜索函数除了 DFS 之外均可，因为无法保证代价最优性，而其他三种算法中的 A*算法的性能最佳。

4、总结

4.1 碰到的问题

在问题 6 中，需要遍历剩余的角落来计算启发函数的值，剩余角落在状态中是以元组的形式存储的，如果直接用来算启发函数的话没办法使用 remove 办法移除掉已经计算过的角落

```
  File "C:\Users\jimmywang\Desktop\AI-2025\lab1\search\search.py", line 83, in <lambda>
    priority_function = lambda item: problem.getCostOfActions(item[1]) + heuristic(item[0], problem)
  File "C:\Users\jimmywang\Desktop\AI-2025\lab1\search\searchAgents.py", line 391, in cornersHeuristic
    cornersLeft.remove(nearestCorner)
AttributeError: 'tuple' object has no attribute 'remove'
```

所以需要在计算启发函数的时候先将 cornerLeft 转换为列表

```
currentPosition,cornersLeft = state
#计算前先转换为列表
cornersLeft=list(cornersLeft)
```

4.2 总结、建议

通过本次实验，直观的认识到启发式函数的设计直接影响 A* 算法的效率和准确性，需平衡估算值的合理性与计算成本，也了解到了 DFS 因缺乏最优性保证而无法用于要求代价最优的场景，收获丰富。

一点小的建议是该实验的数据结构较复杂，一些结构体的内部参数需要较长时间的阅读才能记住，希望能有参数说明文档和一些入门的例子，能够更快上手。