同济大学

TONGJI UNIVERSITY

《人工智能原理课程设计》

实验报告

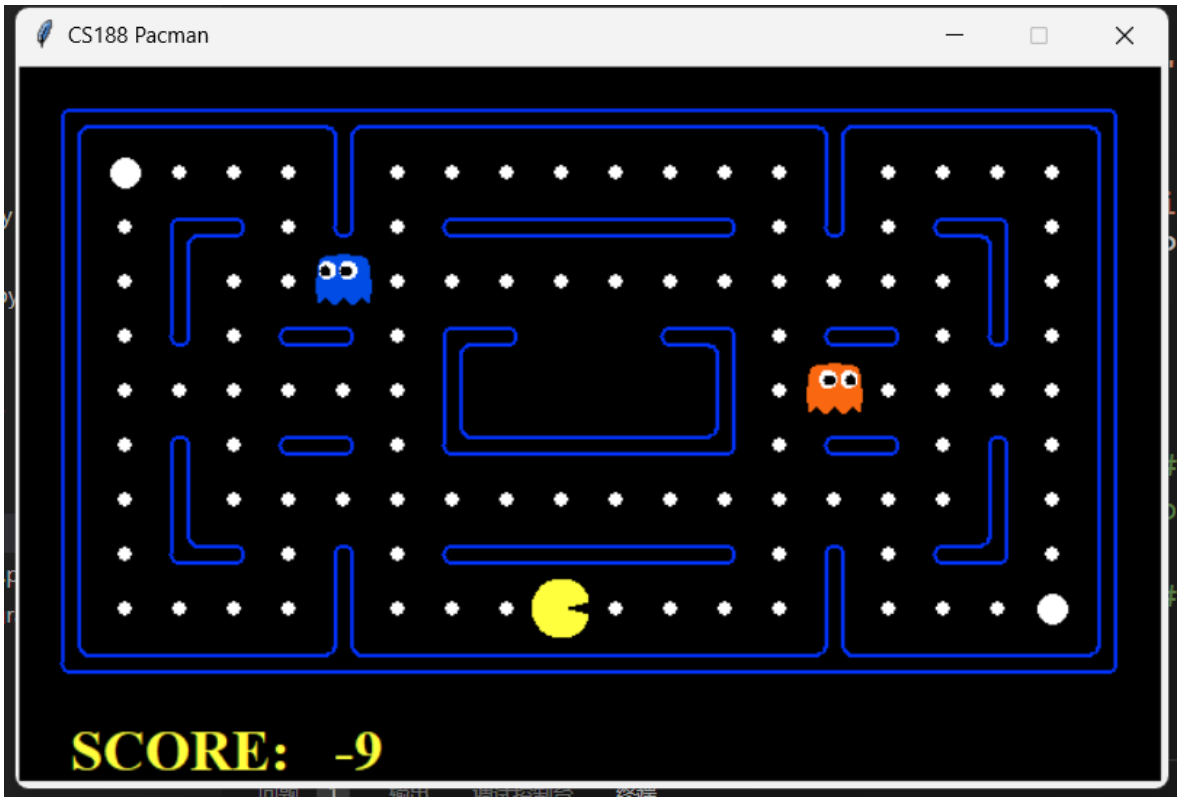
实验名称	Project1:Search
学号姓名	2154057 汪清濯
学院（系）	电子与信息工程学院
专　　业	计算机科学与技术
任课教师	张红云/苗夺谦
日　　期	2023 年 4 月 6 日

Project1：Search

一、实验简介

在本次实验中，我们需要设计各类搜索算法以及智能体，来让吃豆人智能体达到特定的位置或有效的搜集食物。我们将主要完成 search.py 和 searchAgent.py 两个 python 文件。

下载解压代码后，我们就来到了吃豆人的世界。我们可以执行 python pacman.py 来自己手动操控吃豆人完成游戏。也可以通过调用各种参数来让智能体自动寻找路径完成任务。本实验为我们提供了很多有用的参数，合理的调用它们可以很好的测试我们的代码。



二、实验内容

Q1. Finding a Fixed Food Dot using Depth First Search——利用深度优先搜索找到固定的食物

Q1.1 问题概述

Q1.1.1 直观描述

在 searchAgents.py 中，一个 SearchAgent 类已经实现完成，它可通过给定的搜搜算法找出一条穿过 Pacman 世界的路径，并逐步执行该路径。我们的工作就是完成这个搜索算法。

在本问题中，我们需要完成的是 DFS 深度优先搜索算法，我们需要给智能体返回一个行动列表，每次行动都必须合法（不能超出地图、不能撞墙）。

Q1.1.2 已有代码的阅读和理解

本问题中，主要需要阅读和理解的是 SearchAgent、PositionSearchProblem 以及 Stack 三个类

SearchAgent 类：

```
1. class SearchAgent(Agent):
```

该类继承自 game.py 中定义的 Agent 基类，这个非常通用的搜索智能体使用提供的搜索算法找到路径，然后返回遵循该路径的操作。默认情况下，此代理在 PositionSearchProblem 上运行 DFS 以查找位置 (1,1)。它内部定义三个函数

1. **构造函数**

```
1.def __init__(self, fn='depthFirstSearch', prob='PositionSearchPro
   blem', heuristic='nullHeuristic')
```

参数：fn：智能体将要使用的搜索函数，默认为 dfs

prob：要解决的问题，默认为寻找具体的位置

heuristic：启发函数，默认为无

主要功能：找到正确的函数和要解决的问题并初始化对象。它的内部使用了一些高级的 pytohn magic 来找到正确的函数和问题，如果未找到，它会手动设置错误，输出错误提示并退出程序。

2. registerInitialState 函数

```
1.def registerInitialState(self, state)
```

参数：state：一个 GameState 对象（在 pacman.py 中定义）

主要功能：智能体根据地图布局和搜索函数计算出一条通往目标的路径并存在 actions 局部变量中，同时初始化一些如开始时间、总花费代价等局部变量。

3. getAction 函数

```
1.def getAction(self, state):
```

参数：state：一个 GameState 对象（在 pacman.py 中定义）

主要功能：一步步返回 registerInitialState 函数中选择的路径的动作，结束时返回 Directions.STOP。

PositionSearchProblem 类

```
1.class PositionSearchProblem(search.SearchProblem):
```

该类继承自 search.py 中定义的 SearchProblem 基类。搜索问题定义状态空间、开始状态、目标状态测试、后继函数和代价函数。此搜索问题用于查找路径到达地图上的特定点。状态空间由(x,y)坐标组成。它内部定义五个函数

1. 构造函数

```
def __init__(self, gameState, costFn=lambda x: 1, goal=(1, 1), start=None,
    warn=True, visualize=True)
```

参数：gameState：一个 GameState 对象

costFn：单步代价函数，应为非负数，默认为 1

goal：目标位置，默认为（1，1）

start：起始位置，默认为 None

warn：是否警告，默认为 True

visualize：是否可视，默认为 True

主要功能：定义状态空间，初始化各局部变量，如起始位置，目标位置等。

2. getStartState 函数

```
def getStartState(self)
```

参数：无

主要功能：返回起始位置

3. isGoalState 函数

```
def isGoalState(self, state)
```

参数：state：当前位置

主要功能：判断当前位置是否是目标位置，是则返回 True，反之返回 False

4. getSuccessors 函数

```
def getSuccessors(self, state)
```

参数：state：当前位置

主要功能：搜索当前位置能到达的后继位置，判断它是否合法，若合法则定义一个（后继位置、移动、花费代价）三元组，并将三元组加入 successors 列表中，返回 successors 列表。

5. getCostOfActions 函数

```
def getCostOfActions(self, actions)
```

参数：actions：一个移动列表

主要功能：计算从起始状态开始完成此次移动的花费并返回

Stack 类：

```
class Stack:
```

一个满足先进后出排队策略的容器，即数据结构栈。内部定义四个成员函数

1.构造函数

```
1.  def __init__(self)
```

参数：无
主要功能：初始化一个空列表作为栈

2.push 函数

```
1. def push(self,item)
```

参数：item：将要入栈的元素
主要功能：将 item 压入栈

3.pop 函数

```
1. def pop(self)
```

参数：无
主要功能：弹出栈顶元素

4.isEmpty 函数

```
1. def isEmpty(self)
```

参数：无；
主要功能：判断栈是否空

Q1.1.3 解决问题的思路与想法

深度优先搜索是一种优先扩展当前搜索树中最深结点的搜索方法，这种特性很适合用递归或者用栈模拟递归的方式去做。本实验给出的函数不方便进行递归操作，并且本实验在 util.py 中为我们提供了 stack 类，我选择利用栈模拟递归进行深搜。

Q1.2 算法设计

Q1.2.1 算法功能

先扩展搜索最深结点，直到无后继结点时回溯，回溯到次深结点继续搜索，直到搜索到目标状态。

Q1.2.2 设计思路

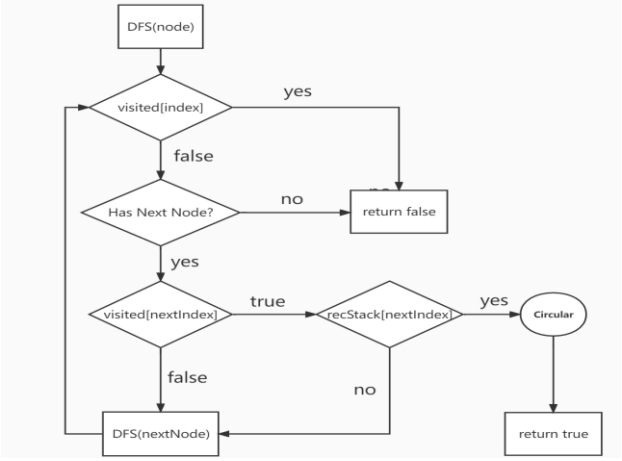
利用 stack 数据结构模拟深搜。开始结点先入栈，栈非空时，先判断栈顶元素是否是目标状态，若是，退出搜索，返回行动列表。若不是，搜索栈顶元素的合法后继结点，将它们依次入栈。

搜索过的结点都加入一个 visit 列表，若要搜索的结点已经在 visit 列表中了，则不入栈，若当前搜索结点的所有合法后继结点全部在 visit 中，则弹出栈顶和 answer 的最后一位。

我为了节省空间，并没有在每个入栈元组中存储它的路径，而是只维护了一个 answer 行动列表，这个列表也可以看作一个栈。这样每次搜索时若栈顶已在 visit 中，先弹出栈顶，但不能直接弹出 answer 最后一位。而是要先判断 answer 的最后一次行动是否是栈顶元素的行动，即当前栈顶的后继元素已经搜索完成。若是，则弹出栈顶，否则继续搜索。这样做在地图很大时只用维护一个行动列表，能为空间复杂度降一级幂。

时间复杂度：O(b^m)，空间复杂度：O(bm)，b 是状态空间分支因子，m 是任一结点最大深度。

Q1.2.3 算法流程图



Q1.3 算法实现

下面贴出代码，细节已在注释中标明

```
def depthFirstSearch(problem: SearchProblem):
    # 若开始位置及是目标位置，返回一个空列表
    if problem.isGoalState(problem.getStartState()) == True:
        return []

    st = util.Stack()  # 栈内存状态元组
    for position, action, cost in problem.getSuccessors(problem.getStartSta
te()):
        st.push((position, action))
    answer = []  # 节省空间，只维护一个 ans 列表
     visit = [problem.getStartState()]

     while not st.isEmpty():
         # 获取头元素，我不知道为什么 stack 类没有 top 方法，如果只维护一个 answer
列表就不能直接 pop 栈顶
         now_position, now_action = st.list[-1]
         if now_position in visit:
             st.pop()
             if now_action == answer[-1]:  # 此点下的深搜已经完成，碰到自己再弹
出 ans
                 answer.pop()
             continue

         answer.append(now_action)
```

```
        visit.append(now_position)
        if problem.isGoalState(now_position):
            break

        tag = 0  # tag 判断当前状态是否能继续搜索
        for next_position, action, cost in problem.getSuccessors(now_posit
    ion):
            if not (next_position in visit):
                st.push((next_position, action))
                tag = 1

        # 若不能继续搜索，弹出栈和 ans 列表
        if tag == 0:
            st.pop()
            answer.pop()
            continue

    return answer
    util.raiseNotDefined()
```

Q1.4 实验结果

所有测试用例均已通过

```
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 ###
```

我们可以返回错误的答案来查看测试用例

```
    graph:
            B1          E1
           ^  \        ^  \
          /    V      /    V
        *A --> C --> D --> F --> [G]
          \    ^      \    ^
           V  /         V  /
            B2          E2

        A is the start state, G is the goal.  Arrows mark
        possible state transitions.  This graph has multiple
        paths to the goal, where nodes with the same state
        are added to the fringe multiple times before they
        are expanded.
    student solution:               []
    student expanded_states:        ['A', 'B2', 'C', 'D', 'E2', 'F']

    correct solution:               ['2:A->B2', '0:B2->C', '0:C->D', '2:D->E2', '0:E2->F', '0:F->G']
    correct expanded_states:        ['A', 'B2', 'C', 'D', 'E2', 'F']
    correct rev_solution:           ['0:A->B1', '0:B1->C', '0:C->D', '0:D->E1', '0:E1->F', '0:F->G']
    correct rev_expanded_states:    ['A', 'B1', 'C', 'D', 'E1', 'F']
```

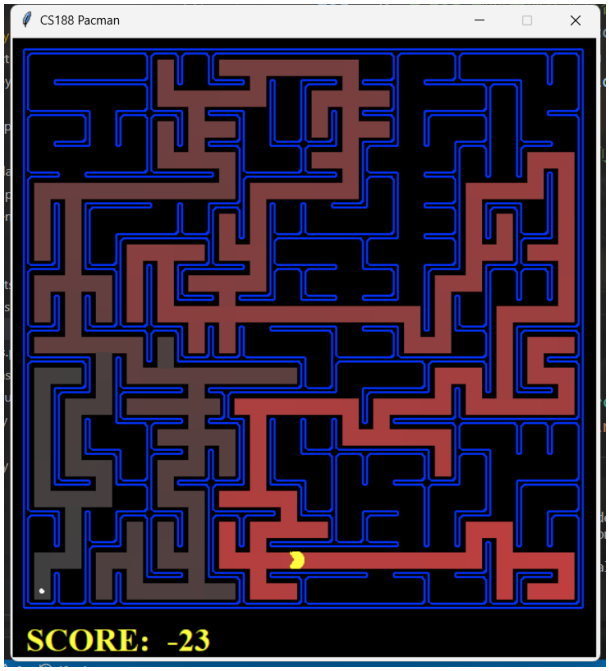
不难发现前四个测试点每个测试点给出的都是一个图，图由简单变复杂，起点到终点的路径可能有一条也可能有多条。测试用例的判断依据就是路径是否正确以及扩展的结点是否符合深搜要求。它会反转起点、终点进行两次判断。

```
***      student solution length: 0
***      student solution:

***
***      correct solution length: 130
***      correct (reversed) solution length: 246
***      correct solution:
```

最后一个测试点给出了一个较大的图，判断的是路径是否正确以及解决路径的长度。

使用命令 python pacman.py -l bigMaze -z .5 -p SearchAgent 查看图形界面下的深搜



红色越深代表搜索的越早，可以直观的看出满足深搜要求

```
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 390
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Q2. Breadth First Search——广度优先搜索

Q2.1 问题概述

Q2.1.1 直观描述

在 search.py 的 breadthFirstSearch 函数中实现按照搜索深度逐层扩展结点的广度优先搜索(BFS) 算法。

Q2.1.2 已有代码的阅读和理解

本问题中需要额外阅读理解的是 uitl.py 中定义的 Queue 类

Queue 类：

```
1. class Queue:
```

一个满足先进先出排队顺序的容器，即数据结构队列。内部定义四个成员函数

1.构造函数

```
1. def __init__(self)
```

参数：无

主要功能：初始化一个队列

2.push 函数

```
2. def push(self,item)
```

参数：item：将要入队的元素

主要功能：将 item 压入队列

3.pop 函数

```
2. def pop(self)
```

参数：无

主要功能：弹出队首元素

4.isEmpty 函数

```
2. def isEmpty(self)
```

参数：无；

主要功能：判断队列是否空

Q2.1.3 解决问题的思路与方法

广度优先搜索是一种优按搜索深度逐层扩展搜索结点的搜索方法，这种特性很适合用队列模拟。本实验在 util.py 中为我们提供了 Queue 类，我选择利用队列模拟进行广搜。

Q2.2 算法设计

Q2.2.1 算法功能

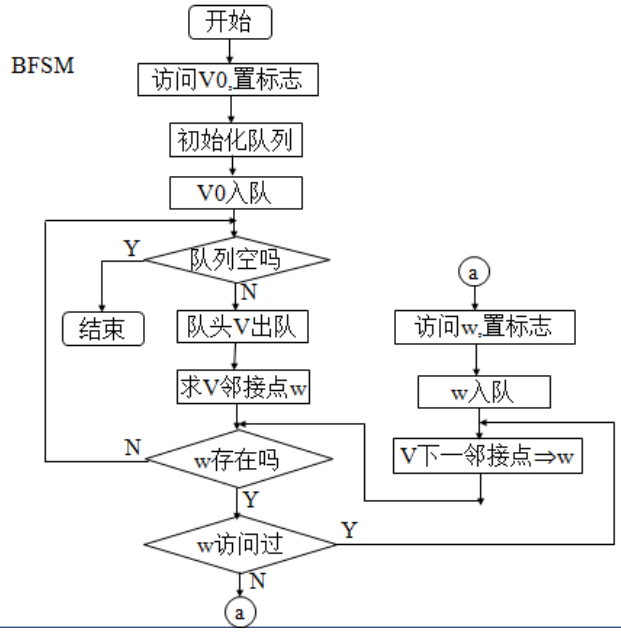
从根节点开始，按照搜索深度逐层扩展结点，直到找到目标结点为止。

Q2.2.2 设计思路

利用 queue 数据结构模拟广搜，队列中存储起始位置和行动列表的二元组。起始结点先入队，加入 visit 列表，队列非空时，弹出队首元素，判断队首元素是否在 visit 中，若在则继续循环，不在则加入 visit 列表。判断该元素是否是目标状态，是则退出搜索，返回行动列表。反之则搜索该元素的合法后继，让它们依次入队

时间复杂度：O(bˆd)，空间复杂度：O(bˆd)，b 是分支因子，d 是起点到终点的最短路径长度。

Q2.2.3 算法流程图



Q2.3 算法实现

下面贴出代码，细节已在注释中标明

```
1.  def breadthFirstSearch(problem: SearchProblem):
2.      if problem.isGoalState(problem.getStartState()) == True:
3.          return []  # 如果起点就是终点，直接返回空路径
4.
5.      qu = util.Queue()  # 队列用于存储待访问的节点
6.      answer = []  # 答案列表用于存储路径
7.      visit = [problem.getStartState()]  # 访问列表用于记录已经访问过的节点
8.
9.      # 将起点的邻居节点添加到队列中
10.     for position, action, cost in problem.getSuccessors(problem.getStartState()):

11.         qu.push((position, [action]))  # 队列中存当前位置和路径
12.
13.     # 循环访问队列中的节点
14.     while not qu.isEmpty():
15.         now_position, now_path = qu.pop()  # 取出队列中的节点
```

```
        if now_position in visit:
            continue  # 如果该节点已经访问过，跳过本次循环

        visit.append(now_position)  # 将当前节点添加到访问列表中

        if problem.isGoalState(now_position):
            answer = now_path  # 如果当前节点是终点，更新答案为当前路径并退出循环
            break

        # 将当前节点的邻居节点添加到队列中
        for next_position, action, cost in problem.getSuccessors(now_position):
            if not next_position in visit:
                qu.push((next_position, now_path+[action]))  # 路径更新

    return answer  # 返回最终答案

    # 如果未实现算法，则抛出异常
    util.raiseNotDefined()
```

Q2.4 实验结果

所有测试用例均已通过

```
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 ###
```

查看测试用例

```
graph:
        B1          E1
      ^  \        ^  \
     /    V      /    V
   *A --> C --> D --> F --> [G]
     \    ^      \    ^
      V  /        V  /
       B2          E2

   A is the start state, G is the goal.  Arrows mark
   possible state transitions.  This graph has multiple
   paths to the goal, where nodes with the same state
   are added to the fringe multiple times before they
   are expanded.
student solution:               []
student expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

correct solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
correct expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
correct rev_solution:           ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
correct rev_expanded_states:    ['A', 'B2', 'C', 'B1', 'D', 'E2', 'F', 'E1']
```

发现与 dfs 的测试用例相同，但返回路径和扩展结点需要按照 bfs 要求

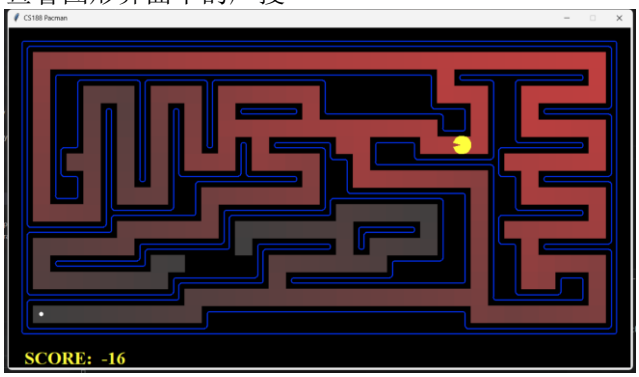
利用 bfs 解决八数码问题 python eightpuzzle.py

```
A random puzzle:
-------------
| 1 | 5 | 4 |
-------------
| 3 | 2 |   |
-------------
| 6 | 7 | 8 |
-------------
BFS found a path of 7 moves: ['up', 'left', 'down', 'right', 'up', 'left', 'left']
```

```
After 7 moves: left
-------------
|   | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
```

成功解决

查看图形界面下的广搜



红色越深代表搜索的越早，可以直观的看出满足深搜要求

```
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Q3.Varying the Cost Funtion——改变成本函数

Q3.1 问题概述

Q3.1.1 直观描述

我们已经找到了最少动作路径，但我们希望可以找到其他意义上的“最佳”路径。通过改变成本函数，我们可以让 Pacman 寻找不同的路径。 例如，我们可以对鬼魂出没地区的危险步骤收

装 订 线

取更多费用，或者对食物丰富地区的步骤收取更少费用，而理性的 Pacman 智能体应该调整其行为以做出响应。我们需要在 search.py 的 uniformCostSearch 函数中实现一致代价搜索算法。

Q3.1.2 已有代码的阅读和理解

本问题中需要阅读理解的是 util.py 中的 PriorityQueue 类以及 searchAgent.py 中的 StayEastSearchAgent 和 StayWestSearchAgent 类

PriorityQueue 类

```
1. class PriorityQueue
```

实现优先级队列数据结构。每个插入的项目具有与之关联的优先级并且用户通常希望快速检索队列中优先级最低的项目。此数据结构允许 O(1) 访问最低优先级的项目。其中，元素的优先级由 priority 参数指定。在实现中，代码使用了 Python 自带的 heapq 库来实现堆。该类内部定义五个成员函数

1.构造函数

```
1. def __init__(self):
```

参数：无

主要功能：初始化优先队列，创建一个空堆和计数器 count。

2.push 函数

```
1. def push(self, item, priority)
```

参数：item：要加入队列的元素

Priority：优先级

主要功能：向优先队列中添加一个元素 item，并指定其优先级 priority。

3.pop 函数

```
1. def pop(self)
```

参数：无

主要功能：从优先队列中取出最小优先级的元素，并将其从队列中删除。

4.isEmpty 函数

```
1. def isEmpty(self)
```

参数：无

主要功能：检查优先队列是否为空。

5.update 函数

```
1. def update(self, item, priority)
```

参数：item：元素

Priority：优先级

主要功能：更新优先队列中元素 item 的优先级为 priority。如果元素不存在，则向队列中添加该元素。如果元素已经存在，且新的优先级比原先优先级更高，则更新其优先级并重新构建堆。

StayEastSearchAgent 类

```
1. class StayEastSearchAgent(SearchAgent):
```

继承自 SearchAgent 类，仅改变了 costFn，规定搜索方法为 ucs。进入位置 (x,y) 的代价函数是 1/2^x。

StayWestSearchAgent 类

```
1. class StayWestSearchAgent(SearchAgent):
```

继承自 SearchAgent 类，仅改变了 costFn，规定搜索方法为 ucs。进入位置 (x,y) 的代价函数是 2^x。

Q3.1.3 解决问题的思路与方法

一致代价搜索扩展的是路径消耗最小的未扩展结点。可以将他们排入优先队列，路径消耗作为 priority。代价最小的优先。

Q3.2 算法设计

Q3.2.1 算法功能

找到一条起点到终点的代价最小的路径。

Q3.2.2 设计思路

利用优先队列模拟一致代价搜索，队列中存储起始位置、行动列表、花费代价的三元组，以花费代价为 priority。起始结点先入队，加入 visit 列表，队列非空时，弹出队首元素，判断队首元素是否在 visit 中，若在则继续循环， 不在则加入 visit 列表。判断该元素是否是目标状态，是则退出搜索，返回行动列表。反之则搜索 该元素的合法后继，让它们依次入队

Q3.3 算法实现

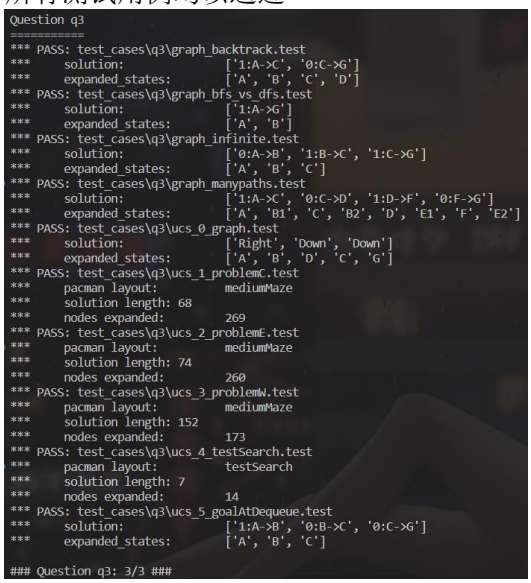
下面贴出代码，细节已在注释中标明

```
1. def uniformCostSearch(problem: SearchProblem):
2.     """Search the node of least total cost first."""
```

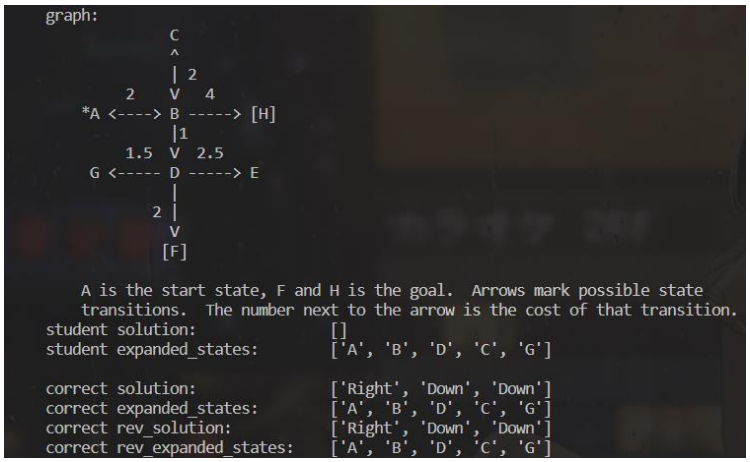
```
3.
4.      if problem.isGoalState(problem.getStartState()) == True:  # 判断起点是否是目标点
5.          return []
6.
7.      myPriorityQueue = util.PriorityQueue()  # 创建一个优先队列
8.      answer = []  # 存储答案路径
9.      visit = [problem.getStartState()]  # 存储访问过的点
10.
11.     for position, action, cost in problem.getSuccessors(problem.getStartState()):
12.         # 将起点的后继节点按 cost 作为 priority 加入队列
13.         myPriorityQueue.push((position, [action], cost), cost)
14.
15.     while not myPriorityQueue.isEmpty():  # 如果队列不为空
16.         now_position, now_path, now_cost = myPriorityQueue.pop()  # 取出当前 cost 最小的点
17.         if now_position in visit:  # 如果已经访问过，则直接跳过
18.             continue
19.         visit.append(now_position)  # 否则标记为已访问
20.
21.         if problem.isGoalState(now_position):  # 如果当前点是目标点
22.             answer = now_path  # 更新答案路径
23.             break
24.
25.         for next_position, action, cost in problem.getSuccessors(now_position):
26.             if not next_position in visit:
27.                 # 这里可以用当前的 cost+下一步的 cost 算 action 的 cost，也可以直接调用 problem.getCostOfActions()
28.                 # 我认为用 cost+下一步的 cost 更快速，但是会多花一部分空间存储每一步的 cost
29.                 myPriorityQueue.push(
30.                     (next_position, now_path+[action], now_cost+cost), now_cost+cost)
31.
32.     return answer
```

Q3.4 实验结果

所有测试用例均以通过



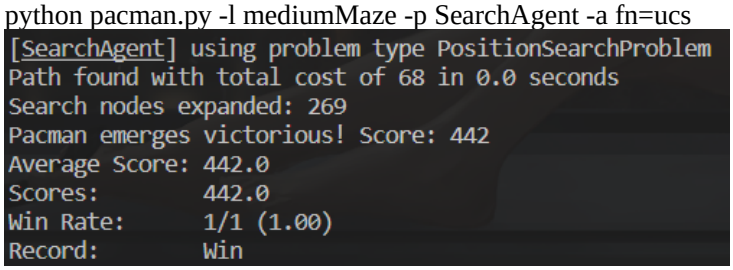
查看测试用例



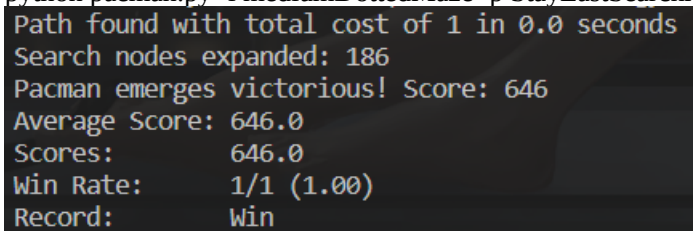
和前两个测试相同，有给出图和判断路径长度两种测试用例，这一问在前两问测试用例的基础上增加了带有边权的新图。判断依据是路径是否正确以及扩展的结点是否符合 ucs 的要求。

三个不同智能体的测试也正确

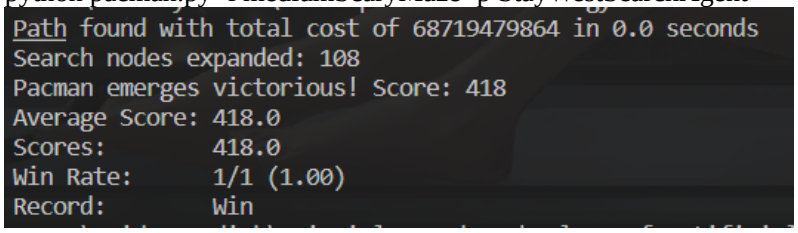
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs



python pacman.py -l mediumDottedMaze -p StayEastSearchAgent



python pacman.py -l mediumScaryMaze -p StayWestSearchAgent



Q4.A*search——A*搜索

Q4.1 问题概述

Q4.1.1 直观描述

在 search.py 中的空函数 aStarSearch 中实现 A* 搜索。A* 搜索以启发函数作为参数。启发函数采用两个参数：搜索问题中的状态（主要参数）和问题本身（用于参考信息）。

Q4.1.2 已有代码的阅读和理解

本问题中需要阅读理解的是 searchAgent.py 中的 manhattanHeuristic 函数

manhattanHeuristic 函数

```
1. def manhattanHeuristic(position, problem, info={})
```

参数：position：当前状态

Problem：搜索问题

主要功能：返回曼哈顿距离作为估计耗散值

Q4.1.3 解决问题的思路与方法

A*搜索与一致代价搜索唯一的区别是 priority，A*以 cost+heuristic 作为 priority，剩下部分相同。

Q4.2 算法设计

装

Q4.2.1 算法功能

根据启发函数找到一条最优路径。

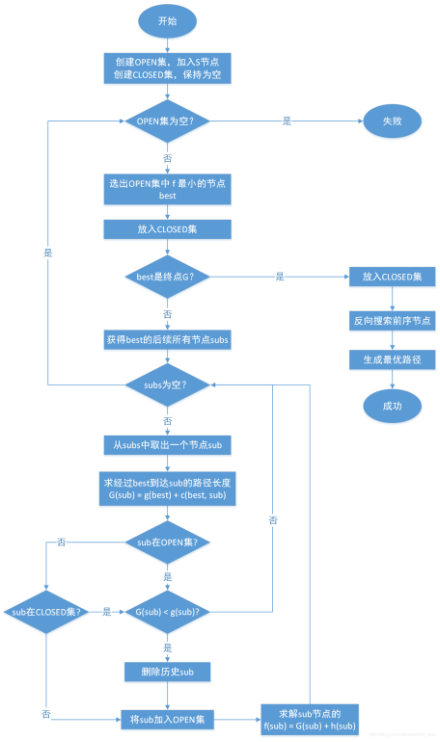
Q4.2.2 设计思路

订

利用优先队列模拟 A*搜索，队列中存储起始位置、行动列表、花费代价的三元组，以 cost+heuristic 为 priority。起始结点先入队，加入 visit 列表，队列非空时，弹出队首元素，判断队首元素是否在 visit 中，若在则继续循环， 不在则加入 visit 列表。判断该元素是否是目标状态，是则退出搜索，返回行动列表。反之则搜索该元素的合法后继，让它们依次入队

Q4.2.3 算法流程图

线



Q4.3 算法实现

下面贴出代码，细节已在注释中标明

```
1.  def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
2.      # 如果起始点就是终点，那么不用搜索
3.      if problem.isGoalState(problem.getStartState()) == True:
4.          return []
5.
6.      # 用优先队列维护待搜索状态
7.      myPriorityQueue = util.PriorityQueue()
8.      answer = []
9.      visit = [problem.getStartState()]
10.
11.     # 将起始点的后继状态加入优先队列
12.     for position, action, cost in problem.getSuccessors(problem.getStartState()):
13.         # 计算估价函数值并作为优先级
14.         myPriorityQueue.push(
15.             (position, [action], cost), cost+heuristic(position, problem))
16.
17.     while not myPriorityQueue.isEmpty():
18.         # 取出优先级最高的状态进行扩展
19.         now_position, now_path, now_cost = myPriorityQueue.pop()
20.         if now_position in visit:
21.             continue
22.         visit.append(now_position)
23.
24.         # 如果该状态是终点，则得到解并退出搜索
25.         if problem.isGoalState(now_position):
26.             answer = now_path
27.             break
28.
29.         # 将当前状态的后继状态加入优先队列
30.         for next_position, action, cost in problem.getSuccessors(now_position):
31.             if not next_position in visit:
32.                 # 计算估价函数值并作为优先级
33.                 myPriorityQueue.push(
34.                     (next_position, now_path+[action], now_cost+cost), now_cost+c
    ost+heuristic(next_position, problem))
35.
36.     return answer
37.     util.raiseNotDefined()
```

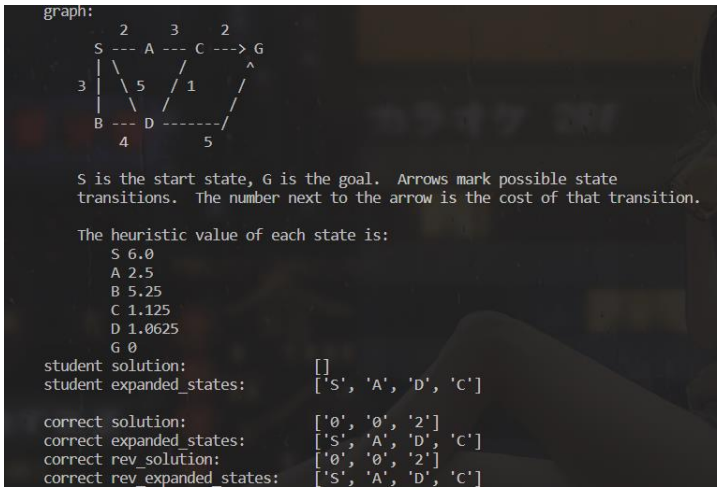
Q4.4 实验结果

所有测试用例均以通过

```
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 ###
```

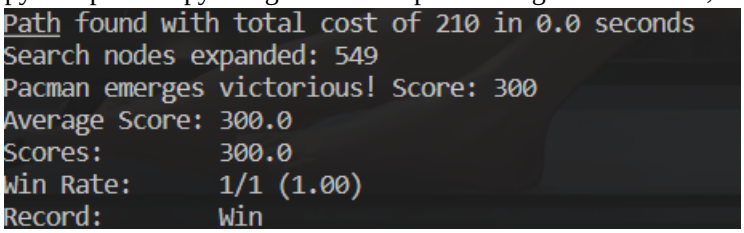
查看测试用例



和一致代价搜索的样例基本相同，新增了启发函数的值。判断依据是路径是否正确以及扩展的结点是否符合 A* 的要求。

在大迷宫下也能正确搜索

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



Q5.Finding All the Corners—— 找出所有的角落

Q5.1 问题概述

Q5.1.1 直观描述

制定一个新的搜索问题，在迷宫中，共有四个角落，新的搜索问题是找到通过迷宫并触及四个角的最短路径。在 searchAgents.py 中实现 CornersProblem 搜索问题。 我需要选择一种状态表示的方法，该表示能检测是否已到达所有四个角。

Q5.1.2 已有代码的阅读和理解

要阅读的代码即要完成的类

Q5.1.3 解决问题的思路与方法

解决该问题的关键是如何表示一种状态，让我们能知道当前是否访问过四个角落。我们可以在之前状态的基础上加一个元组，来存储已经访问过的角落。当元组的长度为 4 时，则表示四个角落都已经访问过。

Q5.2 算法设计

Q5.2.1 算法功能

设置一个搜索问题，让智能体找到一条穿过迷宫并触及四个角的最短路径。

Q5.2.2 设计思路

为每个状态添加一个元组，元组用来存储访问过的角落。当元组的长度为 4 时，表示四个结点都已经访问过，到达目标状态。获取后继元素时，先判断其是否合法，若合法，再判断其是否为角落，若为角落且当前状态未访问过，则将其添加入后继状态的角落元组中。

Q5.3 算法实现

下面贴出代码，细节已在注释中标明

```
1.  def getStartState(self):
2.          """
3.          Returns the start state (in your state space, not the full Pacman state
4.          space)
5.          """
6.          visited_corners = ()  # 存储访问过的角落的元组
7.          # 如果起始位置在一个角落上，把它加入访问过的角落列表中
8.          if self.startingPosition in self.corners:
9.              visited_corners += (self.startingPosition,)  # 连接元组
10.         return (self.startingPosition, visited_corners)
11.         util.raiseNotDefined()
12.
13. def isGoalState(self, state: Any):
14.         """
15.         Returns whether this search state is a goal state of the problem.
16.         """
17.         # 如果访问过的角落数等于 4，则表示已经访问到所有角落，返回 True
18.         visited_corners = state[1]
19.         return len(visited_corners) == 4
20.         util.raiseNotDefined()
21.
22. def getSuccessors(self, state: Any):
23.   successors = []
24.   for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.
    WEST]:
25.         #   x,y = currentPosition
26.         #   dx, dy = Actions.directionToVector(action)
27.         #   nextx, nexty = int(x + dx), int(y + dy)
28.         #   hitsWall = self.walls[nextx][nexty]
29.
30.         x, y = state[0]
31.         dx, dy = Actions.directionToVector(action)
32.         nextx, nexty = int(x + dx), int(y + dy)
33.         hitsWall = self.walls[nextx][nexty]
34.
35.         if not hitsWall:
36.             new_visited_corners = state[1]
37.             next_position = (nextx, nexty)
38.             # 若下一步是角落且当前未访问过，则更新 visit_corners
39.             if (next_position in self.corners) and not (next_position in new_visite
    d_corners):
40.                 new_visited_corners += (next_position,)
41.             # 题目要求 cost 为 1
42.             successor = ((next_position, new_visited_corners), action, 1)
43.             successors.append(successor)
44.   self._expanded += 1  # DO NOT CHANGE
45.   return successors
```

Q5.4 实验结果

所有测试用例均以通过

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

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

查看测试用例

```
Question q5
===========
*** FAIL: test_cases\q5\corner_tiny_corner.test
*** Corners missed: [(1, 1), (1, 6), (6, 1), (6, 6)]
*** Tests failed.
```

判断依据是在 Q2 的 bfs 正确的情况下，是否访问了四个角落。

迷宫中也能正确搜索

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

```
Path found with total cost of 106 in 0.2 seconds
Search nodes expanded: 2448
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Q6. Corners Problem: Heuristic——角落问题：启发式

Q6.1 问题概述

Q6.1.1 直观描述

在 cornersHeuristic 中为 CornersProblem 实现一个可接受的、一致的启发式算法。可接受性与一致性：启发式算法只是取得搜索状态并返回估计最近目标成本的数字的函数。 更有效的启发式方法将返回更接近实际目标成本的值。 为了被接受，启发式值必须是到最近目标的实际最短路径成本的下限（并且非负）。 为了保持一致，它必须额外地认为，如果一个动作的成本为 c，那么采取该动作最多只能导致启发式下降 c。

Q6.1.2 已有代码的阅读和理解

无。

Q6.1.3 解决问题的思路与方法

要实现一个较好的启发式算法，要保证算法的可接受性、一致性、并且要尽量接近实际目标成本的值的下限。根据 Q4 的启发，我知道了在迷宫中一点到达另一点的最短距离是他们的曼哈顿距离。若我们找到从当前结点走完所有未访问过的角落的曼哈顿距离，可保证可接受性与一致性。

Q6.2 算法设计

Q6.2.1 算法功能

返回角落问题的启发函数的值。

Q6.2.2 设计思路

若我们找到从当前结点走完所有未访问过的角落的曼哈顿距离，可以保证该距离小于实际目标成本，即能保证可接受性。

一致性证明：

由曼哈顿距离的性质可知：对于每个结点 n 和通过任一行动 a 生成的 n'的每个后继结点，满足 $h(n) \leq c(n,a,n') + h(n')$

，则 $f(n') = g(n') + h(n') = g(n) + c(n,a,n') + h(n') \geq g(n) + h(n) = f(n)$

证毕

Q6.3 算法实现

下面贴出代码，细节已在注释中标明

```python
def cornersHeuristic(state: Any, problem: CornersProblem):
    # 获取角落坐标和墙的位置
    corners = problem.corners
    walls = problem.walls

    # 初始化
    now_state = state[0]
    visited_corners = state[1]
    no_vis_corners = []

    # 找出未访问过的角落
    for corner in corners:
        if corner not in visited_corners:
            no_vis_corners.append(corner)

    total_dist = 0

    # 寻找未访问过的角落
    while no_vis_corners:
        dist = 99999
        tmp_corner = None
        # 对于未访问过的角落，寻找到当前位置最近的角落
        for corner in no_vis_corners:
            if util.manhattanDistance(now_state, corner) < dist:
                dist = util.manhattanDistance(now_state, corner)
                tmp_corner = corner
        # 将到该角落的距离加到总距离中
        total_dist += dist
        # 将当前状态更新为该角落的位置
        now_state = tmp_corner
        # 将该角落从未访问过的角落列表中删除
        no_vis_corners.remove(now_state)

    return total_dist
```

Q6.4 实验结果

所有测试用例均已通过

```
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', 'Nort
th', 'West', 'West', 'West', 'West', 'South', 'South', 'East',
 'South', 'South', 'West', 'West', 'East', 'East', 'North', 'N
', 'East', 'East', 'East', 'East', 'North', 'North', 'East', '
uth', 'South', 'South', 'East', 'East', 'North', 'North', 'Eas
North', 'North', 'West', 'West', 'North', 'North', 'East', 'Ea
path length: 106
*** PASS: Heuristic resulted in expansion of 901 nodes

### Question q6: 3/3 ###
```

测试用例说明：

```
must use non-trivial heuristic
```

程序会先判断启发式是否非平凡、可接受、一致，若不满足上述任一条件，则测试不通过。接下来会根据扩展结点的个数来给出评分，扩展的结点越少，评分越高。

在迷宫下成功搜索

python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5

```
Path found with total cost of 106 in 0.0 seconds
Search nodes expanded: 901
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Q7. Eating All The Dots——吃掉所有的点

Q7.1 问题概述

Q7.1.1 直观描述

现在我们要解决一个困难的搜索问题：用尽可能少的步骤吃完所有的 Pacman 食物。 为此，我们需要一个新的搜索问题定义来形式化食物清理问题：searchAgents.py 中的 FoodSearchProblem。解决方案被定义为收集 Pacman 世界中所有食物的路径。在 searchAgents.py 中完成 foodHeuristic，为 FoodSearchProblem 使用一致的启发式算法。

Q7.1.2 已有代码的阅读和理解

本问题需要阅读理解 FoodSearchProblem 类和 mazeDistance 函数

FoodSearchProblem 类

```
1. class FoodSearchProblem
```

这个类的功能与之前的搜索问题功能类似，需要找到一条吃点所有食物的路径

1.构造函数

```
1. def __init__(self, startingGameState: pacman.GameState)
```


装 订 线

参数：startingGameState：游戏状态

主要功能：接受一个 pacman.GameState 类型的参数 startingGameState，设置起始状态、墙、起始游戏状态、扩展次数和启发式信息字典。

2. getStartState 函数

```
1. def getStartState(self)
```

参数：无

主要功能：返回起始状态

3. isGoalState 函数

```
1. def isGoalState(self, state)
```

参数：state：当前状态

主要功能：判断当前状态是否为目标状态，即是否所有食物都被收集。

4. getSuccessors 函数

```
1. def getSuccessors(self, state)
```

参数：state：当前状态

主要功能：返回当前状态的所有后继状态，以及到达每个后继状态的动作和代价。

5. getCostOfActions 函数

```
1. def getCostOfActions(self, actions)
```

参数：actions：行动列表

主要功能：返回执行一系列动作的代价。如果这些动作包括非法移动，则返回 999999。

mazeDistance 函数

```
1. def mazeDistance(point1: Tuple[int, int], point2: Tuple[int, int], ga
   meState: pacman.GameState)
```

参数：point1、point2：迷宫内的两点

gameState：游戏状态

主要功能：使用已经构建的搜索函数返回任意两点之间的迷宫距离。 gameState 可以是任何游戏状态——吃豆人在该状态下的位置将被忽略。

Q7.1.3 解决问题的思路与方法

当我愁眉不展时，searchAgent.py 中的 mazeDistance 函数让我眼前一亮，能够知道任意两点间的实际距离无疑对解决这个问题很有帮助。利用这个函数，我可以设计出来一个可接受的、一致的算法。

Q7.2 算法设计

Q7.2.1 算法功能

找到吃点所有食物搜索问题的启发式值。

Q7.2.2 设计思路

找到距离当前点最远的食物的实际距离并返回。

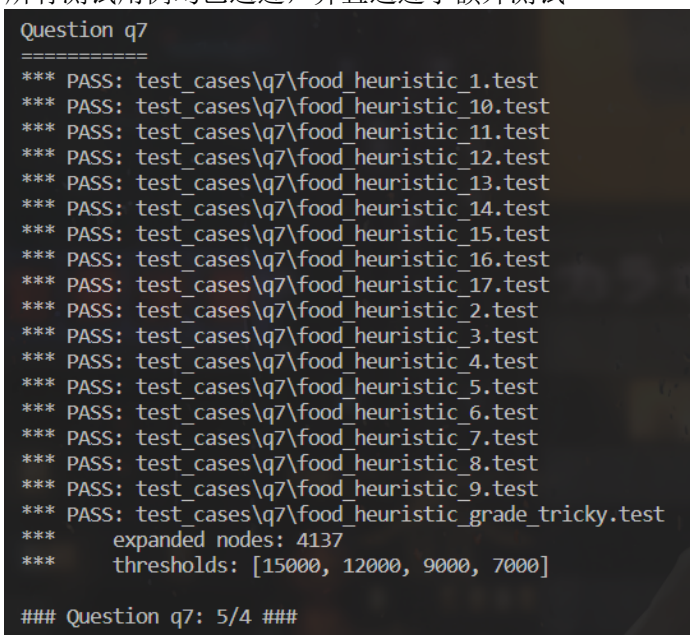
Q7.3 算法实现

下面贴出代码，细节已在注释中标明

```
1.  def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem):
2.      position, foodGrid = state
3.      "*** YOUR CODE HERE ***"
4.      # 获取食物列表
5.      food_list = foodGrid.asList()
6.      now_position = position
7.      total_dist = 0
8.
9.      # 计算当前位置到每个食物的距离，并返回其中的最大值
10.     for food_position in food_list:
11.         total_dist = max(mazeDistance(
12.             now_position, food_position, problem.startingGameState), total_dist)
13.     return total_dist
```

Q7.4 实验结果

所有测试用例均已通过，并且通过了额外测试



检查方式同 Q6。先判断启发式是否非平凡、可接受、一致，若不满足上述任一条件，则测试不通过。接下来会根据扩展结点的个数来给出评分，扩展的结点越少，评分越高。

在特殊迷宫下成功搜索

python pacman.py -l trickySearch -p AStarFoodSearchAgent

```
Path found with total cost of 60 in 10.0 seconds
Search nodes expanded: 4137
Pacman emerges victorious! Score: 570
Average Score: 570.0
Scores:        570.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Q8. Suboptimal Search——次优搜索

Q8.1 问题概述

Q8.1.1 直观描述

有时，即使有 A* 和良好的启发式算法，也很难找到通过所有点的最佳路径。 在这些情况下，我们仍然希望快速找到一条相当好的路径。在本问题中，我们需要编写一个总是贪心地吃掉最近的食物的智能体。 ClosestDotSearchAgent 已在 searchAgents.py 中实现，但它缺少一个关键函数，该函数可以找到到最近点的路径。在 searchAgents.py 中实现函数 findPathToClosestDot。

Q8.1.2 已有代码的阅读和理解

要阅读的代码即要完成的类，理解在后续贴出的代码中由注释表示。

Q8.1.3 解决问题的思路与方法

要找到离当前位置最近的食物，可以沿用之前完成的一致代价搜索，并在 AnyFoodSearchProblem 类中将有食物在的位置都视为目标位置。

Q8.2 算法设计

Q8.2.1 算法功能

找到一条到达离当前位置最近的食物的路径。

Q8.2.2 设计思路

同 Q3 ucs，并在 AnyFoodSearchProblem 类中将有食物在的位置都视为目标位置。

Q8.3 算法实现

下面贴出代码，细节已在注释中标明

```
#定义初始化函数，重写 SearchAgent 类的 registerInitialState 函数
def registerInitialState(self, state):
    self.actions = [] #初始化移动序列为空列表
    currentState = state #当前状态为初始状态
    #只要地图上还有豆子，就不断找最近的豆子并向其移动
    while(currentState.getFood().count() > 0):
        nextPathSegment = self.findPathToClosestDot(currentState) #找到到达最近豆子的移动序列
```

```
8.          self.actions += nextPathSegment #将这段移动序列添加到总移动序列中
9.          for action in nextPathSegment:
10.             legal = currentState.getLegalActions() #获取合法动作
11.             #如果当前移动不合法，则抛出异常
12.             if action not in legal:
13.                 t = (str(action), str(currentState))
14.                 raise Exception('findPathToClosestDot returned an illegal move: %
    s!\n%s' % t)
15.             currentState = currentState.generateSuccessor(0, action) #更新当前状
    态
16.     self.actionIndex = 0 #将动作索引重置为0
17.     print('Path found with cost %d.' % len(self.actions)) #输出找到路径的代价（即移
    动的步数）
18.
19. #定义函数 findPathToClosestDot，返回到达最近豆子的移动序列
20. def findPathToClosestDot(self, gameState: pacman.GameState):
21.     """
22.     Returns a path (a list of actions) to the closest dot, starting from
23.     gameState.
24.     """
25.     #以下是一些有用的初始值
26.     startPosition = gameState.getPacmanPosition() #获取吃豆人的初始位置
27.     food = gameState.getFood() #获取豆子的位置
28.     walls = gameState.getWalls() #获取墙的位置
29.     problem = AnyFoodSearchProblem(gameState) #使用 AnyFoodSearchProblem 类，返回寻
    找任何豆子的问题
30.
31.     "*** YOUR CODE HERE ***"
32.     #如果吃豆人已经在豆子上，返回空列表
33.     if problem.isGoalState(startPosition) == True:
34.         return []
35.
36.     myPriorityQueue = util.PriorityQueue() #定义一个优先队列
37.     answer = [] #初始化答案为空列表
38.     visit = [startPosition] #记录吃豆人已经到达的位置
39.
40.     #对于起点的每个可行动作，将下一步的位置、移动序列和代价加入优先队列
41.     for position, action, cost in problem.getSuccessors(startPosition):
42.         #用 cost 做 priority
43.         myPriorityQueue.push((position, [action], cost), cost)
44.
45.     #不断从队列中取出 priority 最小的元素，直到队列为空
46.     while not myPriorityQueue.isEmpty():
47.         now_position, now_path, now_cost = myPriorityQueue.pop()
48.         if now_position in visit: #如果当前位置已经被访问过，跳过
49.             continue
50.         visit.append(now_position) #将当前位置加入 visit 列表
51.
52.         #如果当前位置已经到达目标（即有豆子），更新 answer 为当前移动路径
53.         if problem.isGoalState(now_position):
54.                 answer = now_path
55.                 break
56.         # 后继结点入队
57.         for next_position, action, cost in problem.getSuccessors(now_position):
58.                 if not next_position in visit:
59.                     myPriorityQueue.push(
60.                         (next_position, now_path+[action], now_cost+cost), now_co
    st+cost)
61.
62.         return answer
```

```python
def isGoalState(self, state: Tuple[int, int]):
        x, y = state
        "*** YOUR CODE HERE ***"
        return self.food[x][y]
        util.raiseNotDefined()
```

Q8.4 实验结果

所有测试用例均以通过

```
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
```

```
*** 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
```

查看测试用例

```
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** FAIL: test_cases\q8\closest_dot_4.test
*** Closest dot not found.
***     student solution length:
0
***
***     correct solution length:
3
```

判断依据是是否找到了距离当前位置最近的食物

在大迷宫下成功搜索

python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5

```
Path found with cost 350.
Pacman emerges victorious! Score: 2360
Average Score: 2360.0
Scores:        2360.0
Win Rate:      1/1 (1.00)
Record:        Win
```

三、总结与分析

在本实验中，我们学习了深度优先搜索、广度优先搜索、一致代价搜索等基本的搜索算法，以及在实际应用中可能会用到的启发式搜索等高级算法。在实验中，我遇到了一些问题，如如何

实现启发式函数、如何解决状态空间过大的问题等。对于这些问题，我们采取了一些具体的解决方案，如采用曼哈顿距离或欧几里得距离等作为启发式函数等。

同时，我也深刻地认识到搜索算法的优缺点和适用场景。

深度优先搜索（DFS）：深度优先搜索是一种非常基础的搜索算法，其主要思想是尽可能地探索每个节点的深度，直到找到目标状态或者无法继续搜索为止。深度优先搜索的优点是空间复杂度较小，但其缺点是可能会无法找到最优解。深度优先搜索适用于状态空间较小、目标状态较深的问题。

广度优先搜索（BFS）：广度优先搜索的主要思想是按照层级的方式逐层遍历状态空间，直到找到目标状态或者遍历完整个状态空间为止。广度优先搜索的优点是可以找到最短路径，但其缺点是空间复杂度较高。广度优先搜索适用于状态空间较大、目标状态较浅的问题。

一致代价搜索（UCS）：是一种启发式搜索算法，它是基于广度优先搜索（BFS）的一种改进，可以保证找到最短路径。它的主要思想是在搜索过程中保持一个优先级队列，根据路径代价递增的顺序对节点进行扩展。UCS 是一种有效的启发式搜索算法，可以保证找到最短路径，但是在搜索空间非常大、代价不均匀的问题上表现不佳。

A*搜索：A*搜索是一种启发式搜索算法，其主要思想是在深度优先搜索或广度优先搜索的基础上，引入一个启发式函数来估算每个状态到目标状态的距离，从而选择最优的状态进行搜索。A 搜索的优点是能够找到最优解，并且可以根据问题特点灵活选择启发式函数，但其缺点是需要比较复杂的实现。A*搜索适用于具有一定结构的问题，并能够根据问题特点来选择合适的启发式函数。

在实验中，我们还完成了一些启发式函数的编写，以下是一些心得总结

1.启发式函数必须满足非平凡性、一致性、可接受性。

2.启发式函数应该尽可能地准确地估计当前状态到达目标状态的距离。这可以通过对目标状态的预测和当前状态的分析得出。例如，在八数码问题中，可以通过计算当前状态中每个数码离其正确位置的距离之和来估计当前状态到目标状态的距离。

3.启发式函数应该是快速可计算的，以便在实际搜索过程中能够高效地计算并使用。在某些情况下，启发式函数可能需要使用预处理或缓存等技术来提高计算效率。

4.启发式函数应该是可调整的，以便在需要时可以进行修改和优化。例如，可以在搜索过程中动态调整启发式函数的权重或参数，以便更好地适应不同的搜索空间和目标状态。

5.启发式函数应该能够充分利用问题的特征，例如对状态的可重复性或对状态的相似性进行分析，从而更准确地估计到目标状态的距离。

实验总得分如下：