实验报告

（2023~2024 学年第二学期）

课程名称 人工智能课程设计

实验名称 课程设计作业 Project1

姓名 范潇 学号 2254298

1. 项目概述

1.1. 主要内容与目标

本次项目的主要内容为搜索算法。具体目标是实现算法使得吃豆人能够在迷宫中达到特定的位置，同时高效地收集食物。因此，本项目中的状态只需包含当前位置，如果要求需要收集多处食物，则还需要包含存储相应记录的数组。

本项目的八个问题可以分为三部分。

第一部分的任务是搜索问题是到达迷宫中指定的位置，要求分别使用 DFS,BFS,UCS 以及 A^* 算法实现。在该部分中，用于测试的迷宫有 tinyMaze,mediumMaze,bigMaze,openMaze，前三个迷宫中只有一处食物，即目标状态。openMaze 的特点在于迷宫的墙壁较为稀疏。

第二部分的任务是对于指定搜索任务进行形式化，并构造相应的启发式函数。在该部分中，搜索任务的目标是让吃豆人到达迷宫的四个角落。

第三部分的任务围绕“收集迷宫中的所有食物”这一搜索问题展开，要求设计启发式函数以及不追求最优解的算法。

1.2. 已有代码

下面对已有代码中与本项目的具体实现相关的代码进行分析。

1.2.1. util.py

该文件中提供了多种可能需要用到的数据结构，并且如果需要使用，不能用 Python 自带库代替，因为可能会影响自动评分程序。

1. Stack：有 push,pop,isEmpty 方法，分别用于进行压栈、出栈以及判断栈是否为空的操作。
2. Queue：有 push,pop,isEmpty 方法，分别用于入队、出队以及判断队列是否为空的操作。
3. PriorityQueue：有 push,pop,isEmpty,update 方法，分别用于入队、出队、判断队列是否为空以及更新元素有限度的操作。优先弹出优先级低的元素。
4. PriorityQueueWithFunction：是 PriorityQueue 的子类，支持指定一个函数 priorityFunction 用于从存储元素中提取优先级。

这些数据结构的底层都是用列表实现的。

同时还提供了用于计算曼哈顿距离的函数 manhattanDistance。

1.2.2. search.py

该文件用于存储各种搜索算法的实现，同时包含了抽象类 SearchProblem。

SearchProblem 类有以下方法：

1. getStartState(self)：返回初始状态
2. isGoalState(self,state)：对所给状态进行目标测试
3. getSuccessors(self,state)：返回三元组 (successor,action,stepCost)，其中 successor 为所给状态的后继，action 为对应的状态，stepCost 为 action 对应的代价
4. getCostOfActions(self,actions)：输入为一系列合法动作，返回对应的代价总和。

同时，该文件还给出了一个样例：tinyMazeSearch(problem)。从中可以了解到需要自行实现的算法返回的解序列的形式是一个列表，元素为 game 库中导入的 Directions 类。

1.2.3. searchAgents.py

该文件中存储的是关于智能体以及搜索问题相关的代码，并不包含搜索算法的实现。

PositionSearchProblem 类是 SearchProblem 类的子类，用于存储状态空间中只包含位置的搜索问题。具体细节有：

1. 状态用二元元组实现
2. 当行动非法，即产生“穿墙”行为时，代价返回 999999

从该类的代码中可以了解到初始化一个问题的基本步骤为：

1. 初始化墙
2. 获取吃豆人初始位置
3. 设置起始状态
4. 设置目标测试
5. 设置损失函数

同时，该文件还提供了函数 mazeDistance。该函数通过广度优先搜索来获取迷宫内两点之间的最短路径长度。

2. 第一部分

2.1. 问题概述

该部分包含第 1，2，3，4 小题。

该部分要求使用各类搜索算法——DFS,BFS,UCS,A^*——来帮助吃豆人规划到达一处固定食物的路径。也就是说，要利用搜素算法寻找一条从初始状态到目标状态的解路径，要确保每一步都是合法的，即遵循迷宫的约束。同时题目特别要求使用图搜索，以确保算法的完整性。

由于只有一处食物，该部分中的状态只需要包含吃豆人当前的位置即可，目标测试即判定当前位置是否与食物所在位置一致。

由于该部分中所需要的各类搜索算法，都遵循搜索算法的通用框架，唯一的区别在于从边缘队列中选取下一个展开结点的策略，因此我先实现通用图搜索算法 generalGraphSearch，然后实现各类从边缘队列中挑选结点的策略，从而达到实现各类搜索算法的目标。

2.2. 算法设计

为了确保通用性，在 generalGraphSearch 中，我采用优先队列来实现边缘队列，并通过传入不同的优先性计算函数 getPrioriy 来实现不同的搜索算法。

由于题目要求的是图搜索算法，不同于树搜索和最佳优先算法，在 generalGraphSearch 中，如果一个结点已经被展开过，那么后续都不会再次被展开；同时，展开获得的后继结点先不经检查便加入边缘序列中，直到依据特定策略从边缘序列中弹出后才检查是否已经被展开过，若没有，则展开。

下面给出的是 generalGraphSearch 的伪代码，其中 getPriority 是优先性计算函数。

Procedure generalGraphSearch(problem,getPriority)

Output: Actions

if *problem.isGoalState(problem.getStartState())* **then return emptyActions** // 检查起始结点

初始化起始结点 start

初始化边缘队列 frontier// 采用优先队列

初始化 reached 表// 采用字典

reached.add(start)// 将 start 加入 reached 中

foreach *child* **in** *problem.getSuccessors(start)* **do** frontier.push(child,`getPriority` (child))
// 添加 start 的所有后继

while not *frontier.isEmpty()* **do**

    next = frontier.pop()// 获取下一个展开结点

    if *next* **in** *reached* **then continue** // 检查是否已访问过

    reached.add(next)// 将 next 加入 reached 中

    if *problem.isGoalState(next)* **then return** `getSolution` *(next,reached)* // 通过目标检测

    foreach *child* **in** *problem.getSuccessors(next)* **do** frontier.push(child,`getPriority` (child))
    // 添加 next 的所有后继

end while

为了获取解序列，在通过目标检测后，需要从目标状态不断回溯至初始状态，同时保存其中所需要的动作。

Procedure getSolution(goal,reached)

Output: Actions

```
初始化 Actions// 采用列表
cur = goal// 初始化工作指针
while reached[cur].parent do // 起始节点无父节点
    solution.append(reached[cur].action)// 添加到当前结点对应的动作
    cur = reached[cur].parent// 移动至父节点
end while
return Actions.reverse()// 取反
```

在 generalGraphSearch 的基础之上，只需定义特定的优先性计算函数，便能得到不同的图搜索算法。

在 DFS 和 BFS 算法中，一个结点的优先级和只它的深度有关。因为已有代码中提供的优先队列优先弹出优先级小的元素，所以在 DFS 算法中当前结点的优先级应该比父节点要小；在 BFS 算法中当前结点的优先级应该比父节点要大。

Function dfsPriority(parent)

```
if not parent then return 0 // 无父节点，即起始结点
return parent.priority-1// 优先级越小越优先
```

Function bfsPriority(parent)

```
if not parent then return 0 // 无父节点，即起始结点
return parent.priority+1// 优先级越小越优先
```

在 UCS 算法中，一个结点的优先级和到该结点的路径的总代价相关。

Function ucsPriority(parent,pathCost)

```
if not parent then return 0 // 无父节点，即起始结点
return parent.priority+pathCost// 到父节点的总代价加上从父节点到该结点的代价
```

在 A^* 算法中，一个结点的优先级不仅和到该结点的路径的总代价相关，也和启发式函数返回的值相关。

Function AStarPriority(parent,pathCost,state)

```
if not parent then return heuristic (state,problem) // heuristic 为给定的启发式函数
return parent.priority+pathCost+heuristic (state,problem)
```

Algorithm 2.2.1: depthFirstSearch(problem)

```
return generalGraphSearch(problem,dfsPriority)
```

Algorithm 2.2.2: breadthFirstSearch(problem)

```
return generalGraphSearch(problem,bfsPriority)
```

Algorithm 2.2.3: uniformCostSearch(problem)

```
return generalGraphSearch(problem,ucsPriority)
```

Algorithm 2.2.4: aStarSearch(problem)

1 **return** *generalGraphSearch(problem,AStarPriority)*

2.3. 算法实现

该部分中所需要的数据结构主要用于表示未展开的结点和已展开的结点。

对于已展开的结点，由于需要用于回溯形成解序列，所以需要存储父节点和行动，所以采用键值对形式，键为状态，值为对应的父节点和行动。由于已展开的结点是由未展开的结点变化而来的，所以未展开的结点中也需要存储父节点和行动，除此之外，为了用于展开和计算子节点的相应性质，还需要包含状态域、优先性域以及路径代价域。

```
from typing import Callable, List, Dict, Tuple
from collections import namedtuple
from game import Stack, PriorityQueue
from game import Directions
#未展开结点
#用于展开结点，以及计算子节点的优先性和路径代价所以需要存储状态，优先性和路径代价
Node = namedtuple('Node',('state','pathCost','priority','property'))
Property = namedtuple('Property',('parent','action'))
#用于记录已访问结点，同时还要用于回溯形成解序列，所以需要存储父节点和行动，所以采用键值对形式，
Reached = Dict[Node.state,Property]
#行为
Action = (Directions.NORTH or Directions.SOUTH or
          Directions.EAST or Directions.WEST or
          Directions.LEFT or Directions.RIGHT or
          Directions.STOP or Directions.REVERSE)
Actions = List[Action]
```

由于字典要求键的类型可以进行散列，所以键中不能包含列表类型。因此，在将结点存放进 Reached 表中时，需要把用于作为键的状态域的类型统一改为元组等可以进行散列的类型。为此，我编写了辅助函数 list2tuple 来将列表以及嵌套列表转换为元组和嵌套元组。这一转换在后续完成状态中包含 foodGrid 的问题时是必要的，因为已有代码中使用嵌套列表来实现 foodGrid。这一转换使得 generalGraphSearch 变得更加通用。

```
def list2tuple(state):
    if not isinstance(state, tuple):#如果是int等类型就直接返回
        return state
    return tuple(list2tuple(item) if isinstance(item, list) else item for
       item in state)#否则就遍历各个元素并判断
```

```
def generalGraphSearch(problem: SearchProblem,
                       getPriority: Callable[[Node or None, float, Node.
                          state], float]) -> Actions:
```

```
    if problem.isGoalState(problem.getStartState()):
        return []
    start = Node(problem.getStartState(), 0, getPriority(None, 0, problem.
       getStartState()), Property(None, None))
    frontier = PriorityQueue()
    reached: Reached = {list2tuple(problem.getStartState()): start.property}
         # 初始化reached并添加初始结点
    for child in problem.getSuccessors(problem.getStartState()):
        state = list2tuple(child[0])
        frontier.push(Node(state, child[2], getPriority(start, child[2],
           state),Property(list2tuple(problem.getStartState()), child[1])),
           getPriority(start, child[2], state))  # 添加各子节点
    while not frontier.isEmpty():
        nextNode: Node = frontier.pop()  # 获取下一个待展开的结点
        if nextNode.state in reached.keys():  # 判断是否已展开
            continue
        reached[nextNode.state] = nextNode.property  # 标记为已展开
        if problem.isGoalState(nextNode.state):  # 目标检测
            return get_solution(nextNode.state, reached)
        for child in problem.getSuccessors(nextNode.state):
            state = list2tuple(child[0])
            new = Node(state, nextNode.pathCost + child[2], getPriority(
               nextNode, child[2], state),
                       Property(nextNode.state, child[1]))  # 初始化子节点
            frontier.push(new, new.priority)  # 添加各子节点
```

```
def get_solution(goal: Node.state, reached: Reached) -> Actions:
    solution: Actions = []
    cur = goal
    while reached[cur].parent:
        solution.append(reached[cur].action)
        cur = reached[cur].parent
    return solution[::-1]  # 取反
```

由于测试用例要求实现的 DFS 搜索算法在处理边缘队列中的两个优先级相同的结点时优先弹出后加入的结点，而 PriorityQueue 则时优先弹出先加入的结点，为了避免改变已有代码，我的处理方式是在每个结点的原有优先级基础上减少一个逐渐增加的小量，以避免出现优先级相同的情况。

```
def cnt():
    i = 0
    while True:
        yield i
        i = i + 1

```

```
counter = cnt()

def dfsPriority(parent: Node or None, pathCost: float, state: Node.state) ->
    float:
    if not parent:
        return 0
    return parent.priority - 1 - next(counter) * epsilon  # 额外减少一个小量
```

```
def bfsPriority(parent: Node or None, pathCost: float, state: Node.state) ->
    float:
    if not parent:
        return 0
    return parent.priority + 1
```

```
def ucsPriority(parent: Node or None, pathCost: float, state: Node.state) ->
    float:
    if not parent:
        return 0
    return parent.priority + pathCost
```

我所设计的优先级计算函数的参数为父节点，路径代价和当前状态，以满足各种搜索算法的需求。只需把上述三个优先级计算函数作为参数传入 generalGraphSearch 便可以得到所要的 DFS，BFS，UCS 算法。

```
def depthFirstSearch(problem: SearchProblem):
    return generalGraphSearch(problem, dfsPriority)

def breadthFirstSearch(problem: SearchProblem):
    return generalGraphSearch(problem, bfsPriority)

def uniformCostSearch(problem: SearchProblem):
    return generalGraphSearch(problem, ucsPriority)
```

在实现 A^* 算法时，需要使用到 Python 的高阶函数功能，因为启发式函数不能够作为参数传入我所设计的优先级计算函数中。

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
    def AStarPriority(parent: Node or None, pathCost: float, state: Node.
        state) -> float:
        if not parent:
            return heuristic(problem.getStartState(), problem)
        return parent.pathCost + pathCost + heuristic(state, problem)

    return generalGraphSearch(problem, AStarPriority)
```

2.4. 实验结果

2.4.1. Question1&2

成功通过 Question1 和 Question2 的全部测试，实验结果截图分别见图 2.1和图 2.2。Question1 的测试针对实现的 DFS 算法，Question2 针对的是 BFS 算法。

```
Question q1
===========
*** PASS: test_cases\q1\graph_backtrack.test
***     solution:       ['1:A->C', '0:C->G']
***     expanded_states:    ['A', 'D', 'C']
*** PASS: test_cases\q1\graph_bfs_vs_dfs.test
***     solution:       ['2:A->D', '0:D->G']
***     expanded_states:    ['A', 'D']
*** PASS: test_cases\q1\graph_infinite.test
***     solution:       ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:    ['A', 'B', 'C']
*** PASS: test_cases\q1\graph_manypaths.test
***     solution:       ['2:A->B2', '0:B2->C', '0:C->D', '2:D->E2', '0:E2->F', '0:F->G']
***     expanded_states:    ['A', 'B2', 'C', 'D', 'E2', 'F']
*** PASS: test_cases\q1\pacman_1.test
***     pacman layout:      mediumMaze
***     solution length: 130
***     nodes expanded:     146

### Question q1: 3/3 ###
```

图 2.1: Question1 实验结果

```
Question q2
===========
*** PASS: test_cases\q2\graph_backtrack.test
***     solution:       ['1:A->C', '0:C->G']
***     expanded_states:    ['A', 'B', 'C', 'D']
*** PASS: test_cases\q2\graph_bfs_vs_dfs.test
***     solution:       ['1:A->G']
***     expanded_states:    ['A', 'B']
*** PASS: test_cases\q2\graph_infinite.test
***     solution:       ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:    ['A', 'B', 'C']
*** PASS: test_cases\q2\graph_manypaths.test
***     solution:       ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:    ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases\q2\pacman_1.test
***     pacman layout:      mediumMaze
***     solution length: 68
***     nodes expanded:     269

### Question q2: 3/3 ###
```

图 2.2: Question2 实验结果

1. **graph_backtrack:** 该测试用例旨在测试算法能否在简单情形下返回正确解路径。该测试样例中，cost 并不都等于 1，因此，实现 DFS 和 BFS 算法时不能够预设路径代价均为 1，结点的优先级（深度）应该依据父节点的优先级（深度）进行计算。

2. **graph_bfs_vs_dfs:** 该测试用例旨在测试算法能否在简单情形下返回正确解路径，同时能够体现出 DFS 无法保证返回最优解的特性

3. **graph_infinite:** 该测试用例旨在测试算法能否在状态图存在回路的情况下返回正确解路径，generalGraphSearch 中的 reached 表确保了算法不会陷入循环

4. **graph_manypaths:** 在该测试用例中，有多条解路径，因此同一结点可能会被放入边缘队列多次。cnt 函数的使用确保了 DFS 算法解决优先级相同时的方法与答案一致，即后进边缘队列的结点先出队。

5. **pacman_1:** 该测试样例为吃豆人场景，迷宫大小为中等。可以看到，DFS 展开的结点较少，但是返回的解路径较长；而 BFS 展开的结点较多，但是返回的解路径较短，实际上是最短的解路径。

上述的测试样例中均有且仅有一个目标位置。

2.4.2. Question3

成功通过本问题的全部测试，实验结果截图见图 2.5。

1. **graph_backtrack,graph_bfs_vs_dfs,graph_infinite:**UCS 在这三个测试样例中返回的解路径和展开的结点与 BFS 的均相同，但这只是人为设置而导致的巧合，在这些测试样例中路径代价并不全为 1。

2. **ucs_1_problemC,ucs_2_problemE,ucs_3_problemW:** 这三个测试样例共用同一个迷宫场景，吃豆人初始位置在右上角，目标位置有且仅有一个，在左下角。三个测试样例中的路径代价的特点分别为：始终为常数、沿东方向指数递减、沿西方向指数递减。由于目标位置在初始位置的西南方向，沿西方向递减的路径代价会诱导吃豆人朝目标探索，而沿东方向递减的路径代价则相反，因此前者对应的展开结点数量小于后者。但是由于迷宫的构造，前者所得到的解路径的长度却远大于后者。

3. **ucs_4_testSearch:** 有两个目标位置，一个就紧邻着起始位置，但是对应的路径代价很高，另一个目标位置则相反。

4. **ucs_5_goalAtDequeue:** 有且仅有一个目标位置，但有两条解路径，且会出现有两个目标结点同时出现在边缘队列的情况，先加入的路径代价更高。通过上述两个测试的关键在于目标检测只在一个结点离开边缘队列是进行。

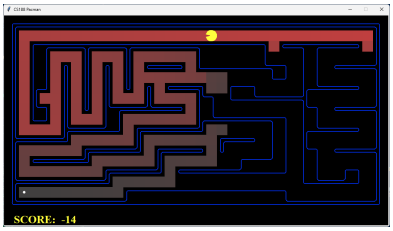

图 2.3: ucs_3_problemW

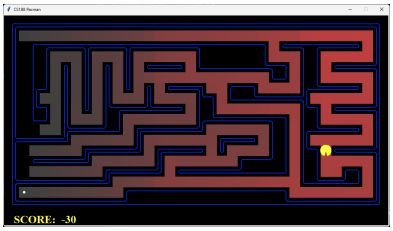

图 2.4: ucs_3_problemE

```
Question q3
===========
*** PASS: test_cases\q3\graph_backtrack.test
***     solution:       ['1:A->C', '0:C->G']
***     expanded_states:    ['A', 'B', 'C', 'D']
*** PASS: test_cases\q3\graph_bfs_vs_dfs.test
***     solution:       ['1:A->G']
***     expanded_states:    ['A', 'B']
*** PASS: test_cases\q3\graph_infinite.test
***     solution:       ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:    ['A', 'B', 'C']
*** PASS: test_cases\q3\graph_manypaths.test
***     solution:       ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:    ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases\q3\ucs_0_graph.test
***     solution:       ['Right', 'Down', 'Down']
***     expanded_states:    ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q3\ucs_1_problemC.test
***     pacman layout:      mediumMaze
***     solution length: 68
***     nodes expanded:     269
*** PASS: test_cases\q3\ucs_2_problemE.test
***     pacman layout:      mediumMaze
***     solution length: 74
***     nodes expanded:     260
*** PASS: test_cases\q3\ucs_3_problemW.test
***     pacman layout:      mediumMaze
***     solution length: 152
***     nodes expanded:     173
*** PASS: test_cases\q3\ucs_4_testSearch.test
***     pacman layout:      testSearch
***     solution length: 7
***     nodes expanded:     14
*** PASS: test_cases\q3\ucs_5_goalAtDequeue.test
***     solution:       ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:    ['A', 'B', 'C']

### Question q3: 3/3 ###
```

图 2.5: Question3 实验结果

2.4.3. Question4

成功通过本问题的全部测试，实验结果截图见图 2.6。

```
Question q4
===========
*** PASS: test_cases\q4\astar_0.test
***     solution:       ['Right', 'Down', 'Down']
***     expanded_states:    ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q4\astar_1_graph_heuristic.test
***     solution:       ['0', '0', '2']
***     expanded_states:    ['S', 'A', 'D', 'C']
*** PASS: test_cases\q4\astar_2_manhattan.test
***     pacman layout:      mediumMaze
***     solution length: 68
***     nodes expanded:     221
*** PASS: test_cases\q4\astar_3_goalAtDequeue.test
***     solution:       ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:    ['A', 'B', 'C']
*** PASS: test_cases\q4\graph_backtrack.test
***     solution:       ['1:A->C', '0:C->G']
***     expanded_states:    ['A', 'B', 'C', 'D']
*** PASS: test_cases\q4\graph_manypaths.test
***     solution:       ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:    ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

### Question q4: 3/3 ###
```

图 2.6: Question4 实验结果

1. **astar_0:** 该测试样例中包含两个目标结点，启发式函数恒为 0，所设置的路径代价会使得算法无法给出解路径
2. **astar_1_graph_heuristic:** 有且仅有一个目标结点，要求算法进行图搜索
3. **astar_2_manhattan:** 在迷宫环境中以曼哈顿距离为启发式函数
4. **astar_3_goalAtDequeue:** 有且仅有一个目标位置，但有两条解路径，且会出现有两个目标结点同时出现在边缘队列的情况，先加入的路径代价更高。通过上述两个测试的关键在于目标检测只在一个结点离开边缘队列是进行。
5. **graph_backtrack:** 同之前的同名测试
6. **graph_manypaths:** 同之前的同名测试

3. 第二部分

3.1. 问题概述

该部分包含第 5、6 小题。

该部分所要解决的搜索问题是让吃豆人遍历迷宫的四个角落。题目要求我们先对该问题进行形式化，然后给出相应的具有一致性的启发式函数。

这一搜索问题的状态除了吃豆人当前位置外，还需要以某种方式记录下目前为止遍历了哪些角落，以便进行目标检验。因此和第一部分中的问题相比，该搜索问题的转移函数等部分还需额外处理状态中新增的记录。

由于该问题中需要遍历的位置是固定的四个角落，因此可以在忽视墙壁的条件下，求出的遍历四个角所需要的最小的曼哈顿距离便可以作为启发式函数。

3.2. 算法设计

由于遍历四个角时，必会先达到其中一个角。而当忽视墙壁时，从任意一个角出发，遍历其他三个角所需要的最短路径显然是一样的，均为宽和高之和，再加上两者中较小的一个。因此，从迷宫中的任意一点出发，不断朝着最近的一个未遍历过的角落前进便可以得到最短的总距离。而由于这是在放松限制条件下得到的最优解，所以它的长度可以作为实际问题的启发式函数。

Procedure greedy(position,goals)

```
Output: cost
cost = 0
while goals do // 还有未遍历的位置
    nearestDest = inf
    nearest = None
    foreach goal in goals do
        if manhattanDistance(position,goal)<nearestDest then
            nearestDest = manhattanDistance(position,goal)
            nearest = goal
        end if
    end foreach
    cost += nearestDest
    goals.remove(nearest)
    position = nearest// 更新为最近目标
end while
return cost
```

要证明一个启发式函数是一致的，只需证明对于任意一个动作，前后状态对应的启发式函数值的减少量不大于该动作的代价即可。而当忽视墙壁时，由于该启发式函数值可以分为两部分：到达最近的一个角

落所需的曼哈顿距离，以及从这个角落出发遍历其他角落所需的最小曼哈顿距离之和，而后者并不会因为吃豆人的动作发生改变，同时前者的减少量不大于动作的代价。所以这一启发式函数是一致的。

3.3. 算法实现

由于我的 generalGraphSearch 中将状态作为 reached 表的键，所以要求状态所使用的类型是可以散列的。同时，我所设计的启发式函数在计算时需要不断更新状态的遍历记录。因此，我采用整数来存储遍历记录。取整数的低 4 位，分别代表 4 个角落，为 1 则表示还未遍历，否则表示已遍历。

在 CornersProblem 类中，___init___ 和 getCostOfAction 方法无需进行修改。下面给出修改后的其他方法

```
def getStartState(self):
    # 用01串来记录是否已经达到各个位置，这样方便改动，且是hashable
    return self.startingPosition, (1 << len(self.corners)) - 1
```

```
def isGoalState(self, state: Any):
    return not state[1]  # 存储历史的整数为0，即各位为0
```

```
Successors = namedtuple('Successors', ('cornerProblemState', 'action', '
   pathCost'))
def getSuccessors(self, state: Any):
    successors = []
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
       Directions.WEST]:
        x, y = state[0]
        goals = state[1]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        if not self.walls[nextx][nexty]:
            nextpos = (nextx, nexty)
            remains = goals
            if nextpos in self.corners:
                index = self.corners.index(nextpos)
                remains = remains ^ (1 << index)  # 取反
            successors.append(Successors((nextpos, remains), action, 1))
    self._expanded += 1  # DO NOT CHANGE
    return successors
```

下面给出的是启发式函数的实现。

```
def greedy(position: tuple, goals: List[tuple]) -> float:
    cost = 0
    while goals:
        nearestDest = inf
        nearest = None
```

```
        for goal in goals:
            if manhattanDistance(position, goal) < nearestDest:
                nearestDest = manhattanDistance(position, goal)
                nearest = goal
        cost += nearestDest
        goals.remove(nearest)
        position = nearest
    return cost
```

```
def cornersHeuristic(state: Any, problem: CornersProblem):
    corners = problem.corners  # These are the corner coordinates
    walls = problem.walls  # These are the walls of the maze, as a Grid (
        game.py)
    if not state[1]:
        return 0
    remains = [corner for index, corner in enumerate(corners) if 1 & (state
        [1] >> index)]  # 获取还未遍历到的角落
    return greedy(state[0], remains)
```

3.4. 实验结果

3.4.1. Question5

成功通过本问题的全部测试，实验结果截图见图 3.1。测试用例是一个小型迷宫。

```
Question q5
===========
*** PASS: test_cases\q5\corner_tiny_corner.test
***     pacman layout:      tinyCorner
***     solution length:        28

### Question q5: 3/3 ###
```

图 3.1: Question5 实验结果

3.4.2. Question6

成功通过本问题的全部测试，实验结果截图见图 3.2。共有四个测试样例，用于测试所设计的启发式函数是否是一致的。前三个样例为小型迷宫，最后一个样例为中型迷宫。

```
Question q6
===========
*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
path: ['North', 'East', 'East', 'East', 'East', 'North', 'North', 'West', 'West', 'West', 'West', 'North', 'North', 'North', 'North', 'North', 'North',
 'North', 'North', 'West', 'West', 'West', 'West', 'South', 'South', 'East', 'East', 'East', 'East', 'South', 'South', 'South', 'South', 'South', 'South',
 'West', 'West', 'South', 'South', 'South', 'West', 'West', 'East', 'East', 'North', 'North', 'North', 'East', 'East', 'East', 'East', 'East', 'East',
 'East', 'East', 'South', 'South', 'East', 'East', 'East', 'East', 'East', 'North', 'North', 'East', 'East', 'North', 'North', 'East', 'East', 'North',
 'North', 'East', 'East', 'East', 'East', 'South', 'South', 'South', 'South', 'East', 'East', 'North', 'North', 'East', 'East', 'South', 'South', 'South',
 'South', 'South', 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'West', 'West', 'North', 'North', 'East', 'East', 'North', 'North']
path length: 106
*** PASS: Heuristic resulted in expansion of 710 nodes
```

图 3.2: Question6 实验结果

4. 第三部分

4.1. 问题概述

该部分包含第 7、8 小题。

该部分所要解决的问题是帮助吃豆人收集迷宫中所有的食物，食物个数和位置不定。题目要求为该问题设计一个一致的启发式函数。同时，题目要求实现 findPathToClosetDot 函数并补充完整 AnyFoodSearchProblem 类的目标检测函数，帮助吃豆人找到前往最近食物的路径，以找到上述问题的一个近似最优解。

4.2. 算法设计

一开始我尝试使用最小生成树的总代价作为启发式函数，但是经过测试后发现这并不一致。后来我尝试使用当前位置到各个目标位置的曼哈顿距离的最大值作为启发式函数，但是需要展开的结点过多，只能获得部分分数。受此启发，我在此基础上进一步优化，将启发式函数调整为各个目标位置加上当前位置所组成的集合中，任意两个位置之间的曼哈顿距离的最大值，这一启发式函数能够得到附加分之外的所有分数。

由于吃豆人移动一格时，上述集合中的任意两个元素之间的曼哈顿距离最多减少一个单位，因此最大值最多减少一个单位，即由此定义的启发式函数最多减少一个单位，即该启发式函是一致的。

对于 AnyFoodSearchProblem 类，它的目标是收集到任意一个食物。因此它的目标检测函数只需判断当前位置是否属于由目标位置组成的集合即可。

在已给的代码中，findPathToCloseDot 函数中创建了一个 AnyFoodSearchProblem 的实例，这提示我们利用该问题来辅助完成该函数的实现。这里所要求的前往最近食物的路径实际上就是 AnyFoodSearchProblem 的一个深度最浅的解。因此只需要使用 BFS 求解该问题即可。

4.3. 算法实现

```
1 def foodHeuristic(state: Tuple[Tuple, List[List]], problem:
    FoodSearchProblem):
2     position, foodGrid = state
3     nodes = foodGrid.asList()
4     nodes.append(position)
5     distance = [manhattanDistance(nodes[i], nodes[j]) for i in range(len(
          nodes) - 1) for j in range(i + 1, len(nodes))]
6     return max(distance) if distance else 0  # 确保目标结点的启发式函数值为0
```

下面给出的是 AnyFoodSearchProblem 类的目标检验函数的实现：

```
1 def isGoalState(self, state: Tuple[int, int]):
2     x, y = state
3     return self.food[x][y]
```

下面给出的是 findPathToClosestDot 的实现：

```
def findPathToClosestDot(self, gameState: pacman.GameState):
    startPosition = gameState.getPacmanPosition()
    food = gameState.getFood()
    walls = gameState.getWalls()
    problem = AnyFoodSearchProblem(gameState)
    return breadthFirstSearch(problem)
```

4.4. 实验结果

4.4.1. Question7

成功通过本问题的除了附加测试外的全部测试，实验结果截图见图 4.1。该问题的测试样例，除了最后一个，均为规模较小的迷宫，主要目的是通过各类情况完整地测试设计的启发式函数是否一致。本题附加测试所用的迷宫虽然规模不大，但是它的食物摆放位置使得我所设计的启发式函数失去了作用——无法正确反映出迷宫中间的墙壁，从而导致 A^* 算法退化，展开的结点较多。

```
Question q7
===========
*** PASS: test_cases\q7\food_heuristic_1.test
*** PASS: test_cases\q7\food_heuristic_10.test
*** PASS: test_cases\q7\food_heuristic_11.test
*** PASS: test_cases\q7\food_heuristic_12.test
*** PASS: test_cases\q7\food_heuristic_13.test
*** PASS: test_cases\q7\food_heuristic_14.test
*** PASS: test_cases\q7\food_heuristic_15.test
*** PASS: test_cases\q7\food_heuristic_16.test
*** PASS: test_cases\q7\food_heuristic_17.test
*** PASS: test_cases\q7\food_heuristic_2.test
*** PASS: test_cases\q7\food_heuristic_3.test
*** PASS: test_cases\q7\food_heuristic_4.test
*** PASS: test_cases\q7\food_heuristic_5.test
*** PASS: test_cases\q7\food_heuristic_6.test
*** PASS: test_cases\q7\food_heuristic_7.test
*** PASS: test_cases\q7\food_heuristic_8.test
*** PASS: test_cases\q7\food_heuristic_9.test
*** FAIL: test_cases\q7\food_heuristic_grade_tricky.test
***     expanded nodes: 8763
***     thresholds: [15000, 12000, 9000, 7000]

### Question q7: 4/4 ###
```

图 4.1: Question7 实验结果

4.4.2. Question8

成功通过本问题的全部测试，实验结果截图见图 4.3。该问题的测试样例均为规模较小的迷宫，用于测试 findPathToClosestDot 返回的最小路径值是否正确。

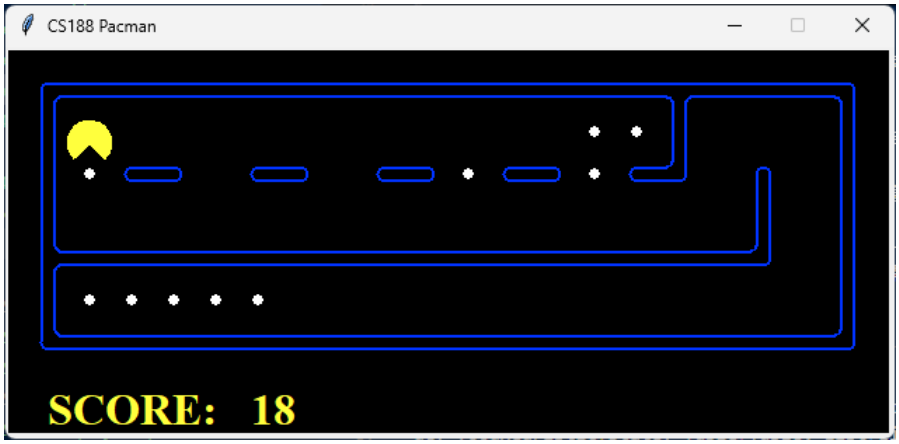

图 4.2: Question7 附加测试

```
Question q8
===========
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_1.test
***     pacman layout:      Test 1
***     solution length:        1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_10.test
***     pacman layout:      Test 10
***     solution length:        1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_11.test
***     pacman layout:      Test 11
***     solution length:        2
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_12.test
***     pacman layout:      Test 12
***     solution length:        3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_13.test
***     pacman layout:      Test 13
***     solution length:        1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_2.test
***     pacman layout:      Test 2
***     solution length:        1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_3.test
***     pacman layout:      Test 3
***     solution length:        1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_4.test
***     pacman layout:      Test 4
***     solution length:        3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_5.test
***     pacman layout:      Test 5
***     solution length:        1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_6.test
***     pacman layout:      Test 6
***     solution length:        2
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_7.test
***     pacman layout:      Test 7
***     solution length:        1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_8.test
***     pacman layout:      Test 8
***     solution length:        1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_9.test
***     pacman layout:      Test 9
***     solution length:        1

### Question q8: 3/3 ###
```

图 4.3: Question8 实验结果

5. 总结与分析

在完成第一部分时，我选择实现了通用图搜索算法 generalGraphSearch，这使得我在实现具体的搜索算法时，只需在额外实现对应的优先级函数即可。这一便利的代价便是在实现通用算法时，需要注意通用性，以兼容不同的搜索问题以及搜索算法。在这一过程中，我所遇到的主要问题便是由于我用字典来存放 reached 表而导致的，使得我必须将状态修改为可以进行散列的数据结构。将来可以将其改进为用列表存储 reached 表，并且是将整个结点直接放入，简化了所需要的数据结构。

在设计第二、第三部分的启发式函数的过程中，我体会到在忽略一定约束后所得到的最优解或者是完整约束下的部分解均有可能成为一个比较好的启发式函数。例如在 CornersProblem 中，我所设计的启发式函数便是在忽略了墙壁的约束下得到的最优解路径的长度；而在 AnyFoodSearchProblem 中，我所设计的启发式函数是在忽略墙壁的约束后，各个食物以及吃豆人当前位置之间的距离的最大值，这实际上相当于只考虑了吃豆人收集其中一个或两个食物所需的最短距离。

同时，我在设计启发式函数的过程中也意识到：启发式函数所需要的代价应该尽量小，即可以快速得到。在设计 AnyFoodSearchProblem 的启发式函数时，我曾尝试在忽略墙壁的约束后将使用 UCS 算法得到的解路径的长度作为函数值，但是经过实验发现当食物个数略多时，所需要的时间便变得不可接受，而这是因为该问题的状态数量随着食物的数量指数级增长。实际上，实验证明，只需把各个食物到当前位置的曼哈顿距离的最大值当作启发函数值便可以获得不错的效果。

在设计启发式函数的过程中，我也意识到：要想设计的启发式函数是一致的，往往需要启发式函数值不随着状态发生突变。例如我在设计 AnyFoodSearchProblem 的启发式函数时，尝试用以剩余食物位置以及吃豆人当前位置为结点的最小生成树的总代价作为启发式函数值，但是由于随着吃豆人的位置改变，最小生成树可能会发生多处变化，所以这一启发式函数是不一致的。

从实验的最后一题中我体会到了即使状态空间很大，如果不追求最优解的话，可以在较短时间内得到一个解路径——即使不是最优的，但通常也是也是可以接受的。而像 UCS 和 BFS 这类算法，当面对较多的状态时，即使能够保证给出最优解，但是由于展开的结点数很多，所需要的时间代价往往是不可接受的。